PingGateway

June 30, 2025

PINGGATEWAY
Version: 2024.3

Table of Contents

ForgeRock Identity Gateway

ForgeRock Identity Gateway (IG) uses ForgeRock Identity Platform capabilities to protect web applications, APIs, and microservices.

Release notes

- All versions

Get started with IG

- Quick install
- Install
- Upgrade

Use IG

- With Identity Cloud
- With Access Management

Troubleshoot IG

- Maintenance
- Manage logs
- Troubleshoot

Learn More

- PingGateway community
- Support portal
- Partner portal (partners)

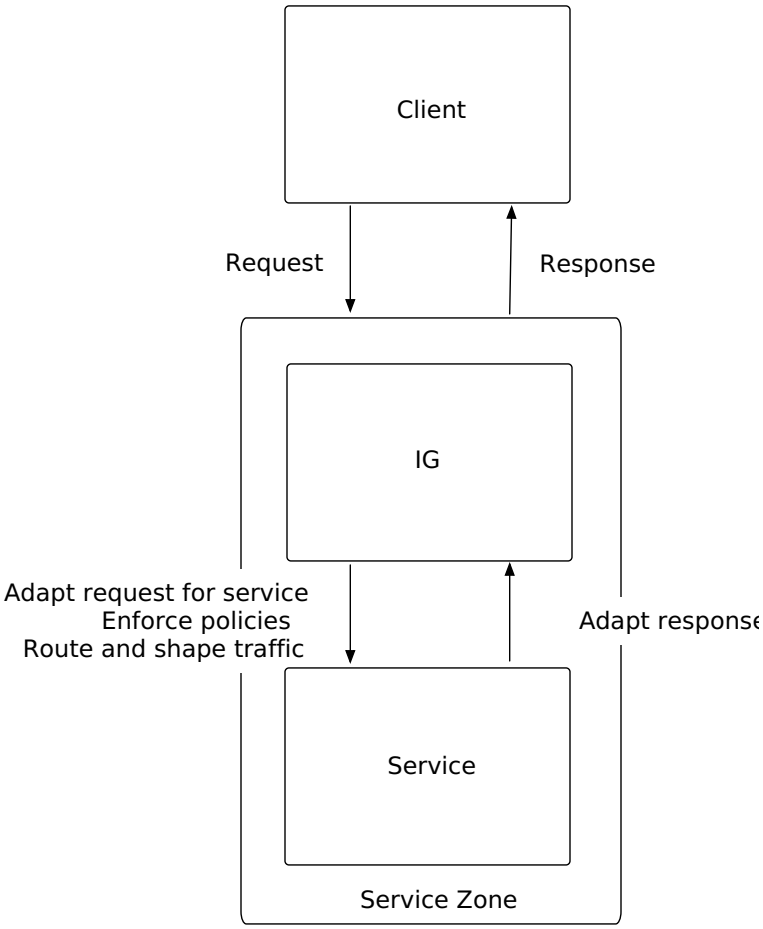
About IG

IG integrates web applications, APIs, and microservices with the ForgeRock Identity Platform. IG enforces security and access control in conjunction with Access Management modules.

This guide shows you an overview of IG.

IG as a reverse proxy

IG as a reverse proxy server is an intermediate connection point between external clients and internal services. IG intercepts client requests and server responses, enforcing policies, and routing and shaping traffic. The following image illustrates IG as a reverse proxy:



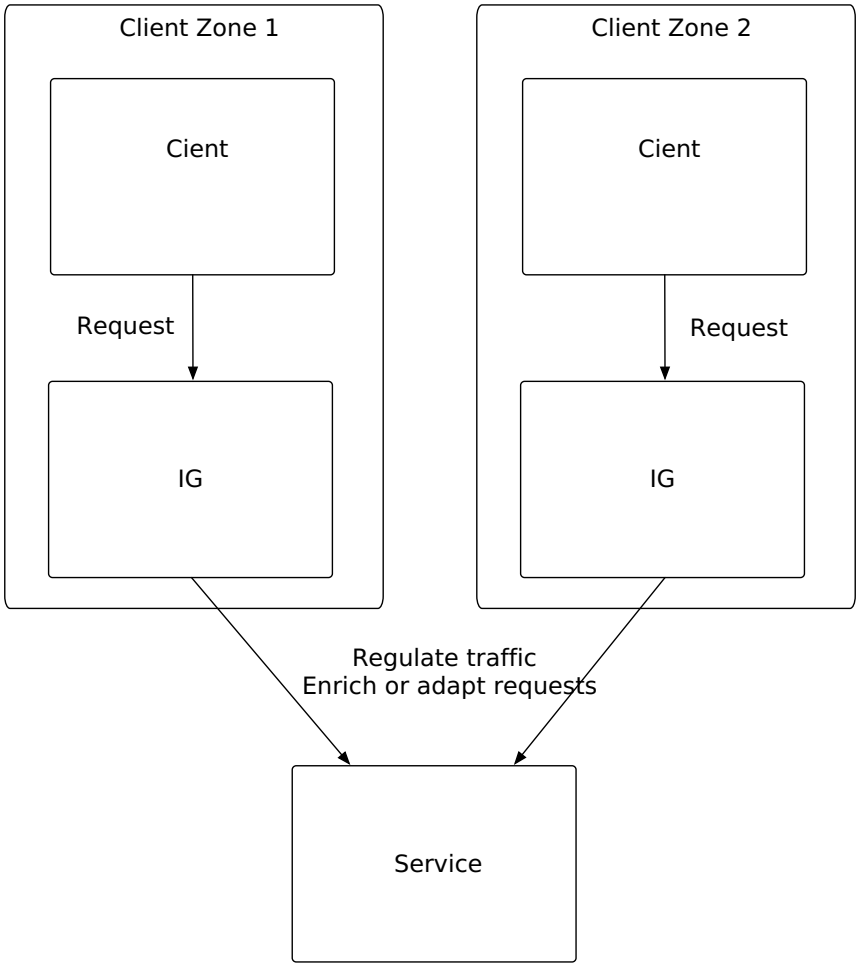
IG provides the following features as a reverse proxy:

- Access management integration
- Application and API security
- Credential replay
- OAuth 2.0 support
- OpenID Connect 1.0 support
- Network traffic control

- Proxy with request and response capture
- Request and response rewriting
- SAML 2.0 federation support
- Single sign-on (SSO)

IG as a forward proxy

In contrast, IG as a forward proxy is an intermediate connection point between an internal client and an external service. IG regulates outbound traffic to the service, and can adapt and enrich requests. The following image illustrates IG as a forward proxy:



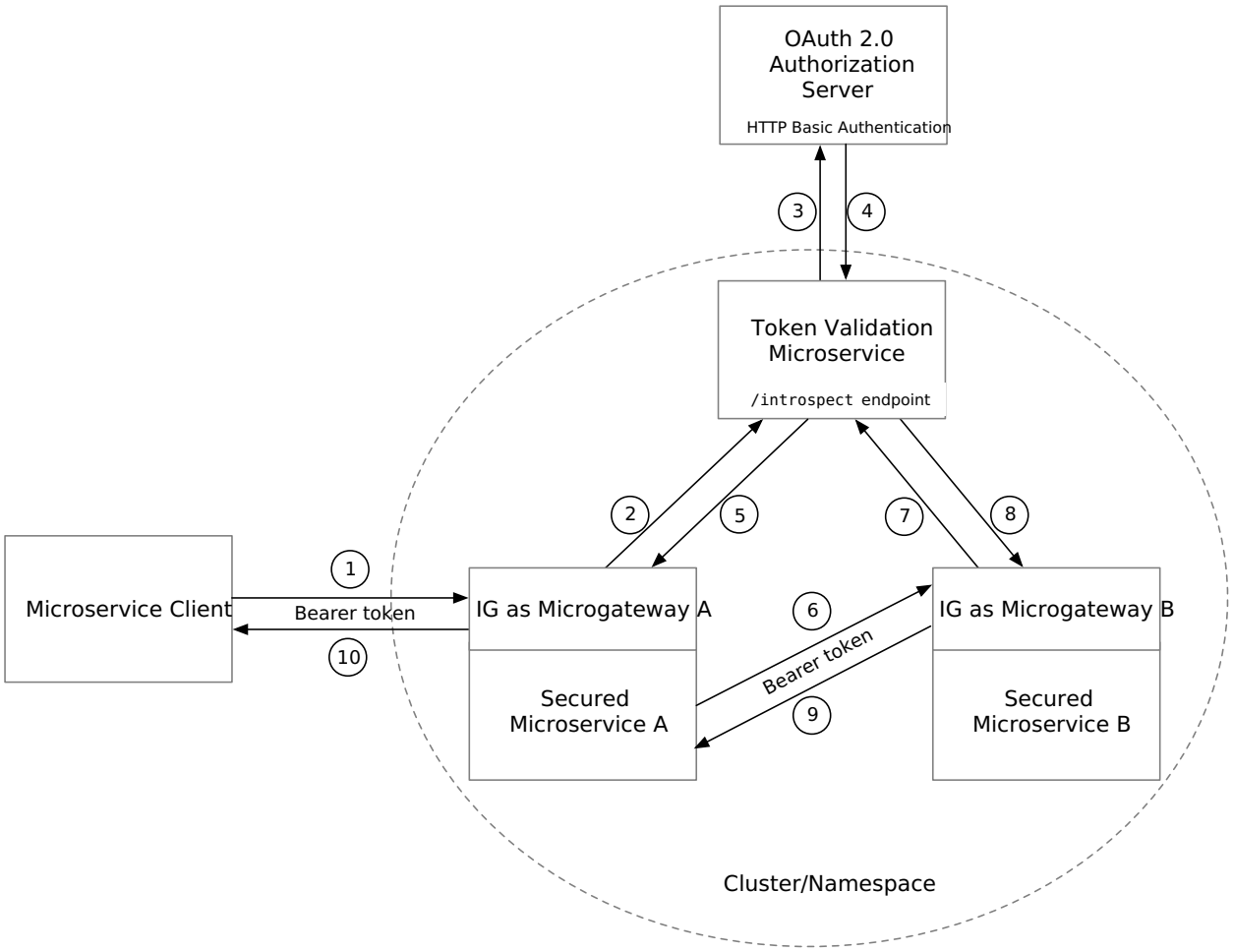
IG provides the following features as a forward proxy:

- Addition of authentication or authorization to the request
- Addition of tracer IDs to the requests
- Addition or removal of request headers or scopes

IG as a microgateway

IG is optimized to run as a microgateway in containerized environments. Use IG with business microservices to separate the security concerns of your applications from their business logic. For example, use IG with the ForgeRock Token Validation Microservice to provide access token validation at the edge of your namespace.

For an example, refer to IG as a microgateway. The following image illustrates the request flow in an example deployment:



IG processes the request in the following steps:

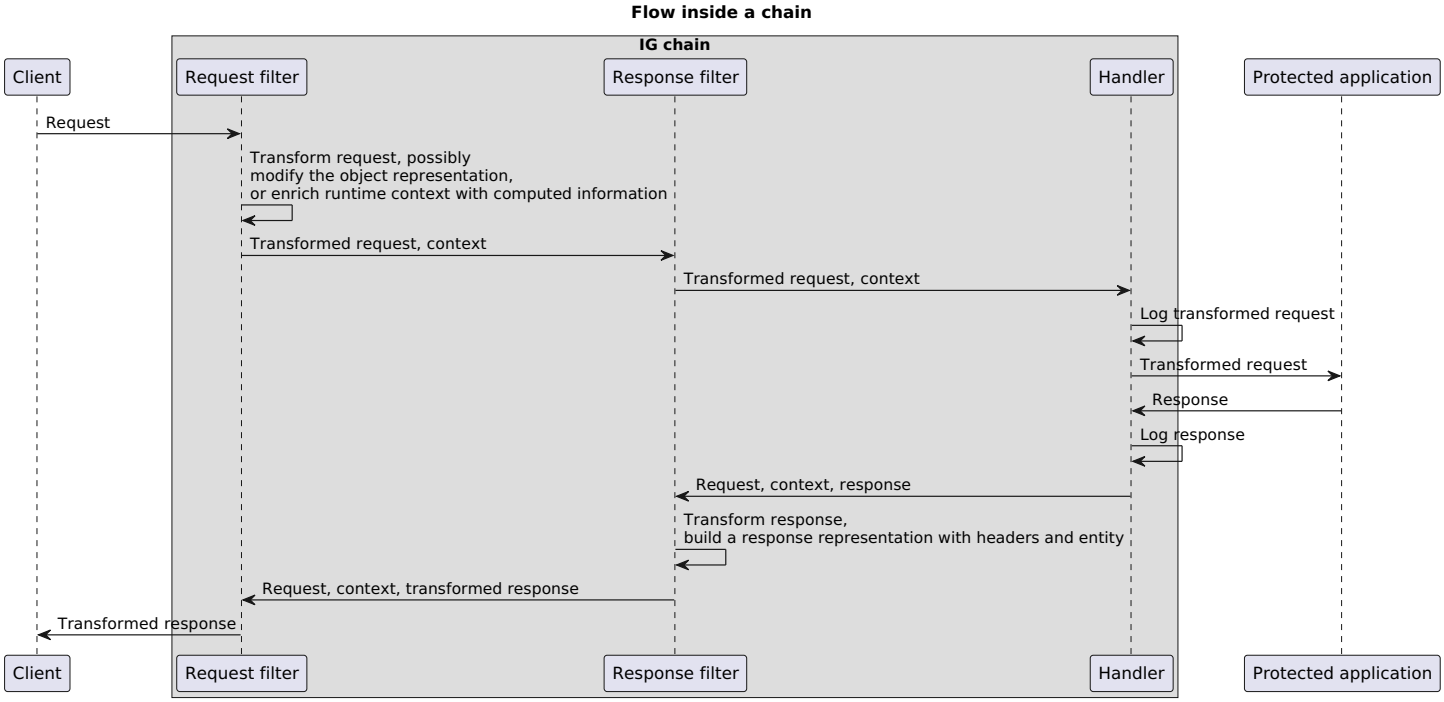
1. A client requests access to Secured Microservice A, providing a stateful OAuth 2.0 access token as credentials.
2. Microgateway A intercepts the request, and passes the access token for validation to the Token Validation Microservice, using the `/introspect` endpoint.
3. The Token Validation Microservice requests the Authorization Server to validate the token.
4. The Authorization Server introspects the token, and sends the introspection result to the Token Validation Microservice.
5. The Token Validation Microservice caches the introspection result, and sends it to Microgateway A, which forwards the result to Secured Microservice A.

6. Secured Microservice A uses the introspection result to decide how to process the request. In this case, it continues processing the request. Secured Microservice A asks for additional information from Secured Microservice B, providing the validated token as credentials.
7. Microgateway B intercepts the request, and passes the access token to the Token Validation Microservice for validation, using the `/introspect` endpoint.
8. The Token Validation Microservice retrieves the introspection result from the cache, and sends it back to Microgateway B, which forwards the result to Secured Microservice B.
9. Secured Microservice B uses the introspection result to decide how to process the request. In this case it passes its response to Secured Microservice A, through Microgateway B.
10. Secured Microservice A passes its response to the client, through Microgateway A.

Object model

IG processes HTTP requests and responses by passing them through user-defined chains of filters and handlers. The filters and handlers provide access to the request and response at each step in the chain, and make it possible to alter the request or response, and collect contextual information.

The following image illustrates a typical sequence of events when IG processes a request and response through a chain:



When IG processes a request, it first builds an object representation of the request, including parsed query/form parameters, cookies, headers, and the entity. IG initializes a runtime context to provide additional metadata about the request and applied transformations. IG then passes the request representation into the chain.

In the request flow, filters modify the request representation and can enrich the runtime context with computed information. In the ClientHandler, the entity content is serialized, and additional query parameters can be encoded as described in RFC 3986: Query.

In the response flow, filters build a response representation with headers and the entity.

The route configuration in Adding headers and logging results demonstrates the flow through a chain to a protected application.

Sessions

IG uses sessions to group requests from a user agent or other source, and collect information from the requests. When multiple requests are made in the same session, the requests can share the session information. Because session sharing is not thread-safe, it is not suitable for concurrent exchanges.

The following table compares stateful and stateless sessions:

Feature	Stateful sessions	Stateless sessions
Cookie size.	Unlimited.	Maximum size of the JWT session cookie is 4 KBytes, as defined by the browser. If the cookie exceeds this size, IG automatically splits it into multiple cookies.
Default name of the session cookie.	`IG_SESSIONID`.	`openig-jwt-session`.
Object types that can be stored in the session.	Only Java serializable objects, when sessions are replicated. Any object, when sessions are not replicated.	JSON-compatible types, such as strings, numbers, booleans, null, structures such as arrays, and list and maps containing only JSON-compatible types.
Session sharing between instances of IG, for load balancing and failover.	Possible when sessions are replicated on multiple IG instances. Possible when sessions are not replicated, if session stickiness is configured.	Possible because the session content is a cookie on the user agent, that can be copied to multiple instances of IG.
Risk of data inconsistency when simultaneous requests modify the content of a session.	Low because the session content is stored on IG and shared by all exchanges. Processing is not thread-safe.	Higher because the session content is reconstructed for each request. Concurrent exchanges don't have the same content.

Stateful sessions

When a JwtSession is not configured for a request, stateful sessions are created automatically. Session information is stored in the IG cookie, called `IG_SESSIONID` by default. When the user agent sends a request with the cookie, the request can access the session information on IG.

When a JwtSession object is configured in the route that processes a request, or in its ascending configuration (a parent route or `config.json`), the session is always stateless and can't be stateful.

When a request enters a route without a JwtSession object in the route or its ascending configuration, a stateful session is created lazily.

Session duration is defined by the `session` property in admin.json, with a default of 30 minutes.

Even if the session is empty, the session remains usable until the timeout.

When IG is not configured for session replication, any object type can be stored in a stateful session.

Because session content is stored on IG, and shared by all exchanges, when IG processes simultaneous requests in a stateful session there is low risk that the data becomes inconsistent. However, sessions are not thread-safe; different requests can simultaneously read and modify a shared session.

Session information is available in SessionContext to downstream handlers and filters. For more information, refer to SessionContext.

Considerations for clustering IG

When a stateful session is replicated on the multiple IG instances, consider the following points:

- The session can store only object types that can be serialized.
- The network latency of session replication introduces a delay that can cause the session information of two IG instances to desynchronize.
- Because the session is replicated on the clustered IG instances, it can be shared between the instances, without configuring session stickiness.
- When sessions are not shared, configure session stickiness to ensure that load balancers serve requests to the same IG instance. For more information, refer to Prepare for load balancing and failover.

Configuration of stateful sessions

To configure stateful sessions, update the `session` property of admin.json.

Stateless sessions

Stateless sessions are provided when a JwtSession object is configured in `config.json` or in a route. For more information about configuring stateless sessions, refer to JwtSession.

IG serializes stateless session information as JSON, stores it in a JWT that can be encrypted and then signed, and places the JWT in a cookie. The cookie contains all of the information about the session, including the session attributes as JSON, and a marker for the session timeout.

The maximum size of the JWT session cookie is 4 KBytes, as defined by the browser. If the cookie exceeds this size, IG automatically splits it into multiple cookies.

Only JSON-compatible object types can be stored in stateless sessions. These object types include strings, numbers, booleans, null, structures such as arrays, and list and maps containing only JSON-compatible types.

Stateless sessions are managed as follows:

- When a request enters a route with a JwtSession object in the route or its ascending configuration, IG creates the SessionContext, verifies the cookie signature, decrypts the content of the cookie, and checks that the current date is before the session timeout.
- When the request passes through the filters and handlers in the route, the request can read and modify the session content.

- When the request returns to the the point where the session was created, for example, at the entrance to a route or at `config.json`, IG updates the cookie as follows:
  - If the session content has changed, IG serializes the session, creates one or more new JWT session cookies with the new content, encrypts and then signs the cookies, assigns them an appropriate expiration time, and returns them in the response.
  - If the session is empty, IG deletes the session, creates a new JWT session cookie with an expiration time that has already passed, and returns the cookie in the response.
  - If the session content has not changed, IG does nothing.

Because the session content is stored in one or more JWT session cookies on the user agent, stateless sessions can be shared easily between IG instances. The cookies are automatically carried over in requests, and any IG instance can unpack and use the session content.

When IG processes simultaneous requests in stateless sessions, there is a high risk that the data becomes inconsistent. This is because the session content is reconstructed for each exchange, and concurrent exchanges don't have the same content.

IG does not share sessions across requests. Instead, each request has its own session objects that it modifies as necessary, writing its own session to the session cookie regardless of what other requests do.

Session information is available in SessionContext to downstream handlers and filters. For more information, refer to SessionContext.

API descriptors

Common REST endpoints in IG serve API descriptors at runtime. When you retrieve an API descriptor for an endpoint, a JSON that describes the API for that endpoint is returned.

To discover and understand APIs, use the API descriptor with a tool such as Swagger UI to generate a web page that helps you to view and test the different endpoints.

When you start IG, or add or edit routes, registered endpoint locations for the routes hosted by the main router are written in `$HOME/.openig/logs/route-system.log`, where `$HOME/.openig` is the instance directory. Endpoint locations for subroutes are written to other log files. To retrieve the API descriptor for a specific endpoint, append one of the following query string parameters to the endpoint:

- `_api`, to represent the API accessible over HTTP. This OpenAPI descriptor can be used with endpoints that are complete or partial URLs.

  The returned JSON respects the OpenAPI specification and can be consumed by Swagger tools, such as Swagger UI.

- `_crestapi`, to provide a compact representation that is independent of the transport protocol. This ForgeRock® Common REST (Common REST) API descriptor can't be used with partial URLs.

  The returned JSON respects a ForgeRock proprietary specification dedicated to describe Common REST endpoints.

For more information about Common REST API descriptors, refer to Common REST API documentation.

Retrieve API descriptors for a router

> **Important**
>
> Switch to development mode to retrieve these API descriptors.

With IG running as described in the Quick install, run the following query to generate a JSON that describes the router operations supported by the endpoint:

```
$ curl http://ig.example.com:8080/openig/api/system/objects/_router/routes\?_api

{
     "swagger": "2.0",
     "info": {
     "version": "IG version",
     "title": "IG"
     },
     "host": "0:0:0:0:0:0:0:1",
     "basePath": "/openig/api/system/objects/_router/routes",
     "tags": [{
     "name": "Routes Endpoint"
     }],
     . . .
```

Alternatively, generate a Common REST API descriptor by using the `?_crestapi` query string.

Retrieve API descriptors for the UMA service

> **Important**
>
> Switch to development mode to retrieve these API descriptors.

With the UMA tutorial running as described in UMA support, run the following query to generate a JSON that describes the UMA share API:

```
$ curl http://ig.example.com:8080/openig/api/system/objects/_router/routes/00-uma/objects/umaservice/share\?_api

{
     "swagger": "2.0",
     "info": {
     "version": "IG version",
     "title": "IG"
     },
     "host": "0:0:0:0:0:0:0:1",
     "basePath": "/openig/api/system/objects/_router/routes/00-uma/objects/umaservice/share",
     "tags": [{
     "name": "Manage UMA Share objects"
     }],
     . . .
```

Alternatively, generate a Common REST API descriptor by using the `?_crestapi` query string.

Retrieve API descriptors for the main router

Run a query to generate a JSON that describes the API for the main router and its subsequent endpoints. For example:

```
$ curl http://ig.example.com:8080/openig/api/system/objects/_router\?_api

{
     "swagger": "2.0",
     "info": {
     "version": "IG version",
     "title": "IG"
     },
     "host": "ig.example.com:8080",
     "basePath": "/openig/api/system/objects/_router",
     "tags": [{
     "name": "Monitoring endpoint"
     }, {
     "name": "Manage UMA Share objects"
     }, {
     "name": "Routes Endpoint"
     }],
     . . .
```

Because the above URL is a partial URL, you cannot use the `?_crestapi` query string to generate a Common REST API descriptor.

Retrieve API descriptors for an IG instance

Run a query to generate a JSON that describes the APIs provided by the IG instance that's responding to a request. For example:

```
$ curl http://ig.example.com:8080/openig/api\?_api

{
     "swagger": "2.0",
     "info": {
     "version": "IG version",
     "title": "IG"
     },
     "host": "ig.example.com:8080",
     "basePath": "/openig/api",
     "tags": [{
     "name": "Internal Storage for UI Models"
     }, {
     "name": "Monitoring endpoint"
     }, {
     "name": "Manage UMA Share objects"
     }, {
     "name": "Routes Endpoint"
     }, {
     "name": "Server Info"
     }],
     . . .
```

If routes are added after the request is performed, they aren't included in the returned JSON.

Because the above URL is a partial URL, you can't use the `?_crestapi` query string to generate a Common REST API descriptor.

Quick install

Use this guide to get a quick, hands-on look at what IG software can do. You will download, install, and use IG on your local computer. For more installation options, refer to Install.

This guide assumes familiarity with the following topics:

- HTTP, including how clients and servers exchange messages, and the role that a reverse proxy (gateway) plays
- JSON, the format for IG configuration files
- Managing services on operating systems and application servers
- Configuring network connections on operating systems

Download IG

The .zip file unpacks into a /path/to/identity-gateway-2024.3.0 directory with the following content:

- `bin`: Start and stop executables
- `classes`: Initially empty; used to install patches from ForgeRock support
- `docker/Dockerfile`: Dockerfile and README to build an IG Docker image
- `legal-notices`: Licenses and copyrights
- `lib`: IG and third-party libraries

1. Create a local installation directory for IG. The examples in this section use `/path/to`.

   > **Important**
   >
   > The installation directory should be a new, empty directory. Installing IG into an existing installation directory can cause errors.

2. Download `IG-2024.3.0.zip` from the Backstage download site, and copy the .zip file to the installation directory:

   ```
   $ cp IG-2024.3.0.zip /path/to/IG-2024.3.0.zip
   ```

3. Unzip the file:

   ```
   $ unzip IG-2024.3.0.zip
   ```

   The directory /path/to/identity-gateway-2024.3.0 is created.

Prepare the network

Configure the network to include hosts for IG, AM, and the sample application. Learn more about host files from the Wikipedia entry, Hosts (file).

1. Add the following entry to your host file:

Linux

```
/etc/hosts
```

Windows

```
%SystemRoot%\system32\drivers\etc\hosts
```

```
127.0.0.1  localhost ig.example.com app.example.com am.example.com
```

Start and stop IG

Start IG with default settings

Use the following step to start the instance of IG, specifying the configuration directory where IG looks for configuration files.

1. Start IG:

Linux

```
$ /path/to/identity-gateway-2024.3.0/bin/start.sh

...
... started in 1234ms on ports : [8080 8443]
```

Windows

```
C:\path\to\identity-gateway-2024.3.0\bin\start.bat
```

By default, IG configuration files are located under `$HOME/.openig` (on Windows `%appdata%\OpenIG`). For information about how to use a different location, refer to Configuration location.

2. Check that IG is running in one of the following ways:
   - Ping IG at `http://ig.example.com:8080/openig/ping` and make sure an `HTTP 200` is returned.
   - Display the product version and build information at `http://ig.example.com:8080/openig/api/info`.

Stop IG

Use the `stop.sh` script to stop an instance of IG, specifying the instance directory as an argument. If the instance directory isn't specified, IG uses the default instance directory:

Linux

```
$ /path/to/identity-gateway-2024.3.0/bin/stop.sh $HOME/.openig
```

Windows

```
C:\path\to\identity-gateway-2024.3.0\bin\stop.bat %appdata%\OpenIG
```

Use the sample application

ForgeRock provides a mockup web application for testing IG configurations. The sample application is used in the examples in this guide and the *Gateway guide*.

Download the sample application

1. Download `IG-sample-application-2024.3.0-jar-with-dependencies.jar`, from the Backstage download site.

Start the sample application

1. Start the sample application:

```
$ java -jar IG-sample-application-2024.3.0-jar-with-dependencies.jar

[...] [INFO   ] All 8 verticles started on ports : [8081, 8444]
[...] [INFO   ] Using session timeout of: 60s
[...] [INFO   ] Using AM base for OpenID Discovery URL: http://am.example.com:8088/openam/oauth2
[...] [INFO   ] Press Ctrl+C to stop the server.
```

   By default this server listens for HTTP on port 8081, and for HTTPS on port 8444. If one or both of those ports are not free, specify other ports:

```
$ java -jar IG-sample-application-2024.3.0-jar-with-dependencies.jar 8888 8889
```

2. Check that the sample application is running in one of the following ways:

   1. Go to http://localhost:8081/home to access the home page of the sample application. Information about the browser request is displayed.

   2. Go to http://localhost:8081/login to access the login page of the sample application, and then log in with username `demo` and password `Ch4ng31t`. The username and some information about the browser request is displayed.

Stop the sample application

1. In the terminal where the sample application is running, select CTRL+C to stop the app.

Serve static resources

When the sample application is used with IG in examples, it must serve static resources, such as the .css. Similarly, browser requests often serve resources that don't need protection, such as icons and .gif files.

Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

Configuration options

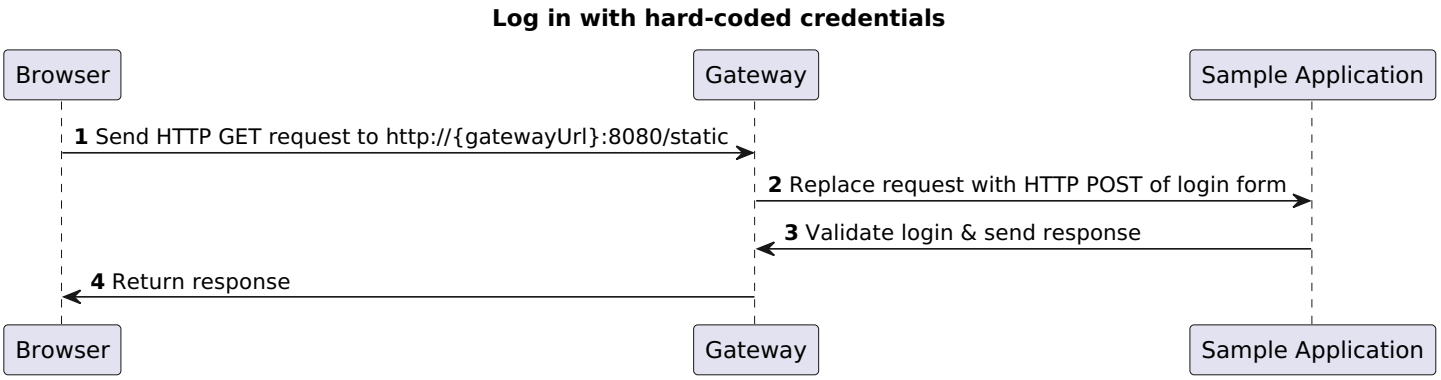
To view the command-line options for the sample application, start it with the `-h` option:

```
$ java -jar IG-sample-application-2024.3.0-jar-with-dependencies.jar -h

Usage: <main class> [options]
Options:
--http
The HTTP port number.
Default: 8081
--https
The HTTPS port number.
Default: 8444
--session
The session timeout in seconds.
Default: 60
--am-discovery-url
The AM URL base for OpenID Provider Configuration.
Default: http://am.example.com:8088/openam/oauth2
-h, --help
Default: false
```

Protect an application with IG

This section gives a simple example of how to use IG to protect an application. For many more examples of how to protect applications with IG, refer to the Gateway guide.

In the following example, a browser requests access to the sample application, and IG intercepts the request to log the user into the application. The following image shows the flow of data in the example:



1. The browser sends an HTTP GET request to the HTTP server on `ig.example.com`.
2. IG replaces the HTTP GET request with an HTTP POST login request containing credentials to authenticate.
3. The sample application validates the credentials, and returns the page for the user `demo`.

   If IG did not provide the credentials, or if the sample application couldn't validate the credentials, the sample application returns the login page.
4. IG returns this response to the browser.

Configure IG to log you in to an application

1. Set up IG and the sample application as described in this guide.
2. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

3. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/01-static.json
```

Windows

```
%appdata%\OpenIG\config\routes\01-static.json
```

```
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "StaticRequestFilter",
          "config": {
            "method": "POST",
            "uri": "http://app.example.com:8081/login",
            "form": {
              "username": [
                "demo"
              ],
              "password": [
                "Ch4ng31t"
              ]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.path, '^/static')}"
}
```

Notice the following features of the route:

- The route matches requests to `/static`.
- The StaticRequestFilter replaces the request with an HTTP POST, specifying the resource to post the request to, and a form to include in the request. The form includes credentials for the username `demo`.
- The ReverseProxyHandler replays the request to the sample application.

4. Check that the route system log includes a message that the new files are loaded into the config:

```
INFO  o.f.o.handler.router.RouterHandler - Loaded the route with id '00-static-resources' registered with the
name '00-static-resources'
INFO  o.f.o.handler.router.RouterHandler - Loaded the route with id '01-static' registered with the name '01-
static'
```

5. Go to http://ig.example.com:8080/static.

   You are directed to the sample application, and logged in automatically with the username `demo`.

Next steps

This section describes some basic options to help you with IG. For information about other installation options, refer to the Installation guide.

Add a base configuration file

The entry point for requests coming in to IG is a JSON-encoded configuration file, expected by default at:

Linux

```
$HOME/.openig/config/config.json
```

Windows

```
%appdata%\OpenIG\config\config.json
```

The base configuration file initializes a heap of objects and provides the main handler to receive incoming requests. Configuration in the file is inherited by all applicable objects in the configuration.

At startup, if IG doesn't find a base configuration file, it provides a default version, given in Default configuration. The default looks for routes in:

Linux

```
$HOME/.openig/config/routes
```

Windows

```
%appdata%\OpenIG\config\routes
```

Consider adding a custom `config.json` for these reasons:

- To prevent using the default `config.json`, whose configuration might not be appropriate in your deployment.
- To define an object once in `config.json`, and then use it multiple times in your configuration.

After adding or editing `config.json`, stop and restart IG to take the changes into effect.

For more information, refer to GatewayHttpApplication (config.json), Heap objects, and Router.

Add a base configuration for IG

1. Add the following file to IG:

Linux

```
$HOME/.openig/config/config.json
```

Windows

```
%appdata%\OpenIG\config\config.json
```

```
{
  "handler": {
    "type": "Router",
    "name": "_router",
    "baseURI": "http://app.example.com:8081",
    "capture": "all"
  },
  "heap": [
    {
      "name": "JwtSession",
      "type": "JwtSession"
    },
    {
      "name": "capture",
      "type": "CaptureDecorator",
      "config": {
        "captureEntity": true,
        "_captureContext": true
      }
    }
  ]
}
```

Notice the following features of the file:

- The handler contains a main router named `_router`. When IG receives an incoming request, `_router` routes the request to the first route in the configuration whose condition is satisfied.
- The `baseURI` changes the request URI to point the request to the sample application.
- The `capture` captures the body of the HTTP request and response.
- The JwtSession object in the heap can be used in routes to store the session information as JSON Web Tokens (JWT) in a cookie. For more information, refer to JwtSession.

2. Stop and restart IG.
3. Check that the route system log includes a message that the file is loaded into the config:

```
INFO  o.f.openig.web.Initializer - Reading the configuration from ...config.json
```

Add a default route

When there are multiple routes in the IG configuration, they are ordered lexicographically, by route name. For example, `01-static.json` is ordered before `zz-default.json`.

When IG processes a request, the request traverses the routes in the configuration. If the request matches the condition for `01-static.json` it is processed by that route. Otherwise, it passes to the next route in the configuration. If a route has no condition, it can process any request.

A default route is the last route in a configuration to which a request is routed. If a request matches no other route in the configuration, it is processed by the default route.

Add a default route to prevent errors described in No handler to dispatch to.

1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/zz-default.json
```

Windows

```
%appdata%\OpenIG\config\routes\zz-default.json
```

```
{
  "handler": "ReverseProxyHandler"
}
```

Notice the following features of the route:

- The route name starts with `zz`, so it is the last route that is loaded into the configuration.
- There is no `condition` property, so the route processes all requests.
- The route calls a ReverseProxyHandler with the default configuration, which proxies the request to the application and returns the response, without changing either the request or the response.

2. Check that the route system log includes a message that the file is loaded into the config:

```
INFO  o.f.o.handler.router.RouterHandler - Loaded the route with id
'zz-default' registered with the name 'zz-default'
```

Switch from production mode to development mode

To prevent unwanted changes to the configuration, IG is by default in production mode after installation. For a description of the modes and information about switching between modes, refer to Operating modes.

Use IG Studio

IG Studio is a user interface to help you build and deploy your IG configuration. For more information, refer to the Studio guide.

Install

This guide shows you how to install and remove IG software. For information about how to install IG for evaluation, refer to the Quick install.

Read the Release notes before you install.

Prepare to install

Before you install, make sure your installation meets the requirements in the release notes.

Create an IG service account

To limit the impact of a security breach, install and run IG from a dedicated service account. This is optional when evaluating IG, but essential in production installations.

A hacker is constrained by the rights granted to the user account where IG runs; therefore, never run IG as root user.

1. In a terminal window, use a command similar to the following to create a service account:

Linux

```
$ sudo /usr/sbin/useradd \
--create-home \
--comment "Account for running IG" \
--shell /bin/bash IG
```

Windows

```
> net user username password /add /comment:"Account for running IG"
```

2. Apply the principle of least privilege to the account, for example:
   - Read/write permissions on the installation directory, /path/to/identity-gateway-2024.3.0.
   - Execute permissions on the scripts in the installation `bin` directory, `/path/to/identity-gateway-2024.3.0/bin`.

Prepare the network

Configure the network to include hosts for IG, AM, and the sample application. Learn more about host files from the Wikipedia entry, Hosts (file).

1. Add the following entry to your host file:

Linux

```
/etc/hosts
```

Windows

```
%SystemRoot%\system32\drivers\etc\hosts
```

```
127.0.0.1  localhost ig.example.com app.example.com am.example.com
```

Set up Identity Cloud

This documentation contains procedures for setting up items in ForgeRock Identity Cloud that you can use with IG. For more information about setting up Identity Cloud, refer to the ForgeRock Identity Cloud docs.

Authenticate an IG agent to Identity Cloud

> **Important**
>
> IG agents are automatically authenticated to Identity Cloud by a non-configurable authentication module.
> Authentication chains and modules are deprecated in Identity Cloud and replaced by journeys.
> You can now authenticate IG agents to Identity Cloud with a journey. The procedure is currently optional, but will be required when authentication chains and modules are removed in a future release of Identity Cloud.
> For more information, refer to Identity Cloud's Journeys.

This section describes how to create a journey to authenticate an IG agent to Identity Cloud. The journey has the following requirements:

- It must be called `Agent`
- Its nodes must pass the agent credentials to the Agent Data Store Decision node.

When you define a journey in Identity Cloud, that same journey is used for all instances of IG, Java agent, and Web agent. Consider this point if you change the journey configuration.

1. Log in to the Identity Cloud admin UI as an administrator.
2. Click **Journeys** > **New Journey**.
3. Add a journey with the following information and click **Create journey**:
   - **Name**: `Agent`
   - **Identity Object**: The user or device to authenticate.

- (Optional) **Description**: Authenticate an IG agent to Identity Cloud

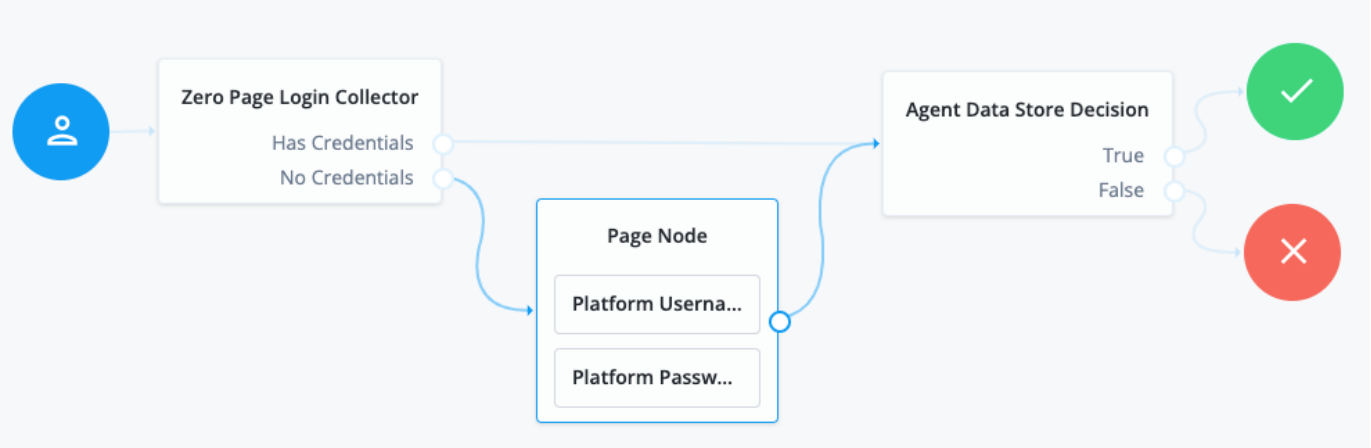
The journey designer is displayed, with the `Start` entry point connected to the `Failure` exit point, and a `Success` node.

4. Using the **Filter nodes** bar, find and then drag the following nodes from the **Components** panel into the designer area:
   - Zero Page Login Collector node to check whether the agent credentials are provided in the incoming authentication request, and use their values in the following nodes.

     This node is required for compatibility with Java agent and Web agent.
   - Page node to collect the agent credentials if they are not provided in the incoming authentication request, and use their values in the following nodes.
   - Agent Data Store Decision node to verify the agent credentials match the registered IG agent profile.

> **Important**
>
> Many nodes can be configured in the panel on the right side of the page. Unless otherwise stated, do not configure the nodes, and use only the default values.

5. Drag the following nodes from the **Components** panel into the Page node:
   - Platform Username node to prompt the user to enter their username.
   - Platform Password node to prompt the user to enter their password.
6. Connect the nodes as follows and save the journey:



Register an IG agent in Identity Cloud

This procedure registers an agent that acts on behalf of IG.

1. Log in to the Identity Cloud admin UI as an administrator.
2. Click **Gateways & Agents** > **New Gateway/Agent** > **Identity Gateway** > **Next**, and add an agent profile:
   - **ID**: agent-name
   - **Password**: agent-password
   - **Redirect URLs**: URL for CDSSO

> **Important**
>
> Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

3. Click **Save Profile** > **Done**. The agent profile page is displayed.
4. Click **Native Consoles** > **Access Management** and make the following optional changes in the AM admin UI.

Change	Action
Store the agent password in AM's secret service.	Set a **Secret Label Identifier**, and configure a mapping to the corresponding secret. If AM finds a matching secret in a secret store, it uses that secret instead of the agent password configured in Step 2. The secret label has the format `am.application.agents.identifier.secret`, where identifier is the **Secret Label Identifier**. The **Secret Label Identifier** can contain only characters `a-z`, `A-Z`, `0-9`, and periods (`.`). It can't start or end with a period. Note the following points: ◦ Set a **Secret Label Identifier** that clearly identifies the agent. ◦ If you update or delete the **Secret Label Identifier**, AM updates or deletes the corresponding mapping for the previous identifier provided no other agent shares the mapping. ◦ When you rotate a secret, update the corresponding mapping.
Direct login to a custom URL instead of the default AM login page.	Configure **Login URL Template for CDSSO**.
Apply a different introspection scope.	Click **Token Introspection** and select a scope from the drop-down list.

Set up a demo user in Identity Cloud

This procedure sets up a demo user in the alpha realm.

1. Log in to the Identity Cloud admin UI as an administrator.
2. Go to **Identities** > **Manage** > **Alpha realm - Users**, and add a user with the following values:
   - **Username**: `demo`
   - **First name**: `demo`
   - **Last name**: `user`
   - **Email Address**: `demo@example.com`
   - **Password**: `Ch4ng3!t`

Set up AM

This documentation contains procedures for setting up items in AM that you can use with IG. For more information about setting up AM, refer to the Access Management docs.

Authenticate an IG agent to AM

Important

From AM 7.3

When AM 7.3 is installed with a default configuration, as described in Evaluation, IG is automatically authenticated to AM by an authentication tree. Otherwise, IG is authenticated to AM by an AM authentication module.

Authentication chains and modules were deprecated in AM 7. When they are removed in a future release of AM, it will be necessary to configure an appropriate authentication tree when you are not using the default configuration.

For more information, refer to AM's Authentication Nodes and Trees.

This section describes how to create an authentication tree to authenticate an IG agent to AM. The tree has the following requirements:

- It must be called `Agent`
- Its nodes must pass the agent credentials to the Agent Data Store Decision node.

When you define a tree in AM, that same tree is used for all instances of IG, Java agent, and Web agent. Consider this point if you change the tree configuration.

1. On the **Realms** page of the AM admin UI, choose the realm in which to create the authentication tree.
2. On the **Realm Overview** page, click **Authentication** > **Trees** > **+ Create tree**.
3. Create a tree named `Agent`.

   The authentication tree designer is displayed, with the `Start` entry point connected to the `Failure` exit point, and a `Success` node.

   The authentication tree designer provides the following features on the toolbar:

Button	Usage
	Lay out and align nodes according to the order they are connected.
	Toggle the designer window between normal and full-screen layout.
	Remove the selected node. Note that the `Start` entry point cannot be deleted.

4. Using the Filter bar, find and then drag the following nodes from the **Components** panel into the designer area:

   - Zero Page Login Collector node to check whether the agent credentials are provided in the incoming authentication request, and use their values in the following nodes.

     This node is required for compatibility with Java agent and Web agent.

   - Page node to collect the agent credentials if they are not provided in the incoming authentication request, and use their values in the following nodes.

   - Agent Data Store Decision node to verify the agent credentials match the registered IG agent profile.

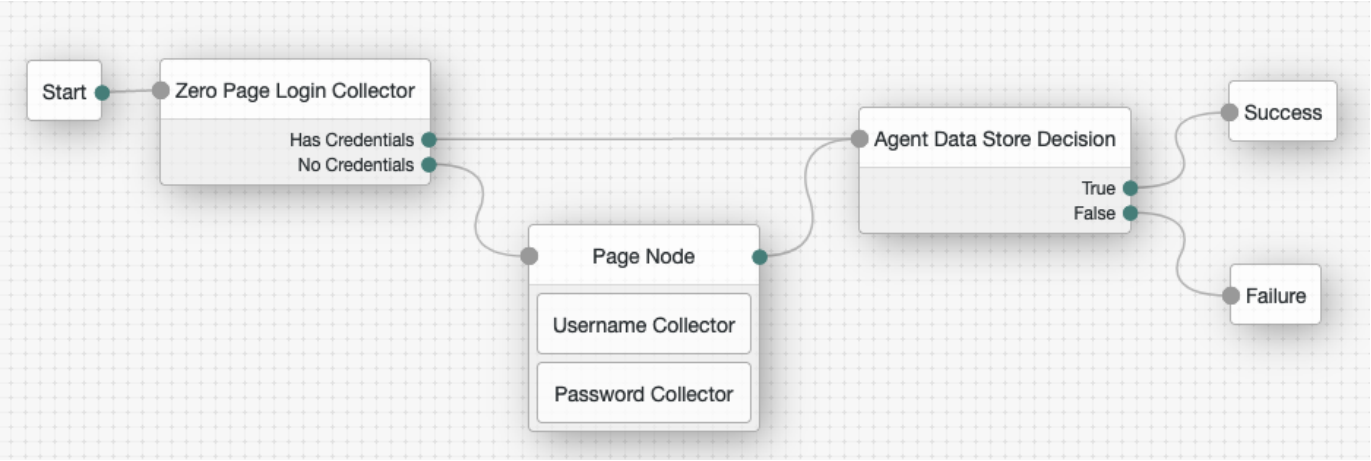
> **Important**
>
> Many nodes can be configured in the panel on the right side of the page. Unless otherwise stated, do not configure the nodes and use only the default values.

5. Drag the following nodes from the **Components** panel into the Page node:

   - Username Collector node to prompt the user to enter their username.

   - Password Collector node to prompt the user to enter their password.

6. Connect the nodes as follows and save the tree:



Register an IG agent in AM

In AM 7 and later versions, follow these steps to register an agent that acts on behalf of IG.

1. In the AM admin UI, select the top-level realm, and then select **Applications** > **Agents** > **Identity Gateway**.

2. Add an agent with the following configuration, leaving other options blank or with the default value:

For SSO

- **Agent ID** : `ig_agent`
- **Password** : `password`

For CDSSO

- **Agent ID** : `ig_agent`
- **Password** : `password`
- **Redirect URL for CDSSO** : `https://ig.ext.com:8443/home/cdsso/redirect`
- **Login URL Template for CDSSO**: Configure this property to direct login to a custom URL instead of the default AM login page.

3. (Optional - From AM 7.5) Use AM's secret service to manage the agent profile password. If AM finds a matching secret in a secret store, it uses that secret instead of the agent password configured in Step 2.

   1. In the agent profile page, set a label for the agent password in **Secret Label Identifier**.

      AM uses the identifier to generate a secret label for the agent.

      The secret label has the format `am.application.agents.identifier.secret`, where identifier is the **Secret Label Identifier**.

      The **Secret Label Identifier** can only contain characters `a-z`, `A-Z`, `0-9`, and periods (`.`). It can't start or end with a period.

   2. Select **Secret Stores** and configure a secret store.
   3. Map the label to the secret. Learn more from AM's mapping.

   Note the following points for using AM's secret service:

   - Set a **Secret Label Identifier** that clearly identifies the agent.
   - If you update or delete the **Secret Label Identifier**, AM updates or deletes the corresponding mapping for the previous identifier provided no other agent shares the mapping.
   - When you rotate a secret, update the corresponding mapping.

Set up a demo user in AM

AM is provided with a demo user in the top-level realm, with the following credentials:

- ID/username: `demo`
- Last name: `user`

- Password: `Ch4ng31t`
- Email address: `demo@example.com`
- Employee number: `123`

For information about how to manage identities in AM, refer to AM's Identity stores.

Find the AM session cookie name

In routes that use AmService, IG retrieves AM's SSO cookie name from the `ssoTokenHeader` property or from AM's `/serverinfo/*` endpoint.

In other circumstances where you need to find the SSO cookie name, access `http://am-base-url/serverinfo/*`. For example, access the AM endpoint with `curl`:

```
$ curl http://am.example.com:8088/openam/json/serverinfo/*
```

Download IG

The .zip file unpacks into a /path/to/identity-gateway-2024.3.0 directory with the following content:

- `bin`: Start and stop executables
- `classes`: Initially empty; used to install patches from ForgeRock support
- `docker/Dockerfile`: Dockerfile and README to build an IG Docker image
- `legal-notices`: Licenses and copyrights
- `lib`: IG and third-party libraries

1. Create a local installation directory for IG. The examples in this section use `/path/to`.

   > **Important**
   >
   > The installation directory should be a new, empty directory. Installing IG into an existing installation directory can cause errors.

2. Download `IG-2024.3.0.zip` from the Backstage download site, and copy the .zip file to the installation directory:

   ```
   $ cp IG-2024.3.0.zip /path/to/IG-2024.3.0.zip
   ```

3. Unzip the file:

   ```
   $ unzip IG-2024.3.0.zip
   ```

   The directory /path/to/identity-gateway-2024.3.0 is created.

Start and stop IG

Start IG with default settings

Use the following step to start the instance of IG, specifying the configuration directory where IG looks for configuration files.

1. Start IG:

Linux

```
$ /path/to/identity-gateway-2024.3.0/bin/start.sh

...
... started in 1234ms on ports : [8080 8443]
```

Windows

```
C:\path\to\identity-gateway-2024.3.0\bin\start.bat
```

By default, IG configuration files are located under `$HOME/.openig` (on Windows `%appdata%\OpenIG`). For information about how to use a different location, refer to Configuration location.

2. Check that IG is running in one of the following ways:
   - Ping IG at `http://ig.example.com:8080/openig/ping` and make sure an `HTTP 200` is returned.
   - Display the product version and build information at `http://ig.example.com:8080/openig/api/info`.

Start IG with custom settings

By default, IG runs on HTTP, on port `8080`, from the instance directory `$HOME/.openig`.

To start IG with custom settings, add the configuration file `admin.json` with the following properties, and restart IG:

- `vertx`: Finely tune Vert.x instances.
- `connectors`: Customize server port, TLS, and Vert.x-specific configurations. Each `connectors` object represents the configuration of an individual port.
- `prefix`: Set the instance directory, and therefore, the base of the route for administration requests.

The following example starts IG on non-default ports, and configures Vert.x-specific options for the connection on port 9091:

```
{
  "connectors": [{
    "port": 9090
  },
  {
    "port": 9091,
    "vertx": {
      "maxWebSocketFrameSize": 128000,
      "maxWebSocketMessageSize": 256000,
      "compressionLevel": 4
    }
  }]
}
```

For more information, refer to AdminHttpApplication (admin.json).

Allow startup when there is an existing PID file

By default, if there is an existing PID file during startup the startup fails. Use one of the following ways to allow startup when there is an existing PID file. IG then removes the existing PID file and creates a new one during startup.

1. Add the following configuration to `admin.json` and restart IG:

   ```
   {
     "pidFileMode": "override"
   }
   ```

2. Define an environment variable for the configuration token `ig.pid.file.mode`, and then start IG in the same terminal:

   Linux

   ```
   $ IG_PID_FILE_MODE=override /path/to/identity-gateway-2024.3.0/bin/start.sh
   ```

   Windows

   ```
   C:\IG_PID_FILE_MODE=override
   C:\path\to\identity-gateway-2024.3.0\bin\start.bat %appdata%\OpenIG
   ```

3. Define a system property for the configuration token `ig.pid.file.mode` when you start IG:

Linux

```
$HOME/.openig/env.sh
```

Windows

```
%appdata%\OpenIG\env.sh
```

```
export "IG_OPTS=-Dig.pid.file.mode=override"
```

Stop IG

Use the `stop.sh` script to stop an instance of IG, specifying the instance directory as an argument. If the instance directory isn't specified, IG uses the default instance directory:

Linux

```
$ /path/to/identity-gateway-2024.3.0/bin/stop.sh $HOME/.openig
```

Windows

```
C:\path\to\identity-gateway-2024.3.0\bin\stop.bat %appdata%\OpenIG
```

Set up environment variables and system properties

Configure environment variables and system properties as follows:

- By adding environment variables on the command line when you start IG.
- By adding environment variables in `$HOME/.openig/bin/env.sh`, where `$HOME/.openig` is the instance directory. After changing `env.sh`, restart IG to load the new configuration.

Start IG with a customized router scan interval

By default, IG scans every 10 seconds for changes to the route configuration files. Any changes to the files are automatically loaded into the configuration without restarting IG. For more information about the router scan interval, refer to Router.

The following example overwrites the default value of the Router scan interval to two seconds when you start up IG:

Linux

```
$ IG_ROUTER_SCAN_INTERVAL='2 seconds' /path/to/identity-gateway-2024.3.0/bin/start.sh
```

Windows

```
C:\IG_ROUTER_SCAN_INTERVAL='2 seconds'
C:\start.bat %appdata%\OpenIG
```

Define environment variables for startup, runtime, and stop

IG provides the following environment variables for Java runtime options:

IG_OPTS

(Optional) Java runtime options for IG and its startup process, such as JVM memory sizing options.

Include all options that are not shared with the `stop` script.

The following example specifies environment variables in the `env.sh` file to customize JVM options and keys:

Linux

```
# Specify JVM options
JVM_OPTS="-Xms256m -Xmx2048m"

# Specify the DH key size for stronger ephemeral DH keys, and to protect against weak keys
JSSE_OPTS="-Djdk.tls.ephemeralDHKeySize=2048"

# Wrap them up into the IG_OPTS environment variable
export IG_OPTS="${IG_OPTS} ${JVM_OPTS} ${JSSE_OPTS}"
```

Windows

```
C:\set "JVM_OPTS=-Xms256m -Xmx2048m"
C:\set "JSSE_OPTS=-Djdk.tls.ephemeralDHKeySize=2048"
C:\set "IG_OPTS=%IG_OPTS% %JVM_OPTS% %JSSE_OPTS%"
```

JAVA_OPTS

(Optional) Java runtime options for IG include all options that are shared by the `start` and `stop` script.

Encrypt and share JWT sessions

JwtSession objects store session information in JWT cookies on the user agent. The following sections describe how to set authenticated encryption for JwtSession, using symmetric keys.

Authenticated encryption encrypts data and then signs it with HMAC, in a single step. For more information, refer to Authenticated Encryption. For information about JwtSession, refer to JwtSession.

Encrypt JWT sessions

This section describes how to set up a keystore with a symmetric key for authenticated encryption of a JWT session.

1. Set up a keystore to contain the encryption key, where the keystore and the key have the password `password`:

    1. Locate a directory for secrets, and go to it:

    ```
    $ cd /path/to/secrets
    ```

    2. Generate the key:

    ```
    $ keytool \
      -genseckey \
      -alias symmetric-key \
      -keystore jwtsessionkeystore.pkcs12 \
      -storepass password \
      -storetype pkcs12 \
      -keyalg HmacSHA512 \
      -keysize 512
    ```

    Note

    Because keytool converts all characters in its key aliases to lowercase, use only lowercase in alias definitions of a keystore.

2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/jwt-session-encrypt.json
```

Windows

```
%appdata%\OpenIG\config\routes\jwt-session-encrypt.json
```

```
{
  "name": "jwt-session-encrypt",
  "heap":  [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file": "/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePasswordSecretId": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{
        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
      }]
    }
  },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    }
  ],
  "session": {
    "type": "JwtSession",
    "config": {
      "authenticatedEncryptionSecretId": "jwtsession.symmetric.secret.id",
      "encryptionMethod": "A256CBC-HS512",
      "secretsProvider": ["KeyStoreSecretStore-1"],
      "cookie": {
        "name": "IG",
        "domain": ".example.com"
      }
    }
  },
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/plain; charset=UTF-8" ]
      },
      "entity": "Hello world!"
    }
  },
  "condition": "${request.uri.path == '/jwt-session-encrypt'}"
}
```

Notice the following features of the route:

- The route matches requests to `/jwt-session-encrypt`.
- The KeyStoreSecretStore uses the SystemAndEnvSecretStore in the heap to manage the store password.
- The JwtSession uses the KeyStoreSecretStore in the heap to manage the session encryption secret.

3. In the terminal where you will run the IG instance, create an environment variable for the value of the keystore password:

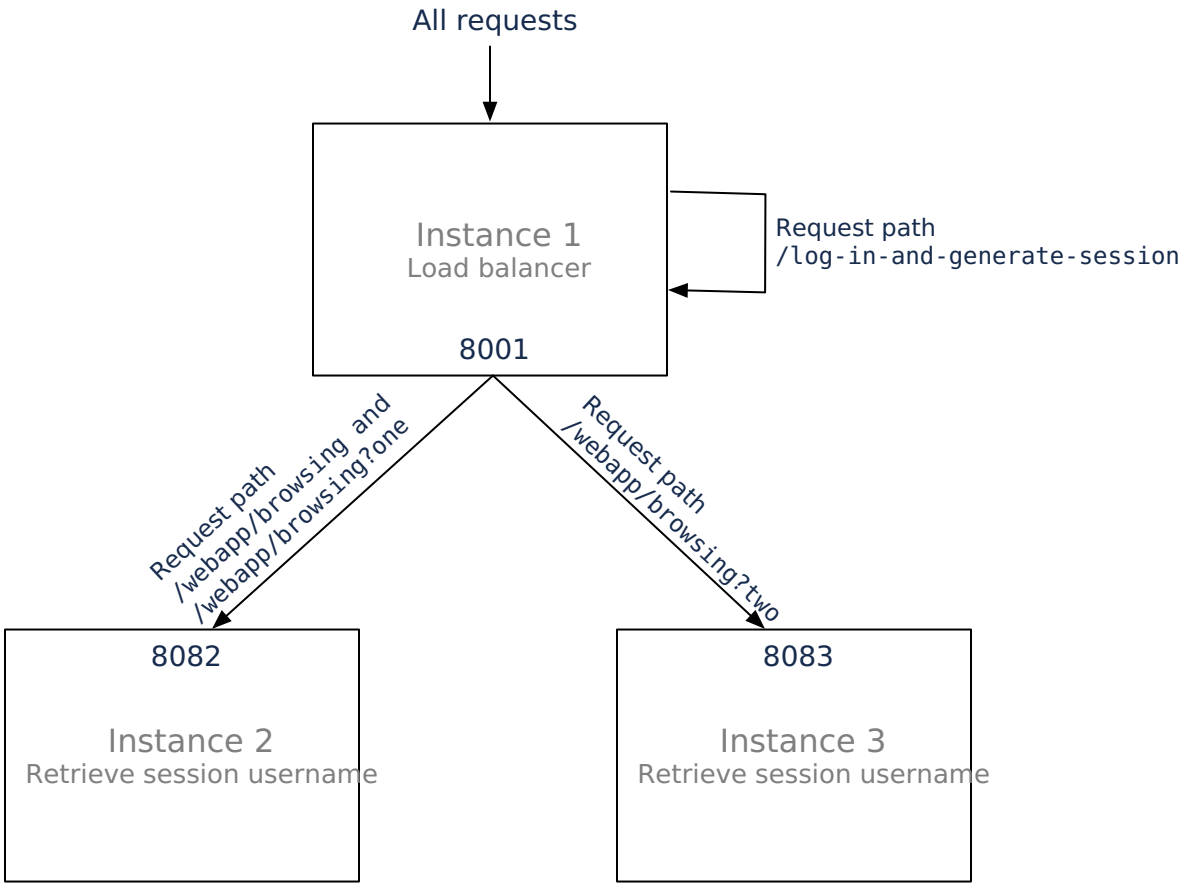
```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by the SystemAndEnvSecretStore, and must be base64-encoded.

Share JWT sessions between multiple instances of IG

When a session is shared between multiple instances of IG, the instances are able to share the session information for load balancing and failover.

This section gives an example of how to set up a deployment with three instances of IG that share a JwtSession.



1. Set up a keystore to contain the encryption key, where the keystore and the key have the password `password`:

    1. Locate a directory for secrets, and go to it:

    ```
    $ cd /path/to/secrets
    ```

    2. Generate the key:

```
$ keytool \
  -genseckey \
  -alias symmetric-key \
  -keystore jwtsessionkeystore.pkcs12 \
  -storepass password \
  -storetype pkcs12 \
  -keyalg HmacSHA512 \
  -keysize 512
```

> **Note**
>
> Because keytool converts all characters in its key aliases to lowercase, use only lowercase in alias definitions of a keystore.

2. Prepare the IG installation:
    1. Create an installation directory for IG in `/path/to`.
    2. Download and unzip IG-2024.3.0.zip in `/path/to`, as described in the Install. The directory /path/to/identity-gateway-2024.3.0 is created.
3. Set up the first instance of IG, which acts as the load balancer:
    1. Create a configuration directory for the instance and go to it:

```
$ mkdir -p /path/to/config-instance1/config/routes
```

    2. Add the following route:

Linux

```
/path/to/config-instance1/config/routes/instance1-loadbalancer.json
```

Windows

```
%appdata%\path\to\config-instance1\config\routes\instance1-loadbalancer.json
```

```
{
  "name": "instance1-loadbalancer",
  "heap": [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file": "/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePasswordSecretId": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{
        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
      }]
    }
  },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    }
  ],
  "session": {
    "type": "JwtSession",
    "config": {
      "authenticatedEncryptionSecretId": "jwtsession.symmetric.secret.id",
      "encryptionMethod": "A256CBC-HS512",
      "secretsProvider": ["KeyStoreSecretStore-1"],
      "cookie": {
        "name": "IG",
        "domain": ".example.com"
      }
    }
  },
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [{
        "condition": "${find(request.uri.path, '/webapp/browsing') and (contains(request.uri.query,
'one') or empty(request.uri.query))}",
        "baseURI": "http://ig.example.com:8002",
        "handler": "ReverseProxyHandler"
      }, {
        "condition": "${find(request.uri.path, '/webapp/browsing') and contains(request.uri.query,
'two')}",
        "baseURI": "http://ig.example.com:8003",
        "handler": "ReverseProxyHandler"
      }, {
        "condition": "${find(request.uri.path, '/log-in-and-generate-session')}",
        "handler": {
          "type": "Chain",
          "config": {
            "filters": [{
              "type": "AssignmentFilter",
              "config": {
                "onRequest": [{
                  "target": "${session.authUsername}",
                  "value": "Sam Carter"
                }]
              }
            }],
```

```
            "handler": {
              "type": "StaticResponseHandler",
              "config": {
                "status": 200,
                "headers": {
                  "Content-Type": [ "text/html; charset=UTF-8" ]
                },
                "entity": "<html><body>Sam Carter logged IN. (JWT session generated)</body></html>"
              }
            }
          }
        }
      }]
    }
  },
  "capture": "all"
}
```

Notice the following features of the route:

- The route has no condition, so it matches all requests.
- When the request matches `/log-in-and-generate-session`, the DispatchHandler creates a JWT session, whose `authUsername` attribute contains the name `Sam Carter`.
- When the request matches `/webapp/browsing`, the DispatchHandler dispatches the request to instance 2 or instance 3, depending on the rest of the request path.

3. Add the following configuration:

Linux

```
/path/to/config-instance1/config/admin.json
```

Windows

```
%appdata%\path\to\config-instance1\config\admin.json
```

```
{
  "connectors": [{
  "port": 8001
  }]
}
```

4. In the terminal where you will run the IG instance, create an environment variable for the value of the keystore password:

```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='
```

   The password is retrieved by the SystemAndEnvSecretStore, and must be base64-encoded.

5. Start IG:

Linux

```
$ /path/to/identity-gateway-2024.3.0/bin/start.sh /path/to/config-instance1/

...
... started in 1234ms on ports : [8001]
```

Windows

```
C:\path\to\identity-gateway-2024.3.0\bin\start.bat %appdata%/path/to/config-instance1
```

4. Set up and start the second instance of IG:

   1. Create a configuration directory for the instance:

```
$ mkdir -p /path/to/config-instance2/config/routes
```

   2. Add the following route:

Linux

```
/path/to/config-instance2/config/routes/instance2-retrieve-session-username.json
```

Windows

```
%appdata%\path\to\config-instance2\config\routes\instance2-retrieve-session-
username.json
```

```
{
  "name": "instance2-retrieve-session-username",
  "heap":  [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file": "/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePasswordSecretId": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{
        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
      }]
    }
  },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    }
  ],
  "session": {
    "type": "JwtSession",
    "config": {
      "authenticatedEncryptionSecretId": "jwtsession.symmetric.secret.id",
      "encryptionMethod": "A256CBC-HS512",
      "secretsProvider": ["KeyStoreSecretStore-1"],
      "cookie": {
        "name": "IG",
        "domain": ".example.com"
      }
    }
  },
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/html; charset=UTF-8" ]
      },
      "entity": [
        "<html>",
        "  <body>",
        "    ${session.authUsername!= null?'Hello,
'.concat(session.authUsername).concat(' !'):'Session.authUsername is not defined'}! (instance2)",
        "  </body>",
        "</html>"
      ]
    }
  },
  "condition": "${find(request.uri.path, '/webapp/browsing')}",
  "capture": "all"
}
```

Notice the following features of the route compared to the route for instance 1:

- The route matches the condition `/webapp/browsing`. When a request matches `/webapp/browsing`, the DispatchHandler dispatches it to instance 2.

- The StaticResponseHandler displays information from the session context.

3. Add the following configuration:

Linux

```
/path/to/config-instance2/config/admin.json
```

Windows

```
%appdata%\path\to\config-instance2\config\admin.json
```

```
{
  "connectors": [{
  "port": 8002
  }]
}
```

4. In the terminal where you will run the IG instance, create an environment variable for the value of the keystore password:

```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by the SystemAndEnvSecretStore, and must be base64-encoded.

5. Start IG:

Linux

```
$ /path/to/identity-gateway-2024.3.0/bin/start.sh /path/to/config-instance2/

...
... started in 1234ms on ports : [8002]
```

Windows

```
C:\path\to\identity-gateway-2024.3.0\bin\start.bat %appdata%/path/to/config-instance2
```

5. Set up and start the third instance of IG:

    1. Create a configuration directory:

    ```
    $ mkdir -p /path/to/config-instance3/config/routes
    ```

    2. Add the following route:

    Linux

    ```
    /path/to/config-instance3/config/routes/instance3-retrieve-session-username.json
    ```

    Windows

    ```
    %appdata%\path\to\config-instance3\config\routes\instance3-retrieve-session-
    username.json
    ```

```
{
  "name": "instance3-retrieve-session-username",
  "heap":  [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file": "/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePasswordSecretId": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{
        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
      }]
    }
  },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    }
  ],
  "session": {
    "type": "JwtSession",
    "config": {
      "authenticatedEncryptionSecretId": "jwtsession.symmetric.secret.id",
      "encryptionMethod": "A256CBC-HS512",
      "secretsProvider": ["KeyStoreSecretStore-1"],
      "cookie": {
        "name": "IG",
        "domain": ".example.com"
      }
    }
  },
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/html; charset=UTF-8" ]
        },
        "entity": [
          "<html>",
          "  <body>",
          "    ${session.authUsername!= null?'Hello,
'.concat(session.authUsername).concat(' !'):'Session.authUsername is not defined'}! (instance3)",
          "  </body>",
          "</html>"
        ]
      }
    },
  "condition": "${find(request.uri.path, '/webapp/browsing')}",
  "capture": "all"
}
```

Notice that the route is the same as that for instance 2, apart from the text in the entity of the StaticResponseHandler.

3. Add the following configuration:

Linux

```
/path/to/config-instance3/config/admin.json
```

Windows

```
%appdata%\path\to\config-instance3\config\admin.json
```

```
{
  "connectors": [{
  "port": 8003
  }]
}
```

4. In the terminal where you will run the IG instance, create an environment variable for the value of the keystore password:

```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by the SystemAndEnvSecretStore, and must be base64-encoded.

5. Start IG:

Linux

```
$ /path/to/identity-gateway-2024.3.0/bin/start.sh /path/to/config-instance3/

...
... started in 1234ms on ports : [8003]
```

Windows

```
C:\path\to\identity-gateway-2024.3.0\bin\start.bat %appdata%/path/to/config-instance3
```

6. Test the setup:

    1. Access instance 1, to generate a session:

    ```
    $ curl -v http://ig.example.com:8001/log-in-and-generate-session

    GET /log-in-and-generate-session HTTP/1.1
    ...

    HTTP/1.1 200 OK
    Content-Length: 84
    Set-Cookie: IG=eyJ...HyI; Path=/; Domain=.example.com; HttpOnly
    ...
    Sam Carter logged IN. (JWT session generated)
    ```

    2. Using the JWT cookie returned in the previous step, access instance 2:

    ```
    $ curl -v http://ig.example.com:8001/webapp/browsing\?one --header "cookie:IG=eyJ...HyI"

    GET /webapp/browsing?one HTTP/1.1
    ...
    cookie: IG=eyJ...HyI
    ...
    HTTP/1.1 200 OK
    ...
    Hello, Sam Carter !! (instance2)
    ```

    Note that instance 2 can access the session info.

    3. Using the JWT cookie again, access instance 3:

    ```
    $ curl -v http://ig.example.com:8001/webapp/browsing\?two --header "cookie:IG=eyJ...HyI"

    GET /webapp/browsing?two HTTP/1.1
    ...
    cookie: IG=eyJ...HyI
    ...
    HTTP/1.1 200 OK
    ...
    Hello, Sam Carter !! (instance3)
    ```

    Note that instance 3 can access the session info.

Prepare for load balancing and failover

For high scale or highly available deployments, consider using a pool of IG servers with nearly identical configurations. Load balance requests across the pool to handle more load. Route around any servers that become unavailable.

Manage state information

Before spreading requests across multiple servers, decide how to manage state information. IG manages state information in the following ways:

Stores state information in a context

By using filters that can store information in the context. Most filters depend on information in the request, response, or context, some of which is first stored by IG. For a summary of filters that can populate a context, refer to Summary of contexts.

By using a handler such as the ScriptableHandler that can store state information in the context. Most handlers depend on information in the context, some of which is first stored by IG.

Retrieves state information to local memory

By using filters and handlers that depend on the configuration of the local file system, such as the following filters:

- FileAttributesFilter
- ScriptableFilter
- ScriptableHandler
- SqlAttributesFilter

When a server becomes unavailable, state information held in local memory is lost. To prevent data loss when a server becomes unavailable, set up failover. Server failover should be transparent to client applications.

Prepare stateless sessions

For example configurations, refer to Encrypt and share JWT sessions.

JwtSession

Manage stateless sessions though JwtSession. Session content is stored on a JWT cookie on the user agent.

So that any server can read or update a JWT cookie from any other server in the same cookie domain, encrypt JWT sessions and share keys and secret across all IG configurations.

Encrypt JWT sessions. The maximum size of the JWT session cookie is 4 KBytes, as defined by the browser. If the cookie exceeds this size, IG automatically splits it into multiple cookies.

Session stickiness

Session stickiness helps to ensure that a client request goes to the server holding the original session data.

If data attached to a context must be stored on the server-side, configure session stickiness so that the load balancer sends all requests from the same client session to the same server.

For an example configuration, refer to Share JWT sessions between multiple instances of IG.

SAML in deployments with multiple instances of IG

IG uses AM federation libraries to implement SAML. When IG acts as a SAML service provider, some internal state information is maintained in the fedlet instead of the session cookie. In deployments that use multiple instances of IG as a SAML service provider, set up sticky sessions so that requests go to the server that started the SAML interaction.

For information, refer to Session state considerations in AM's *SAML v2.0 guide*.

Secure connections

IG is often deployed to replay credentials or other security information. In a real world deployment, this information must be communicated over a secure connection using HTTPS, meaning HTTP over encrypted Transport Layer Security (TLS). Never send real credentials, bearer tokens, or other security information unprotected over HTTP.

This guide describes how to install self-signed certificates for trying out the software or for deployments where you manage all clients that access IG. For information about how to use well-known CA-signed certificates, refer to the documentation for the Java Virtual Machine (JVM).

After installing certificates for client-server trust, consider which cipher suites to use. IG inherits the list of cipher suites from the underlying Java environment.

IG uses the JSSE to secure connections. You can set security and system properties to configure the JSSE. For a list of properties to customize the JSSE in Oracle Java, refer to the *Customization* section of the JSSE Reference guide.

Configure IG for HTTPS (client-side)

When IG sends requests over HTTP to a proxied application, or requests services from a third-party application, IG is acting as a client of the application, and the application is acting as a server. IG is *client-side*.

When IG sends requests securely over HTTPS, IG must be able to trust the server. By default, IG uses the Java environment truststore to trust server certificates. The Java environment truststore includes public key signing certificates from many well-known Certificate Authorities (CAs).

When servers present certificates signed by trusted CAs, then IG can send requests over HTTPS to those servers, without any configuration to set up the HTTPS client connection. When server certificates are self-signed or signed by a CA whose certificate is not automatically trusted, the following objects can be required to configure the connection:

- KeyStoreSecretStore, to manage a secret store for cryptographic keys and certificates, based on a standard Java keystore.
- SecretsTrustManager, to manage trust material that verifies the credentials presented by a peer.
- (Optional) SecretsKeyManager, to manage keys that authenticate a TLS connection to a peer.
- ClientHandler and ReverseProxyHandler reference to ClientTlsOptions, for connecting to TLS-protected endpoints.

The following procedure describes how to set up IG for HTTPS (client-side), when server certificates are self-signed or signed by untrusted CAs.

Set up IG for HTTPS (client-side) for untrusted servers

1. Locate or set up the following directories:
   - Directory containing the sample application .jar: sampleapp_install_dir
   - Directory to store the sample application certificate and IG keystore: /path/to/secrets
2. Extract the public certificate from the sample application:

```
$ cd /path/to/secrets
```

```
$ jar --verbose --extract \
--file sampleapp_install_dir/IG-sample-application-2024.3.0-jar-with-dependencies.jar tls/sampleapp-cert.pem

inflated: tls/sampleapp-cert.pem
```

   The file `/path/to/secrets/tls/sampleapp-cert.pem` is created.
3. From the same directory, import the certificate into the IG keystore, and answer `yes` to trust the certificate:

```
$ keytool -importcert \
-alias ig-sampleapp \
-file tls/sampleapp-cert.pem \
-keystore reverseproxy-truststore.p12 \
-storetype pkcs12 \
-storepass password

...
Trust this certificate? [no]: yes

Certificate was added to keystore
```

> **Note**
>
> Because keytool converts all characters in its key aliases to lowercase, use only lowercase in alias definitions of a keystore.

4. List the keys in the IG keystore to make sure that a key with the alias `ig-sampleapp` is present:

```
$ keytool -list \
-v \
-keystore /path/to/secrets/reverseproxy-truststore.p12 \
-storetype pkcs12 \
-storepass password

Keystore type: PKCS12
Keystore provider: SUN
Your keystore contains 1 entry
Alias name: ig-sampleapp
...
```

5. In the terminal where you run IG, create an environment variable for the value of the keystore password:

```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by the SystemAndEnvSecretStore, and must be base64-encoded.

6. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

7. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/client-side-https.json
```

Windows

```
%appdata%\OpenIG\config\routes\client-side-https.json
```

```
{
  "name": "client-side-https",
  "condition": "${find(request.uri.path, '/home/client-side-https')}",
  "baseURI": "https://app.example.com:8444",
  "heap": [
    {
      "name": "Base64EncodedSecretStore-1",
      "type": "Base64EncodedSecretStore",
      "config": {
        "secrets": {
          "keystore.secret.id": "cGFzc3dvcmQ="
        }
      }
    },
    {
      "name": "KeyStoreSecretStore-1",
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "/path/to/secrets/reverseproxy-truststore.p12",
        "storeType": "PKCS12",
        "storePasswordSecretId": "keystore.secret.id",
        "secretsProvider": "Base64EncodedSecretStore-1",
        "mappings": [
          {
            "secretId": "trust.manager.secret.id",
            "aliases": [ "ig-sampleapp" ]
          }
        ]
      }
    },
    {
      "name": "SecretsTrustManager-1",
      "type": "SecretsTrustManager",
      "config": {
        "verificationSecretId": "trust.manager.secret.id",
        "secretsProvider":"KeyStoreSecretStore-1"
      }
    },
    {
      "name": "ReverseProxyHandler-1",
      "type": "ReverseProxyHandler",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": "SecretsTrustManager-1"
          }
        },
        "hostnameVerifier": "ALLOW_ALL"
      },
      "capture": "all"
    }
  ],
  "handler": "ReverseProxyHandler-1"
}
```

Notice the following features of the route:

- The route matches requests to `/home/client-side-https`.

- The `baseURI` changes the request URI to point to the HTTPS port for the sample application.
- The Base64EncodedSecretStore provides the keystore password.
- The SecretsTrustManager uses a KeyStoreSecretStore to manage the trust material.
- The KeyStoreSecretStore points to the sample application certificate. The password to access the keystore is provided by the SystemAndEnvSecretStore.
- The ReverseProxyHandler uses the SecretsTrustManager for the connection to TLS-protected endpoints. All hostnames are allowed.

8. Test the setup:

   1. Start the sample application

      ```
      $ java -jar sampleapp_install_dir/IG-sample-application-2024.3.0-jar-with-dependencies.jar
      ```

   2. Go to http://ig.example.com:8080/home/client-side-https.

      The request is proxied transparently to the sample application, on the TLS port `8444`.

   3. Check the route log for a line like this:

      ```
      GET https://app.example.com:8444/home/client-side-https
      ```

Configure IG for HTTPS (server-side)

When IG is *server-side*, applications send requests to IG or request services from IG. IG is acting as a server of the application, and the application is acting as a client.

To run IG as a server over HTTPS, you must configure connections to TLS-protected endpoints, based on ServerTlsOptions.

For more information, refer to About keys and certificates.

Serve the same certificate for TLS connections to all server names

This example uses PEM files and a PKCS#12 keystore for self-signed certificates, but you can adapt it to use official (non self-signed) keys and certificates.

Before you start, install IG, as described in the Install.

1. Locate a directory for the secrets, for example, `/path/to/secrets`.
2. Create self-signed keys in one of the following ways. If you have your own keys, use them and skip this step.

   If you have your own keys, use them and skip this step.

   1. Create the keystore, replacing `/path/to/secrets` with your path:

```
$ keytool \
-genkey \
-alias https-connector-key \
-keyalg RSA \
-keystore /path/to/secrets/IG-keystore \
-storepass password \
-keypass password \
-dname "CN=ig.example.com,O=Example Corp,C=FR"
```

Note

Because keytool converts all characters in its key aliases to lowercase, use only lowercase in alias definitions of a keystore.

2. In the secrets directory, add a file called `keystore.pass`, containing the keystore password `password`:

```
$ cd /path/to/secrets/
$ echo -n 'password' > keystore.pass
```

Make sure the password file contains only the password, with no trailing spaces or carriage returns.

1. Locate a directory for secrets, and go to it:

```
$ cd /path/to/secrets
```

2. Create the following secret key and certificate pair as PEM files:

```
$ openssl req \
-newkey rsa:2048 \
-new \
-nodes \
-x509 \
-days 3650 \
-subj "/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
-keyout ig.example.com-key.pem \
-out ig.example.com-certificate.pem
```

Two PEM files are created, one for the secret key, and another for the associated certificate.

3. Map the key and certificate to the same secret ID in IG:

```
$ cat ig.example.com-key.pem ig.example.com-certificate.pem > key.manager.secret.id.pem
```

3. Set up TLS on IG in one of the following ways:

Add the following file to IG, replacing `/path/to/secrets` with your path:

Linux

```
$HOME/.openig/config/admin.json
```

Windows

```
%appdata%\OpenIG\config\admin.json
```

```
{
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443,
      "tls": "ServerTlsOptions-1"
    }
  ],
  "heap": [
    {
      "name": "ServerTlsOptions-1",
      "type": "ServerTlsOptions",
      "config": {
        "keyManager": {
          "type": "SecretsKeyManager",
          "config": {
            "signingSecretId": "key.manager.secret.id",
            "secretsProvider": "ServerIdentityStore"
          }
        }
      }
    },
    {
      "type": "FileSystemSecretStore",
      "name": "SecretsPasswords",
      "config": {
        "directory": "/path/to/secrets",
        "format": "PLAIN"
      }
    },
    {
      "name": "ServerIdentityStore",
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "/path/to/secrets/IG-keystore",
        "storePasswordSecretId": "keystore.pass",
        "secretsProvider": "SecretsPasswords",
        "mappings": [
          {
            "secretId": "key.manager.secret.id",
            "aliases": ["https-connector-key"]
          }
        ]
      }
    }
  ]
}
```

Notice the following features of the file:

- IG starts on port `8080`, and on `8443` over TLS.
- IG's private keys for TLS are managed by the SecretsKeyManager, whose ServerIdentityStore references a KeyStoreSecretStore.

- The KeyStoreSecretStore maps the keystore alias to the secret ID for retrieving the server keys (private key + certificate).
- The password of the KeyStoreSecretStore is provided by the FileSystemSecretStore.

Add the following file to IG, replacing `/path/to/secrets` with your path:

Linux

```
$HOME/.openig/config/admin.json
```

Windows

```
%appdata%\OpenIG\config\admin.json
```

```
{
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443,
      "tls": "ServerTlsOptions-1"
    }
  ],
  "heap": [
    {
      "name": "ServerTlsOptions-1",
      "type": "ServerTlsOptions",
      "config": {
        "keyManager": {
          "type": "SecretsKeyManager",
          "config": {
            "signingSecretId": "key.manager.secret.id",
            "secretsProvider": "ServerIdentityStore"
          }
        }
      }
    },
    {
      "name": "ServerIdentityStore",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "/path/to/secrets",
        "suffix": ".pem",
        "mappings": [{
          "secretId": "key.manager.secret.id",
          "format": {
            "type": "PemPropertyFormat"
          }
        }]
      }
    }
  ]
}
```

Notice how this file differs to that for the keystore-based approach:

- The ServerIdentityStore is a FileSystemSecretStore.
- The FileSystemSecretStore reads the keys that are stored as file in the PEM standard format.

4. Start IG:

Linux

```
$ /path/to/identity-gateway-2024.3.0/bin/start.sh

...
... started in 1234ms on ports : [8080 8443]
```

Windows

```
C:\path\to\identity-gateway-2024.3.0\bin\start.bat
```

By default, IG configuration files are located under `$HOME/.openig` (on Windows `%appdata%\OpenIG`). For information about how to use a different location, refer to Configuration location.

5. Test the connection in one of the following ways:

   - Ping IG and make sure an `HTTP 200` is returned:

     ```
     $ curl -v --cacert /path/to/secrets/ig.example.com-certificate.pem \
     https://ig.example.com:8443/openig/ping
     ```

   - Display the product version and build information:

     ```
     $ curl --cacert /path/to/secrets/ig.example.com-certificate.pem \
     https://ig.example.com:8443/openig/api/info
     ```

Serve different certificates for TLS connections to different server names

This example uses PEM files for self-signed certificates, but you can adapt it to use official (non self-signed) keys and certificates.

Before you start, install IG, as described in the Install.

1. Locate a directory for secrets, for example, `/path/to/secrets`, and go to it.

   ```
   $ cd /path/to/secrets
   ```

2. Create the following secret key and certificate pair as PEM files:

   1. For `ig.example.com`:

      1. Create a key and certificate:

```
$ openssl req \
-newkey rsa:2048 \
-new \
-nodes \
-x509 \
-days 3650 \
-subj "/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
-keyout ig.example.com-key.pem \
-out ig.example.com-certificate.pem
```

Two PEM files are created, one for the secret key, and another for the associated certificate.

2. Map the key and certificate to the same secret ID in IG:

```
$ cat ig.example.com-key.pem ig.example.com-certificate.pem > key.manager.secret.id.pem
```

2. For servers grouped by a wildcard:

1. Create a key and certificate:

```
$ openssl req \
-newkey rsa:2048 \
-new \
-nodes \
-x509 \
-days 3650 \
-subj "/CN=*.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
-keyout wildcard.example.com-key.pem \
-out wildcard.example.com-certificate.pem
```

2. Map the key and certificate to the same secret ID in IG:

```
$ cat wildcard.example.com-key.pem wildcard.example.com-certificate.pem > wildcard.secret.id.pem
```

3. For other, unmapped servers

1. Create a key and certificate:

```
$ openssl req \
-newkey rsa:2048 \
-new \
-nodes \
-x509 \
-days 3650 \
-subj "/CN=un.mapped.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
-keyout default.example.com-key.pem \
-out default.example.com-certificate.pem
```

2. Map the key and certificate to the same secret ID in IG:

```
$ cat default.example.com-key.pem default.example.com-certificate.pem > default.secret.id.pem
```

3. Add the following file to IG, replacing `/path/to/secrets` with your path, and then restart IG:

Linux

```
$HOME/.openig/config/admin.json
```

Windows

```
%appdata%\OpenIG\config\admin.json
```

```
{
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443,
      "tls": "ServerTlsOptions-1"
    }
  ],
  "heap": [
    {
      "name": "ServerTlsOptions-1",
      "type": "ServerTlsOptions",
      "config": {
        "sni": {
          "serverNames": {
            "ig.example.com": "key.manager.secret.id",
            "*.example.com": "wildcard.secret.id"
          },
          "defaultSecretId" : "default.secret.id",
          "secretsProvider": "ServerIdentityStore"
        }
      }
    },
    {
      "name": "ServerIdentityStore",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "path/to/secrets",
        "suffix": ".pem",
        "mappings": [
          {
            "secretId": "key.manager.secret.id",
            "format": {
              "type": "PemPropertyFormat"
            }
          },
          {
            "secretId": "wildcard.secret.id",
            "format": {
              "type": "PemPropertyFormat"
            }
          },
          {
            "secretId": "default.secret.id",
            "format": {
              "type": "PemPropertyFormat"
            }
          }
        ]
      }
    }
  ]
}
```

Notice the following features of the file:

- The ServerTlsOptions object maps two servers to secret IDs, and includes a default secret ID
- The secret IDs correspond to the secret IDs in the FileSystemSecretStore, and the PEM files generated in an earlier step.

4. Run the following commands to request TLS connections to different servers, using different certificates:

   1. Connect to `ig.example.com`, and note that the certificate subject corresponds to the certificate created for `ig.example.com`:

      ```
      $ openssl s_client -connect localhost:8443 -servername ig.example.com

      ...
      Server certificate
      -----BEGIN CERTIFICATE-----
      MII...dZC
      -----END CERTIFICATE-----
      subject=/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr
      issuer=/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr
      ```

   2. Connect to `other.example.com`, and note that the certificate subject corresponds to the certificate created with the wildcard, `*.example.com`:

      ```
      $ openssl s_client -connect localhost:8443 -servername other.example.com

      ...
      Server certificate
      -----BEGIN CERTIFICATE-----
      MII...fY=
      -----END CERTIFICATE-----
      subject=/CN=*.example.com/OU=example/O=com/L=fr/ST=fr/C=fr
      issuer=/CN=*.example.com/OU=example/O=com/L=fr/ST=fr/C=fr
      ```

   3. Connect to `unmapped.site.com`, and note that the certificate subject corresponds to the certificate created for the default secret ID:

      ```
      $ openssl s_client -connect localhost:8443 -servername unmapped.site.com

      ...
      Server certificate
      -----BEGIN CERTIFICATE-----
      MII..rON
      -----END CERTIFICATE-----
      subject=/CN=un.mapped.com/OU=example/O=com/L=fr/ST=fr/C=fr
      issuer=/CN=un.mapped.com/OU=example/O=com/L=fr/ST=fr/C=fr
      ```

Configure

This guide shows you how to configure IG software features.

Configuration files and routes

IG processes requests and responses by using the following JSON files: admin.json, config.json, Route, and Router.

Configuration location

The following table summarizes the default location of the IG configuration and logs.

Purpose	Default location on Linux	Default location on Windows
Log messages from IG and third-party dependencies	`$HOME/.openig/logs`	`%appdata%\OpenIG\logs`
Administration (admin.json) Gateway (config.json)	`$HOME/.openig/config`	`%appdata%\OpenIG\config`
Routes (Route)	`$HOME/.openig/config/routes`	`%appdata%\OpenIG\config\routes`
SAML 2.0	`$HOME/.openig/SAML`	`%appdata%\OpenIG\SAML`
Groovy scripts for scripted filters and handlers, and other objects	`$HOME/.openig/scripts/groovy`	`%appdata%\OpenIG\scripts\groovy`
Temporary directory To change the directory, configure `temporaryDirectory` in admin.json	`$HOME/.openig/tmp`	`%appdata%\OpenIG\tmp`
JSON schema for custom audit To change the directory, configure `topicsSchemasDirectory` in AuditService.	`$HOME/.openig/audit-schemas`	`%appdata%\OpenIG\audit-schemas`

Configuration security

Allow the following access to `$HOME/.openig/logs`, `$HOME/.openig/tmp`, and all configuration directories:

- Highest privilege the IG system account.
- Least priviledge for specific accounts, on a case-by-case basis
- No priviledge for all other accounts, by default

Change the base location

By default, the base location for IG configuration files is in the following directory:

Linux

```
$HOME/.openig
```

Windows

```
%appdata%\OpenIG
```

Change the location in the following ways:

- Edit the `prefix` property of admin.json.
- Use an argument with the startup command. The following example reads the configuration from the `config` directory under `/path/to/config-dir`:

Linux

```
$ /path/to/identity-gateway-2024.3.0/bin/start.sh /path/to/config-dir
```

Windows

```
C:\path\to\identity-gateway-2024.3.0\bin\start.bat /path/to/config-dir
```

Route names, IDs, and filenames

The filenames of routes have the extension .json, in lowercase.

The router scans the `$HOME/.openig/config/routes` folder for files with the .json extension. It uses the route `name` property to order the routes in the configuration. If the route does not have a `name` property, the router uses the route ID.

The route ID is managed as follows:

- When you add a route manually to the routes folder, the route ID is the value of the `_id` field. If there is no `_id` field, the route ID is the filename of the added route.
- When you add a route through the Common REST endpoint, the route ID is the value of the mandatory `_id` field.
- When you add a route through Studio, you can edit the default route ID.

> **Caution**
>
> The filename of a route can't be `default.json`, and the route's `name` property and route ID can't be `default`.

Inline and heap objects

Inline objects

An inline object is declared in a route or configuration, outside of the heap.

The following example shows an inline declaration for a handler to route requests:

```
{
  "handler": {
    "name": "My Router",
    "type": "Router"
  }
}
```

The `name` property for inline objects is optional but useful for logging.

Other objects in the configuration can never refer to named or unnamed inline objects.

Heap objects

A heap object is declared inside the heap.

The following example shows a named router in the heap, and a handler that refers to the router by its name:

```
{
  "handler": "My Router",
  "heap": [
    {
      "name": "My Router",
      "type": "Router"
    }
  ]
}
```

The `name` property for heap objects is required. Other objects in the configuration or its child configirations can refer to the heap obect by its `name` property.

Comment the configuration

JSON format doesn't specify a notation for comments. If IG does not recognize a JSON field name, it ignores the field. As a result, it's possible to use comments in configuration files.

The following conventions are available for commenting:

- A `comment` field to add text comments. The following example includes a text comment.

```
{
  "name": "capture",
  "type": "CaptureDecorator",
  "comment": "Write request and response information to the logs",
  "config": {
    "captureEntity": true
  }
}
```

- An underscore (`_`) to comment a field temporarily. The following example comments out `"captureEntity": true`, and as a result it uses the default setting (`"captureEntity": false`).

```
{
  "name": "capture",
  "type": "CaptureDecorator",
  "config": {
    "_captureEntity": true
  }
}
```

Restart after configuration change

You can change routes or change a property that's read at runtime or that relies on a runtime expression without needing to restart IG to take the change into account.

Stop and restart IG only when you make the following changes:

- Change the configuration of any route, when the `scanInterval` of Router is `disabled` (see Router).
- Add or change an external object used by the route, such as an environment variable, system property, external URL, or keystore.
- Add or update `config.json` or `admin.json` .

Prevent reload of routes

To prevent routes from being reloaded after startup, stop IG, edit the router `scanInterval` , and restart IG. When the interval is set to `disabled` , routes are loaded only at startup:

```
{
  "name": "Router",
  "type": "Router",
  "config": {
    "scanInterval": "disabled"
  }
}
```

The following example changes the location where the router looks for the routes:

```
{
  "name": "Router",
  "type": "Router",
  "config": {
    "directory": "/path/to/safe/routes",
    "scanInterval": "disabled"
  }
}
```

Reserved routes

For information about reserved routes, refer to Reserved routes.

Routes and Common REST

> **Note**
>
> When IG is in production mode, you can't manage, list, or read routes through Common REST. For information about switching to development mode, refer to Operating modes.

Through Common REST, you can read, add, delete, and edit routes on IG without manually accessing the file system. You can also list the routes in the order that they're loaded in the configuration, and set fields to filter the information about the routes.

The following examples show some ways to manage routes through Common REST. For more information, refer to About ForgeRock Common REST.

Manage routes through Common REST

Before you start, prepare IG as described in the Quick install.

1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/00-crest.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-crest.json
```

```
{
  "name": "crest",
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/plain; charset=UTF-8" ]
      },
      "entity": "Hello world!"
    }
  },
  "condition": "${find(request.uri.path, '^/crest')}"
}
```

To check that the route is working, access the route on: http://ig.example.com:8080/crest.

2. To read a route through Common REST:

    1. Enter the following command in a terminal window:

    ```
    $ curl -v http://ig.example.com:8080/openig/api/system/objects/_router/routes/00-crest\?
    _prettyPrint\=true
    ```

    The route is displayed. Note that the route `_id` is displayed in the JSON of the route.

3. To add a route through Common REST:

    1. Move `$HOME/.openig/config/routes/00-crest.json` to `/tmp/00-crest .json`.

    2. Check in `$HOME/.openig/logs/route-system.log` that the route has been removed from the configuration, where `$HOME/.openig` is the instance directory. To double check, go to http://ig.example.com:8080/crest. You should get an HTTP 404 error.

    3. Enter the following command in a terminal window:

    ```
    $ curl -X PUT http://ig.example.com:8080/openig/api/system/objects/_router/routes/00-crest \
           -d "@/tmp/00-crest.json" \
           --header "Content-Type: application/json"
    ```

    This command posts the file in `/tmp/00-crest.json` to the `routes` directory.

    4. Check in `$HOME/.openig/logs/route-system.log` that the route has been added to configuration, where `$HOME/.openig` is the instance directory. To double-check, go to http://ig.example.com:8080/crest. You should see the "Hello world!" message.

4. To edit a route through Common REST:

    1. Edit `/tmp/00-crest.json` to change the message displayed by the response handler in the route.

    2. Enter the following command in a terminal window:

```
$ curl -X PUT http://ig.example.com:8080/openig/api/system/objects/_router/routes/00-crest \
       -d "@/tmp/00-crest.json" \
       --header "Content-Type: application/json" \
       --header "If-Match: *"
```

This command deploys the route with the new configuration. Because the changes are persisted into the configuration, the existing `$HOME/.openig/config/routes/00-crest.json` is replaced with the edited version in `/tmp/00-crest.json`.

3. Check in `$HOME/.openig/logs/route-system.log` that the route has been updated, where `$HOME/.openig` is the instance directory. To double-check, go to http://ig.example.com:8080/crest to confirm that the displayed message has changed.

5. To delete a route through Common REST:

   1. Enter the following command in a terminal window:

   ```
   $ curl -X DELETE http://ig.example.com:8080/openig/api/system/objects/_router/routes/00-crest
   ```

   2. Check in `$HOME/.openig/logs/route-system.log` that the route has been removed from the configuration, where `$HOME/.openig` is the instance directory. To double-check, go to http://ig.example.com:8080/crest. You should get an HTTP 404 error.

6. To list the routes deployed on the router, in the order that they are tried by the router:

   1. Enter the following command in a terminal window:

   ```
   $ curl "http://ig.example.com:8080/openig/api/system/objects/_router/routes?_queryFilter=true"
   ```

   The list of loaded routes is displayed.

Decorators

Decorators are heap objects to extend what other objects can do. IG defines `baseURI`, `capture`, and `timer` decorators that you can use without explicitly configuring them. For information about available decorators, refer to Decorators.

Use decorations that are compatible with the object type. For example, `timer` records the time to process filters and handlers, but does not record information for other object types. Similarly, `baseURI` overrides the scheme, host, and ports, but has no other effect.

In a route, you can decorate individual objects, the route handler, and the heap. IG applies decorations in this order:

1. Decorations declared on individual objects. Local decorations that are part of an object's declaration are inherited wherever the object is used.

2. globalDecorations declared in parent routes, then in child routes, and then in the current route.

3. Decorations declared on the route handler.

Decorate individual objects in a route

To decorate individual objects, add the decorator's name value as a top-level field of the object, next to `type` and `config`.

In this example, the decorator captures all requests going into the SingleSignOnFilter, and all responses coming out of the SingleSignOnFilter:

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "capture": "all",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Decorate the route handler

To decorate the handler for a route, add the decorator as a top-level field of the route.

In this example, the decorator captures all requests and responses that traverse the route:

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent" : {
          "username" : "ig_agent",
          "passwordSecretId" : "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "capture": "all",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Decorate the route heap

To decorate all compatible objects in a route, configure globalDecorators as a top-level field of the route. The globalDecorators field takes a map of the decorations to apply.

To decorate all compatible objects declared in `config.json` or `admin.json`, configure globalDecorators as a top-level field in `config.json` or `admin.json`.

In the following example, the route has capture and timer decorations. The capture decoration applies to AmService, Chain, SingleSignOnFilter, and ReverseProxyHandler. The timer decoration doesn't apply to AmService because it is not a filter or handler, but does apply to Chain, SingleSignOnFilter, and ReverseProxyHandler:

```
{
  "globalDecorators":
  {
    "capture": "all",
    "timer": true
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Decorate named objects differently in different parts of the configuration

When a filter or handler is configured in `config.json` or in the heap, it can be used many times in the configuration. To decorate each use of the filter or handler individually, use a Delegate.

In the following example, an AmService heap object configures an `amHandler` to delegate tasks to `ForgeRockClientHandler`, and capture all requests and responses passing through the handler.

```
{
  "type": "AmService",
  "config": {
    "agent" : {
      "username" : "ig_agent",
      "passwordSecretId" : "agent.secret.id"
      },
    "secretsProvider": "SystemAndEnvSecretStore-1",
    "amHandler": {
      "type": "Delegate",
      "capture": "all",
      "config": {
        "delegate": "ForgeRockClientHandler"
      }
    },
    "url": "http://am.example.com:8088/openam"
  }
}
```

You can use the same `ForgeRockClientHandler` in another part of the configuration, in a different route for example, without adding a capture decorator. Requests and responses that pass through that use of the handler are not captured.

Decorate IG's interactions with AM

To log interactions between IG and AM, delegate message handling to a ForgeRockClientHandler, and capture the requests and responses passing through the handler. When the ForgeRockClientHandler communicates with an application, it sends ForgeRock Common Audit transaction IDs.

In the following example, the `accessTokenResolver` delegates message handling to a decorated ForgeRockClientHandler:

```
"accessTokenResolver": {
  "name": "token-resolver-1",
  "type": "TokenIntrospectionAccessTokenResolver",
  "config": {
    "amService": "AmService-1",
    "providerHandler": {
      "capture": "all",
      "type": "Delegate",
      "config": {
        "delegate": "ForgeRockClientHandler"
      }
    }
  }
}
```

To try the example, replace the `accessTokenResolver` in the IG route of Validate access tokens through the introspection endpoint. Test the setup as described for the example, and note that the route's log file contains an HTTP call to the introspection endpoint.

Decorate an object multiple times

Decorations can apply more than once. For example, if you set a decoration on a route and another decoration on an object defined within the route, IG applies the decoration twice. In the following route, the request is captured twice:

```
{
 "handler": {
   "type": "ReverseProxyHandler",
   "capture": "request"
 },
 "capture": "all"
}
```

When an object has multiple decorations, the decorations are applied in the order they appear in the JSON.

In the following route, the handler is decorated with a `baseURI` first, and a `capture` second:

```
{
  "name": "myroute",
  "baseURI": "http://app.example.com:8081",
  "capture": "all",
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/plain; charset=UTF-8" ]
      },
      "entity": "Hello world, from myroute!"
    }
  },
  "condition": "${find(request.uri.path, '^/myroute1')}"
}
```

The decoration can be represented as `capture[ baseUri[ handler ] ]`. When a request is processed, it is captured, and then rebased, and then processed by the handler: The log for this route shows that the capture occurs before the rebase:

```
2018-09-10T13:23:18,990Z | INFO  | http-nio-8080-exec-1 | o.f.o.d.c.C.c.top-level-handler | @myroute |

--- (request) id:f792d2ad-4409-4907-bc46-28e1c3c19ac3-7 --->

GET http://ig.example.com:8080/myroute HTTP/1.1
...
```

Conversely, in the following route, the handler is decorated with a `capture` first, and a `baseURI` second:

```
{
  "name": "myroute",
  "capture": "all",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/plain; charset=UTF-8" ]
      }
      "entity": "Hello, world from myroute1!"
    }
  },
  "condition": "${find(request.uri.path, '^/myroute')}"
}
```

The decoration can be represented as `baseUri[ capture[ handler ] ]`. When a request is processed, it is rebased, and then captured, and then processed by the handler. The log for this route shows that the rebase occurs before the capture:

```
2018-09-10T13:07:07,524Z | INFO  | http-nio-8080-exec-1 | o.f.o.d.c.C.c.top-level-handler | @myroute |

--- (request) id:3c26ab12-3cc0-403e-bec6-43bf5621f657-7 --->

GET http://app.example.com:8081/myroute HTTP/1.1
...
```

Guidelines for naming decorators

To prevent unwanted behavior, consider the following points when you name decorators:

- Avoid decorators named `comment` or `comments`, and avoid reserved field names. Instead of using alphanumeric field names, consider using dots in your decorator names, such as `my.decorator`.
- For heap objects, avoid the reserved names `config`, `name`, and `type`.
- For routes, avoid the reserved names `auditService`, `baseURI`, `condition`, `globalDecorators`, `heap`, `handler`, `name`, `secrets`, and `session`.
- In `config.json`, avoid the reserved name `temporaryStorage`.

Operating modes

Production mode (immutable mode)

To prevent unwanted changes to the configuration, IG is by default in production mode after installation. Production mode has the following characteristics:

- The `/routes` endpoint isn't exposed or accessible.

- Studio is effectively disabled. You can't manage, list, or even read routes through Common REST.
- By default, other endpoints, such as `/share` and `api/info` are exposed to the loopback address only.

  To change the default protection for specific endpoints, configure an ApiProtectionFilter in `admin.json` and add it to the IG configuration.

Development mode (mutable mode)

In development mode, by default all endpoints are open and accessible.

You can create, edit, and deploy routes through IG Studio, manage routes through Common REST without authentication or authorization, and access API descriptors.

Use development mode to evaluate or demo IG, or to develop configurations on a single instance. This mode isn't suitable for production.

For information about Restrict access to Studio in development mode, refer to Restrict access to Studio.

Switch from production mode to development mode

Switch from production mode to development mode in one of the following ways, applied in order of precedence:

1. Add the following configuration to `admin.json`, and restart IG:

```
{
  "mode": "DEVELOPMENT",
  "connectors": [
    { "port" : 8080 }
  ]
}
```

2. Define an environment variable for the configuration token `ig.run.mode`, and then start IG in the same terminal.

   If `mode` is not defined in `admin.json`, the following example starts an instance of IG in development mode:

Linux

```
$ IG_RUN_MODE=development /path/to/identity-gateway-2024.3.0/bin/bin/start.sh
```

Windows

```
C:\IG_RUN_MODE=development
C:\path\to\identity-gateway-2024.3.0\bin\start.bat %appdata%\OpenIG
```

3. Define a system property for the configuration token `ig.run.mode` when you start IG.

   If `mode` is not defined in `admin.json`, or an `IG_RUN_MODE` environment variable is not set, the following file starts an instance of IG with the system property `ig.run.mode` to force development mode:

   Linux

   ```
   $HOME/.openig/env.sh
   ```

   Windows

   ```
   %appdata%\OpenIG\env.sh
   ```

   ```
   export JAVA_OPTS='-Dig.run.mode=development'
   ```

Switch from development mode to production mode

Switch from development mode to production mode to prevent unwanted changes to the configuration.

1. In `$HOME/.openig/config/admin.json` (on Windows, `%appdata%\OpenIG\config`), change the value of `mode` from `DEVELOPMENT` to `PRODUCTION`:

   ```
   {
     "mode": "PRODUCTION"
   }
   ```

   The file changes the operating mode from development mode to production mode. For more information about the `admin.json` file, refer to AdminHttpApplication (admin.json).

   The value set in `admin.json` overrides any value set by the `ig.run.mode` configuration token when it is used in an environment variable or system property. For information about `ig.run.mode`, refer to Configuration Tokens.

2. (Optional) Prevent routes from being reloaded after startup:

   - To prevent all routes in the configuration from being reloaded, add a `config.json` as described in the Quick install, and configure the `scanInterval` property of the main Router.
   - To prevent individual routes from being reloaded, configure the `scanInterval` of the routers in those routes.

```
{
  "type": "Router",
  "config": {
    "scanInterval": "disabled"
  }
}
```

For more information, refer to Router.

3. Restart IG.

   When IG starts up, the route endpoints are not displayed in the logs, and are not available. You can't access Studio on http://ig.example.com:8080/openig/studio.

Configuration templates

This chapter contains template routes for common configurations. To use a template, set up IG as described in the Quick install, and modify the template for your deployment. Before you use a route in production, review the points in Security.

Proxy and capture

If you installed and configured IG with a router and default route as described in the Quick install, then you already proxy and capture the application requests coming in and the server responses going out.

This template route uses a `DispatchHandler` to change the scheme to HTTPS on login:

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert for HTTPS.",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            },
            "hostnameVerifier": "ALLOW_ALL"
          }
        }
      }
    }
  ],
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${request.uri.path == '/login'}",
          "handler": "ReverseProxyHandler",
          "baseURI": "https://app.example.com:8444"
        },
        {
          "condition": "${request.uri.scheme == 'http'}",
          "handler": "ReverseProxyHandler",
          "baseURI": "http://app.example.com:8081"
        },
        {
          "handler": "ReverseProxyHandler",
          "baseURI": "https://app.example.com:8444"
        }
      ]
    }
  },
  "condition": "${find(request.uri.query, 'demo=capture')}"
}
```

To try this example with the sample application:

1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/20-capture.json
```

Windows

```
%appdata%\OpenIG\config\routes\20-capture.json
```

2. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

3. Go to http://ig.example.com:8080/login?demo=capture.

   The login page of the sample application is displayed.

To use this as a default route with a real application:

1. Replace the test ReverseProxyHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

   In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.

2. Change the baseURI settings to match the target application.

3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

Simple login form

This template route intercepts the login page request, replaces it with a login form, and logs the user into the target application with hard-coded username and password:

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert for HTTPS.",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            },
            "hostnameVerifier": "ALLOW_ALL"
          }
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${request.uri.path == '/login'}",
            "request": {
              "method": "POST",
              "uri": "https://app.example.com:8444/login",
              "form": {
                "username": [
                  "MY_USERNAME"
                ],
                "password": [
                  "MY_PASSWORD"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.query, 'demo=simple')}"
}
```

To try this example with the sample application:

1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/21-simple.json
```

Windows

```
%appdata%\OpenIG\config\routes\21-simple.json
```

2. Replace `MY_USERNAME` with `demo`, and `MY_PASSWORD` with `Ch4ng31t`.

3. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

4. Go to http://ig.example.com:8080/login?demo=simple.

   The sample application profile page for the demo user displays information about the request:

```
Username        demo

REQUEST INFORMATION
Method  POST
URI     /login
Cookies
…
```

To use this as a default route with a real application:

1. Replace the test ReverseProxyHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

   In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.

2. Change the `uri`, `form`, and `baseURI` to match the target application.

3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

Login form with cookie from login page

Like the previous route, this template route intercepts the login page request, replaces it with the login form, and logs the user into the target application with hard-coded username and password. This route also adds a CookieFilter to manage cookies.

The route uses a default `CookieFilter` to manage cookies. In this default configuration, cookies from the protected application are intercepted and stored in the IG session. They are not sent to the browser. For information, see CookieFilter.

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert for HTTPS.",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            },
            "hostnameVerifier": "ALLOW_ALL"
          }
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${request.uri.path == '/login'}",
            "request": {
              "method": "POST",
              "uri": "https://app.example.com:8444/login",
              "form": {
                "username": [
                  "MY_USERNAME"
                ],
                "password": [
                  "MY_PASSWORD"
                ]
              }
            }
          }
        },
        {
          "type": "CookieFilter"
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.query, 'demo=cookie')}"
}
```

To try this example with the sample application:

1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/22-cookie.json
```

Windows

```
%appdata%\OpenIG\config\routes\22-cookie.json
```

2. Replace `MY_USERNAME` with `kramer`, and `MY_PASSWORD` with `N3wman12`.
3. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

4. Go to http://ig.example.com:8080/login?demo=cookie.

   The sample application page is displayed.

```
Method    POST
URI       /login
Cookies
Headers  content-type: application/x-www-form-urlencoded
         content-length: 31
         host: app.example.com:8444
         connection: Keep-Alive
         user-agent: Apache-HttpAsyncClient/… (Java/…)
```

5. Refresh your connection to http://ig.example.com:8080/login?demo=cookie.

   Compared to the example in Login form with cookie from login page, this example displays additional information about the session cookie:

   ```
   Cookies  session-cookie=123…
   ```

To use this as a default route with a real application:

1. Replace the test ReverseProxyHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

   In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.

2. Change the `uri` and `form` to match the target application.

3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

Login form with password replay and cookie filters

When a user without a valid session tries to access a protected application, this template route works with an application to return a login page.

The route uses a PasswordReplayFilter to find the login page by using a pattern that matches a mock AM Classic UI page.

Cookies sent by the user agent are retained in the CookieFilter, and not forwarded to the protected application. Similarly, set-cookies sent by the protected application are retained in the CookieFilter and not forwarded back to the user agent.

The route uses a default `CookieFilter` to manage cookies. In this default configuration, cookies from the protected application are intercepted and stored in the IG session. They are not sent to the browser. For information, see CookieFilter.

```
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPageContentMarker": "OpenAM\\s\\(Login\\)",
            "request": {
              "comments": [
                "An example based on OpenAM classic UI: ",
                "uri is for the OpenAM login page; ",
                "IDToken1 is the username field; ",
                "IDToken2 is the password field; ",
                "host takes the OpenAM FQDN:port.",
                "The sample app simulates OpenAM."
              ],
              "method": "POST",
              "uri": "http://app.example.com:8081/openam/UI/Login",
              "form": {
                "IDToken0": [
                  ""
                ],
                "IDToken1": [
                  "demo"
                ],
                "IDToken2": [
                  "Ch4ng31t"
                ],
                "IDButton": [
                  "Log+In"
                ],
                "encoded": [
                  "false"
                ]
              },
              "headers": {
                "host": [
                  "app.example.com:8081"
                ]
              }
            }
          }
        },
        {
          "type": "CookieFilter"
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.query, 'demo=classic')}"
}
```

To try this example with the sample application:

1. Save the file as `$HOME/.openig/config/routes/23-classic.json`.

2. Use the following `curl` command to check that it works:

```
$ curl -D- http://ig.example.com:8080/login?demo=classic

HTTP/1.1 200 OK
Set-Cookie: IG_SESSIONID=24446BA29E866F840197C8E0EAD57A89; Path=/; HttpOnly
...
```

To use this as a default route with a real application:

1. Change the `uri` and `form` to match the target application.
2. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

Login which requires a hidden value from the login page

This template route extracts a hidden value from the login page, and includes it the static login form that it then POSTs to the target application.

```
{
  "properties": {
    "appBaseUri":  "https://app.example.com:8444"
  },
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert for HTTPS.",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            },
            "hostnameVerifier": "ALLOW_ALL"
          }
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${request.uri.path == '/login'}",
            "loginPageExtractions": [
              {
                "name": "hidden",
                "pattern": "loginToken\\s+value=\"(.*)\""
              }
            ],
            "request": {
              "method": "POST",
              "uri": "${appBaseUri}/login",
              "form": {
                "username": [
                  "MY_USERNAME"
                ],
                "password": [
                  "MY_PASSWORD"
                ],
                "hiddenValue": [
                  "${attributes.extracted.hidden}"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.query, 'demo=hidden')}",
  "baseURI": "${appBaseUri}"
}
```

The parameters in the PasswordReplayFilter form, `MY_USERNAME` and `MY_PASSWORD`, can have string values or can use expressions.

To try this example with the sample application:

1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/24-hidden.json
```

Windows

```
%appdata%\OpenIG\config\routes\24-hidden.json
```

2. Replace `MY_USERNAME` with `scarter`, and `MY_PASSWORD` with `S9rain12`.

3. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

4. Go to http://ig.example.com:8080/login?demo=hidden.

To use this as a default route with a real application:

1. Replace the test ReverseProxyHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

   In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.

2. Change the `loginPage`, `loginPageExtractions`, `uri`, and `form` to match the target application.

3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

HTTP and HTTPS application

This template route proxies traffic to an application with both HTTP and HTTPS ports. The application uses HTTPS for authentication and HTTP for the general application features. Assuming that all login requests are made over HTTPS, you must add the login filters and handlers to the chain.

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert for HTTPS.",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            },
            "hostnameVerifier": "ALLOW_ALL"
          }
        }
      }
    }
  ],
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${request.uri.scheme == 'http'}",
          "handler": "ReverseProxyHandler",
          "baseURI": "http://app.example.com:8081"
        },
        {
          "condition": "${request.uri.path == '/login'}",
          "handler": {
            "type": "Chain",
            "config": {
              "comment": "Add one or more filters to handle login.",
              "filters": [],
              "handler": "ReverseProxyHandler"
            }
          },
          "baseURI": "https://app.example.com:8444"
        },
        {
          "handler": "ReverseProxyHandler",
          "baseURI": "https://app.example.com:8444"
        }
      ]
    }
  },
  "condition": "${find(request.uri.query, 'demo=https')}"
}
```

To try this example with the sample application:

1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/25-https.json
```

Windows

```
%appdata%\OpenIG\config\routes\25-https.json
```

2. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

3. Go to http://ig.example.com:8080/login?demo=https.

   The login page of the sample application is displayed.

To use this as a default route with a real application:

1. Replace the test ReverseProxyHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.

2. Change the `loginPage`, `loginPageExtractions`, `uri`, and `form` to match the target application.
3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

1. Replace the test ReverseProxyHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

   In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.

2. Change the `loginPage`, `loginPageExtractions`, `uri`, and `form` to match the target application.
3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

AM integration with headers

This template route logs the user into the target application by using headers such as those passed in from an AM policy agent. If the passed in header contains only a user name or subject and requires a lookup to an external data source, you must add an attribute filter to the chain to retrieve the credentials.

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert for HTTPS.",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            },
            "hostnameVerifier": "ALLOW_ALL"
          }
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${request.uri.path == '/login'}",
            "request": {
              "method": "POST",
              "uri": "https://app.example.com:8444/login",
              "form": {
                "username": [
                  "${request.headers['username'][0]}"
                ],
                "password": [
                  "${request.headers['password'][0]}"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.query, 'demo=headers')}"
}
```

To try this example with the sample application:

1. Add the route to IG:

Linux

```
$HOME/.openig/config/routes/26-headers.json
```

Windows

```
%appdata%\OpenIG\config\routes\26-headers.json
```

2. Use the `curl` command to simulate the headers being passed in from an AM policy agent, as in the following example:

```
$ curl \
--header "username: kvaughan" \
--header "password: B5ibery12" \
http://ig.example.com:8080/login?demo=headers

...
<title id="welcome">Howdy, kvaughan</title>
...
```

To use this as a default route with a real application:

1. Replace the test ReverseProxyHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ReverseProxyHandler that references a truststore holding the certificate.

   Configure the ReverseProxyHandler to strictly verifiy hostnames for outgoing SSL connections.

   In production, do not use `TrustAllManager` for `trustManager`, or `ALLOW_ALL` for `hostnameVerifier`.

2. Change the `loginPage`, `uri`, and `form` to match the target application.

3. Remove the route-level condition on the handler that specifies a `demo` query string parameter.

Extend

To achieve complex server interactions or intensive data transformations that you can't currently achieve with scripts or existing handlers, filters, or expressions, extend IG through scripting and customization. The following sections describe how to extend IG:

Add .jar files for extensions

IG includes a complete Java application programming interface for extending your deployment with customizations. For more information, refer to Extend IG through the Java API

Create a directory to hold .jar files for IG extensions:

Linux

```
$HOME/.openig/extra
```

Windows

```
%appdata%\OpenIG\extra
```

When IG starts up, the JVM loads .jar files in the `extra` directory.

Extend IG through scripts

The following sections describe how to extend IG through scripts:

About scripts

Important

When writing scripts or Java extensions that use the Promise API, avoid the blocking methods `get()`, `getOrThrow()`, and `getOrThrowUninterruptibly()`. A promise represents the result of an asynchronous operation; therefore, using a blocking method to wait for the result can cause deadlocks and/or race issues.
Instead, consider using `then()` methods, such as `thenOnResult()`, `thenAsync()`, or `thenCatch()`, which allow execution blocks to be executed when the response is available.

Blocking code example

```
def response = next.handle(ctx, req).get() // Blocking method 'get' used
response.headers['new']="new header value"
return response
```

Non-blocking code example

```
return next.handle(ctx, req)
            //Process result when it is available
           .thenOnResult { response ->
             response.headers['new']="new header value"
           }
```

IG supports the Groovy dynamic scripting language through the use the scriptable objects. For information about scriptable object types, their configuration, and properties, refer to Scripts.

Scriptable objects are configured by the script's Internet media type, and either a source script included in the JSON configuration, or a file script that IG reads from a file. The configuration can optionally supply arguments to the script.

IG provides global variables to scripts at runtime, and provides access to Groovy's built-in functionality. Scripts can access the request and the context, store variables across executions, write messages to logs, make requests to a web service, and access responses returned in promise callback methods.

Before trying the scripts in this chapter, install and configure IG as described in the Quick install.

When developing and debugging your scripts, consider configuring a capture decorator to log requests, responses, and context data in JSON form. You can then turn off capturing when you move to production. For information, refer to CaptureDecorator.

Use a reference file script

The following example defines a ScriptableFilter written in Groovy, and stored in the following file:

Linux

```
$HOME/.openig/scripts/groovy/SimpleFormLogin.groovy
```

Windows

```
%appdata%\OpenIG\scripts\groovy\SimpleFormLogin.groovy
```

```
{
  "name": "SimpleFormLogin",
  "type": "ScriptableFilter",
  "config": {
    "type": "application/x-groovy",
    "file": "SimpleFormLogin.groovy"
  }
}
```

Relative paths in the file field depend on how IG is installed. If IG is installed in an application server, then paths for Groovy scripts are relative to `$HOME/.openig/scripts/groovy` (or `%appdata%\OpenIG\scripts\groovy`).

The base location `$HOME/.openig/scripts/groovy` (or `%appdata%\OpenIG\scripts\groovy`) is on the classpath when the scripts are executed. If some Groovy scripts are not in the default package, but instead have their own package names, they belong in the directory corresponding to their package name. For example, a script in package `com.example.groovy` belongs in `$HOME/.openig/scripts/groovy/com/example/groovy/` (or `%appdata%\OpenIG\scripts\groovy\com\example\groovy\`).

Scripts in Studio

You can use Studio to configure a ScriptableFilter or scriptableThrottlingPolicy, or use scripts to configure scopes in OAuth2ResourceServerFilter.

During configuration, you can enter the script directly into the object, or you can use a stored reference script. Note the following points about creating and using reference scripts:

- When you enter a script directly into an object, the script is added to the list of reference scripts.
- You can use a reference script in multiple objects in a route, but if you edit a reference script, all objects that use it are updated with the change.
- If you delete an object that uses a script, or remove the object from the chain, the script that it references remains in the list of scripts.
- If a reference script is used in an object, you can't rename or delete the script.

For an example of creating a ScriptableThrottlingPolicy in Studio, refer to Configure Scriptable Throttling. For information about using Studio, refer to Adding Configuration to a Route.

Script dispatch

To route requests when the conditions are complicated, use a `ScriptableHandler` instead of a `DispatchHandler` as described in DispatchHandler.

1. Add the following script to IG:

Linux

```
$HOME/.openig/scripts/groovy/DispatchHandler.groovy
```

Windows

```
%appdata%\OpenIG\scripts\groovy\DispatchHandler.groovy
```

```
/*
 * This simplistic dispatcher matches the path part of the HTTP request.
 * If the path is /mylogin, it checks Username and Password headers,
 * accepting bjensen:H1falutin, and returning HTTP 403 Forbidden to others.
 * Otherwise it returns HTTP 401 Unauthorized.
 */

// Rather than returning a Promise of a Response from an external source,
// this script returns the response itself.
response = new Response(Status.OK);

switch (request.uri.path) {

    case "/mylogin":

        if (request.headers.Username.values[0] == "bjensen" &&
                request.headers.Password.values[0] == "H1falutin") {

            response.status = Status.OK
            response.entity = "<html><p>Welcome back, Babs!</p></html>"

        } else {

            response.status = Status.FORBIDDEN
            response.entity = "<html><p>Authorization required</p></html>"

        }

        break

    default:

        response.status = Status.UNAUTHORIZED
        response.entity = "<html><p>Please <a href='./mylogin'>log in</a>.</p></html>"

        break

}

// Return the locally created response, no need to wrap it into a Promise
return response
```

2. Add the following route to IG, to set up headers required by the script when the user logs in:

Linux

```
$HOME/.openig/config/routes/98-dispatch.json
```

Windows

```
%appdata%\OpenIG\config\routes\98-dispatch.json
```

```
{
  "heap": [
    {
      "name": "DispatchHandler",
      "type": "DispatchHandler",
      "config": {
        "bindings": [{
          "condition": "${find(request.uri.path, '/mylogin')}",
          "handler": {
            "type": "Chain",
            "config": {
              "filters": [
                {
                  "type": "HeaderFilter",
                  "config": {
                    "messageType": "REQUEST",
                    "add": {
                      "Username": [
                        "bjensen"
                      ],
                      "Password": [
                        "H1falutin"
                      ]
                    }
                  }
                }
              ],
              "handler": "Dispatcher"
            }
          }
        },
          {
            "handler": "Dispatcher",
            "condition": "${find(request.uri.path, '/dispatch')}"
          }
        ]
      }
    },
    {
      "name": "Dispatcher",
      "type": "ScriptableHandler",
      "config": {
        "type": "application/x-groovy",
        "file": "DispatchHandler.groovy"
      }
    }
  ],
  "handler": "DispatchHandler",
  "condition": "${find(request.uri.path, '^/dispatch') or find(request.uri.path, '^/mylogin')}"
}
```

3. Go to http://ig.example.com:8080/dispatch, and click `log in`.

   The HeaderFilter sets `Username` and `Password` headers in the request, and passes the request to the script. The script responds, `Welcome back, Babs!`

Script HTTP basic access authentication

HTTP basic access authentication is a simple challenge and response mechanism, where a server requests credentials from a client, and the client passes them to the server in an `Authorization` header. The credentials are base-64 encoded. To protect them, use SSL encryption for the connections between the server and client. For more information, refer to RFC 2617.

1. Add the following script to IG, to add an `Authorization` header based on a username and password combination:

Linux

```
$HOME/.openig/scripts/groovy/BasicAuthFilter.groovy
```

Windows

```
%appdata%\OpenIG\scripts\groovy\BasicAuthFilter.groovy
```

```
/*
 * Perform basic authentication with the user name and password
 * that are supplied using a configuration like the following:
 *
 * {
 *     "name": "BasicAuth",
 *     "type": "ScriptableFilter",
 *     "config": {
 *         "type": "application/x-groovy",
 *         "file": "BasicAuthFilter.groovy",
 *         "args": {
 *             "username": "bjensen",
 *             "password": "H1falutin"
 *             }
 *         }
 * }
 */

def userPass = username + ":" + password
def base64UserPass = userPass.getBytes().encodeBase64()
request.headers.add("Authorization", "Basic ${base64UserPass}" as String)

// Credentials are only base64-encoded, not encrypted: Set scheme to HTTPS.

/*
 * When connecting over HTTPS, by default the client tries to trust the server.
 * If the server has no certificate
 * or has a self-signed certificate unknown to the client,
 * then the most likely result is an SSLPeerUnverifiedException.
 *
 * To avoid an SSLPeerUnverifiedException,
 * set up HTTPS correctly on the server.
 * Either use a server certificate signed by a well-known CA,
 * or set up the gateway to trust the server certificate.
 */
request.uri.scheme = "https"

// Calls the next Handler and returns a Promise of the Response.
// The Response can be handled with asynchronous Promise callbacks.
next.handle(context, request)
```

2. Add the following route to IG, to set up headers required by the script when the user logs in:

Linux

```
$HOME/.openig/config/routes/09-basic.json
```

Windows

```
%appdata%\OpenIG\config\routes\09-basic.json
```

```
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ScriptableFilter",
          "config": {
            "type": "application/x-groovy",
            "file": "BasicAuthFilter.groovy",
            "args": {
              "username": "bjensen",
              "password": "H1falutin"
            }
          },
          "capture": "filtered_request"
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/plain; charset=UTF-8" ]
          },
          "entity": "Hello bjensen!"
        }
      }
    }
  },
  "condition": "${find(request.uri.path, '^/basic')}"
}
```

When the request path matches `/basic`, the route calls the Chain, which runs the ScriptableFilter. The capture setting captures the request as updated by the ScriptableFilter. Finally, IG returns a static page.

3. Go to http://ig.example.com:8080/basic.

   The captured request in the console log shows that the scheme is now HTTPS, and that the `Authorization` header is set for HTTP Basic:

```
GET https://app.example.com:8081/basic HTTP/1.1
 ...
 Authorization: Basic Ymp...aW4=
```

Script SQL queries

> **Important**
>
> The example in this section uses SqlClient, which exposes a JdbcDataSource. Because the JDBC API provides only blocking APIs, using SqlClient to execute long operations can cause deadlocks and/or race issues.
> Consider updating the example in this section to offload JdbcDataSource calls to another thread.

This example builds on Password replay from a database to use scripts to look up credentials in a database, set the credentials in headers, and set the scheme in HTTPS to protect the request.

1. Set up and test the example in Password replay from a database.
2. Add the following script to IG, to look up user credentials in the database, by email address, and set the credentials in the request headers for the next handler:

Linux

```
$HOME/.openig/scripts/groovy/SqlAccessFilter.groovy
```

Windows

```
%appdata%\OpenIG\scripts\groovy\SqlAccessFilter.groovy
```

```
/*
 * Look up user credentials in a relational database
 * based on the user's email address provided in the request form data,
 * and set the credentials in the request headers for the next handler.
 */

def client = new SqlClient(dataSource)
def credentials = client.getCredentials(request.queryParams?.mail[0])
request.headers.add("Username", credentials.Username)
request.headers.add("Password", credentials.Password)

// The credentials are not protected in the headers, so use HTTPS.
request.uri.scheme = "https"

// Calls the next Handler and returns a Promise of the Response.
// The Response can be handled with asynchronous Promise callbacks.
next.handle(context, request)
```

3. Add the following script to IG to access the database, and get credentials:

Linux

```
$HOME/.openig/scripts/groovy/SqlClient.groovy
```

Windows

```
%appdata%\OpenIG\scripts\groovy\SqlClient.groovy
```

```
import groovy.sql.Sql

import javax.sql.DataSource

/**
 * Access a database with a well-known structure,
 * in particular to get credentials given an email address.
 */
class SqlClient {

    // DataSource supplied as constructor parameter.
    def sql

    SqlClient(DataSource dataSource) {
        if (dataSource == null) {
            throw new IllegalArgumentException("DataSource is null")
        }
        this.sql = new Sql(dataSource)
    }

    // The expected table is laid out like the following.

    // Table USERS
    // ----------------------------------------
    // | USERNAME  | PASSWORD |   EMAIL   |...|
    // ----------------------------------------
    // | <username>| <passwd> | <mail@...>|...|
    // ----------------------------------------

    String tableName = "USERS"
    String usernameColumn = "USERNAME"
    String passwordColumn = "PASSWORD"
    String mailColumn = "EMAIL"

    /**
     * Get the Username and Password given an email address.
     *
     * @param mail Email address used to look up the credentials
     * @return Username and Password from the database
     */
    def getCredentials(mail) {
        def credentials = [:]
        def query = "SELECT " + usernameColumn + ", " + passwordColumn +
                " FROM " + tableName + " WHERE " + mailColumn + "='$mail';"

        sql.eachRow(query) {
            credentials.put("Username", it."$usernameColumn")
            credentials.put("Password", it."$passwordColumn")
        }
        return credentials
    }
}
```

4. Add the following route to IG to set up headers required by the scripts when the user logs in:

Linux

```
$HOME/.openig/config/routes/11-db.json
```

Windows

```
%appdata%\OpenIG\config\routes\11-db.json
```

```
{
  "heap":  [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "JdbcDataSource-1",
      "type": "JdbcDataSource",
      "config": {
        "driverClassName": "org.h2.Driver",
        "jdbcUrl": "jdbc:h2:tcp://localhost/~/test",
        "username": "sa",
        "passwordSecretId": "database.password",
        "secretsProvider": "SystemAndEnvSecretStore-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ScriptableFilter",
          "config": {
            "args": {
              "dataSource": "${heap['JdbcDataSource-1']}"
            },
            "type": "application/x-groovy",
            "file": "SqlAccessFilter.groovy"
          }
        },
        {
          "type": "StaticRequestFilter",
          "config": {
            "method": "POST",
            "uri": "http://app.example.com:8081/login",
            "form": {
              "username": [
                "${request.headers['Username'][0]}"
              ],
              "password": [
                "${request.headers['Password'][0]}"
              ]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.path, '^/db')}"
}
```

Notice the following features of the route:

- The route matches requests to `/db`.
- The JdbcDataSource in the heap sets up the connection to the database.

- The ScriptableFilter calls `SqlAccessFilter.groovy` to look up credentials over SQL.

  `SqlAccessFilter.groovy`, in turn, calls `SqlClient.groovy` to access the database to get the credentials.

- The StaticRequestFilter uses the credentials to build a login request.

  Although the script sets the scheme to HTTPS, for convenience in this example, the StaticRequestFilter resets the URI to HTTP.

5. To test the setup, go to a URL with a query string parameter that specifies an email address in the database, such as `http://ig.example.com:8080/db?mail=george@example.com`.

   The sample application profile page for the user is displayed.

Extend IG through the Java API

Important

When writing scripts or Java extensions that use the Promise API, avoid the blocking methods `get()`, `getOrThrow()`, and `getOrThrowUninterruptibly()`. A promise represents the result of an asynchronous operation; therefore, using a blocking method to wait for the result can cause deadlocks and/or race issues.
Instead, consider using `then()` methods, such as `thenOnResult()`, `thenAsync()`, or `thenCatch()`, which allow execution blocks to be executed when the response is available.

Blocking code example

```
def response = next.handle(ctx, req).get() // Blocking method 'get' used
response.headers['new']="new header value"
return response
```

Non-blocking code example

```
return next.handle(ctx, req)
            //Process result when it is available
           .thenOnResult { response ->
             response.headers['new']="new header value"
           }
```

IG includes a complete Java application programming interface to allow you to customize IG to perform complex server interactions or intensive data transformations that you cannot achieve with scripts or the existing handlers, filters, and expressions described in Expressions. The following sections describe how to extend IG through the Java API:

Key extension points

Interface Stability: Evolving, as defined in ForgeRock product stability labels.

The following interfaces are available:

Decorator

A `Decorator` adds new behavior to another object without changing the base type of the object.

When suggesting custom `Decorator` names, know that IG reserves all field names that use only alphanumeric characters. To avoid clashes, use dots or dashes in your field names, such as `my-decorator`.

ExpressionPlugin

An `ExpressionPlugin` adds a node to the `Expression` context tree, alongside `env` (for environment variables), and `system` (for system properties). For example, the expression `${system['user.home']}` yields the home directory of the user running the application server for IG.

In your `ExpressionPlugin`, the `getKey()` method returns the name of the node, and the `getObject()` method returns the unified expression language context object that contains the values needed to resolve the expression. The plugins for `env` and `system` return Map objects, for example.

When you add your own `ExpressionPlugin`, you must make it discoverable within your custom library. You do this by adding a services file named after the plugin interface, where the file contains the fully qualified class name of your plugin, under `META-INF/services/org.forgerock.openig.el.ExpressionPlugin` in the .jar file for your customizations. When you have more than one plugin, add one fully qualified class name per line. For information, refer to the reference documentation for the Java class ServiceLoader. If you build your project using Maven, then you can add this under the `src/main/resources` directory. Add custom libraries, as described in Embed customizations in IG.

Remember to provide documentation for IG administrators on how your plugin extends expressions.

Filter

A `Filter` serves to process a request before handing it off to the next element in the chain, in a similar way to an interceptor programming model.

The `Filter` interface exposes a `filter()` method, which takes a Context, a Request, and the Handler, which is the next filter or handler to dispatch to. The `filter()` method returns a Promise that provides access to the Response with methods for dealing with both success and failure conditions.

A filter can elect not to pass the request to the next filter or handler, and instead handle the request itself. It can achieve this by merely avoiding a call to `next.handle(context, request)`, creating its own response object and returning that in the promise. The filter is also at liberty to replace a response with another of its own. A filter can exist in more than one chain, therefore should make no assumptions or correlations using the chain it is supplied. The only valid use of a chain by a filter is to call its `handle()` method to dispatch the request to the rest of the chain.

Handler

A `Handler` generates a response for a request.

The `Handler` interface exposes a `handle()` method, which takes a Context, and a Request. It processes the request and returns a Promise that provides access to the Response with methods for dealing with both success and failure conditions. A handler can elect to dispatch the request to another handler or chain.

ClassAliasResolver

A `ClassAliasResolver` makes it possible to replace a fully qualified class name with a short name (an alias) in an object declaration's type.

The `ClassAliasResolver` interface exposes a `resolve(String)` method to do the following:

- Return the class mapped to a given alias
- Return `null` if the given alias is unknown to the resolver

  All resolvers available to IG are asked until the first non-null value is returned or until all resolvers have been contacted.

  The order of resolvers is nondeterministic. To prevent conflicts, don't use the same alias for different types.

Implement a customized sample filter

The `SampleFilter` class implements the `Filter` interface to set a header in the incoming request and in the outgoing response.

In the following example, the sample filter adds an arbitrary header:

```
package org.forgerock.openig.doc.examples;

import org.forgerock.http.Filter;
import org.forgerock.http.Handler;
import org.forgerock.http.protocol.Request;
import org.forgerock.http.protocol.Response;
import org.forgerock.openig.heap.GenericHeaplet;
import org.forgerock.openig.heap.HeapException;
import org.forgerock.openig.model.type.service.NoTypeInfo;
import org.forgerock.services.context.Context;
import org.forgerock.util.promise.NeverThrowsException;
import org.forgerock.util.promise.Promise;

/**
 * Filter to set a header in the incoming request and in the outgoing response.
 */
public class SampleFilter implements Filter {

    /** Header name. */
    String name;

    /** Header value. */
    String value;

    /**
     * Set a header in the incoming request and in the outgoing response.
     * A configuration example looks something like the following.
     *
     * <pre>
     * {
     *     "name": "SampleFilter",
     *     "type": "SampleFilter",
     *     "config": {
     *         "name": "X-Greeting",
     *         "value": "Hello world"
     *     }
     * }
     * </pre>
     *
     * @param context           Execution context.
     * @param request           HTTP Request.
     * @param next              Next filter or handler in the chain.
     * @return A {@code Promise} representing the response to be returned to the client.
     */
    @Override
    public Promise<Response, NeverThrowsException> filter(final Context context,
                                                          final Request request,
                                                          final Handler next) {

        // Set header in the request.
        request.getHeaders().put(name, value);

        // Pass to the next filter or handler in the chain.
        return next.handle(context, request)
                   // When it has been successfully executed, execute the following callback
                   .thenOnResult(response -> {
                       // Set header in the response.
                       response.getHeaders().put(name, value);
                   });
    }
```

```
    /**
     * Create and initialize the filter, based on the configuration.
     * The filter object is stored in the heap.
     */
    @NoTypeInfo
    public static class Heaplet extends GenericHeaplet {

        /**
         * Create the filter object in the heap,
         * setting the header name and value for the filter,
         * based on the configuration.
         *
         * @return                  The filter object.
         * @throws HeapException    Failed to create the object.
         */
        @Override
        public Object create() throws HeapException {

            SampleFilter filter = new SampleFilter();
            filter.name  = config.get("name").as(evaluatedWithHeapProperties()).required().asString();
            filter.value = config.get("value").as(evaluatedWithHeapProperties()).required().asString();

            return filter;
        }
    }
}
```

The corresponding filter configuration is similar to this:

```
{
  "name": "SampleFilter",
  "type": "org.forgerock.openig.doc.examples.SampleFilter",
  "config": {
    "name": "X-Greeting",
    "value": "Hello world"
  }
}
```

Note how `type` is configured with the fully qualified class name for `SampleFilter`. To simplify the configuration, implement a class alias resolver, as described in Implement a Class Alias Resolver.

Implement a class alias resolver

To simplify the configuration of a customized object, implement a `ClassAliasResolver` to allow the use of short names instead of fully qualified class names.

In the following example, a `ClassAliasResolver` is created for the `SampleFilter` class:

```
package org.forgerock.openig.doc.examples;

import static java.util.stream.Collectors.toUnmodifiableSet;

import java.util.HashMap;
import java.util.Map;
import java.util.Optional;
import java.util.Set;

import org.forgerock.openig.alias.ClassAliasResolver;
import org.forgerock.openig.heap.Heaplet;
import org.forgerock.openig.heap.Heaplets;

/**
 * Allow use of short name aliases in configuration object types.
 *
 * This allows a configuration with {@code "type": "SampleFilter"}
 * instead of {@code "type": "org.forgerock.openig.doc.examples.SampleFilter"}.
 */
public class SampleClassAliasResolver implements ClassAliasResolver {

    private static final Map<String, Class<?>> ALIASES =
            new HashMap<>();

    static {
        ALIASES.put("SampleFilter", SampleFilter.class);
    }

    /**
     * Get the class for a short name alias.
     *
     * @param alias Short name alias.
     * @return      The class, or null if the alias is not defined.
     */
    @Override
    public Class<?> resolve(final String alias) {
        return ALIASES.get(alias);
    }

    @Override
    public Set<Class<? extends Heaplet>> supportedTypes() {
        return ALIASES.values()
                      .stream()
                      .map(Heaplets::findHeapletClass)
                      .filter(Optional::isPresent)
                      .map(Optional::get)
                      .collect(toUnmodifiableSet());
    }
}
```

With this `ClassAliasResolver`, the filter configuration in Implement a Customized Sample Filter can use the alias instead of the fully qualified class name, as follows:

```
{
  "name": "SampleFilter",
  "type": "SampleFilter",
  "config": {
    "name": "X-Greeting",
    "value": "Hello world"
  }
}
```

To create a customized `ClassAliasResolver`, add a services file with the following characteristics:

- Name the file after the class resolver interface.
- Store the file under `META-INF/services/org.forgerock.openig.alias.ClassAliasResolver`, in the customization .jar file.

  If you build your project using Maven, you can add the file under the `src/main/resources` directory.
- In your ClassAliasResolver file, add a line for the fully qualified class name of your resolver as follows:

  ```
  org.forgerock.openig.doc.examples.SampleClassAliasResolver
  ```

  If you have more than one resolver in your .jar file, add one line for each fully qualified class name.

Configure the heap object for the customization

Objects are added to the heap and supplied with configuration artifacts at initialization time. To be integrated with the configuration, a class must have an accompanying implementation of the Heaplet interface. The easiest and most common way of exposing the heaplet is to extend the GenericHeaplet class in a nested class of the class you want to create and initialize, overriding the heaplet's `create()` method.

Within the `create()` method, you can access the object's configuration through the `config` field.

Embed customizations in IG

1. Build your IG extension into a .jar file.
2. Create the directory `$HOME/.openig/extra`, where `$HOME/.openig` is the instance directory:

   ```
   $ mkdir $HOME/.openig/extra
   ```

3. Add the .jar file to the directory. The following example adds `sample-filter.jar` to `$HOME/.openig/extra`:

   ```
   $ cp ~/sample-filter/target/sample-filter.jar $HOME/.openig/extra
   ```

4. If the extension has dependencies that are not included in IG, also add them to the directory.
5. Start IG, as described in Start and stop IG.

Record custom audit events

This section describes how to record a custom audit event to standard output. The example is based on the example in Validate access tokens through the introspection endpoint, adding an audit event for the custom topic `OAuth2AccessTopic`.

To record custom audit events to other outputs, adapt the route in the following procedure to use another audit event handler.

For information about how to configure supported audit event handlers, and exclude sensitive data from log files, refer to Audit the deployment. For more information about audit event handlers, refer to Audit framework.

Record custom audit events to standard output

Before you start, prepare IG and the sample application as described in the Quick install.

1. Set up AM as described in Validate access tokens through the introspection endpoint.
2. Define the schema of an event topic called `OAuth2AccessTopic` by adding the following route to IG:

Linux

```
$HOME/.openig/audit-schemas/OAuth2AccessTopic.json
```

Windows

```
%appdata%\OpenIG\audit-schemas/OAuth2AccessTopic.json
```

```
{
  "schema": {
    "$schema": "http://json-schema.org/draft-04/schema#",
    "id": "OAuth2Access",
    "type": "object",
    "properties": {
      "_id": {
        "type": "string"
      },
      "timestamp": {
        "type": "string"
      },
      "transactionId": {
        "type": "string"
      },
      "eventName": {
        "type": "string"
      },
      "accessToken": {
        "type": "object",
        "properties": {
          "scopes": {
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          "expiresAt": "number",
          "sub": "string"
        },
        "required": [ "scopes" ]
      },
      "resource": {
        "type": "object",
        "properties": {
          "path": {
            "type": "string"
          },
          "method": {
            "type": "string"
          }
        }
      }
    }
  },
  "filterPolicies": {
    "field": {
      "includeIf": [
        "/_id",
        "/timestamp",
        "/eventName",
        "/transactionId",
        "/accessToken",
        "/resource"
      ]
    }
  },
  "required": [ "_id", "timestamp", "transactionId", "eventName" ]
}
```

Notice that the schema includes the following fields:

- Mandatory fields `_id`, `timestamp`, `transactionId`, and `eventName`.
- `accessToken`, to include the access token scopes, expiry time, and the subject.
- `resource`, to include the path and method.
- `filterPolicies`, to specify additional event fields to include in the logs.

3. Define a script to generate audit events on the topic named `OAuth2AccessTopic`, by adding the following file to the IG configuration as:

Linux

```
$HOME/.openig/scripts/groovy/OAuth2Access.groovy
```

Windows

```
%appdata%\OpenIG\scripts\groovy/OAuth2Access.groovy
```

```
import static org.forgerock.json.resource.Requests.newCreateRequest;
import static org.forgerock.json.resource.ResourcePath.resourcePath;

// Helper functions
def String transactionId() {
    return contexts.transactionId.transactionId.value;
}

def JsonValue auditEvent(String eventName) {
    return json(object(field('eventName', eventName),
            field('transactionId', transactionId()),
            field('timestamp', clock.instant().toEpochMilli())));
}

def auditEventRequest(String topicName, JsonValue auditEvent) {
    return newCreateRequest(resourcePath("/" + topicName), auditEvent);
}

def accessTokenInfo() {
    def accessTokenInfo = contexts.oauth2.accessToken;
    return object(field('scopes', accessTokenInfo.scopes as List),
            field('expiresAt', accessTokenInfo.expiresAt),
            field('subname', accessTokenInfo.info.subname));
}

def resourceEvent() {
    return object(field('path', request.uri.path),
            field('method', request.method));
}

// ----------------------------------------------

// Build the event
JsonValue auditEvent = auditEvent('OAuth2AccessEvent')
        .add('accessToken', accessTokenInfo())
        .add('resource', resourceEvent());

// Send the event, and log a message if there is an error
auditService.handleCreate(context, auditEventRequest("OAuth2AccessTopic", auditEvent))
        .thenOnException(e -> logger.warn("An error occurred while sending the audit event", e));

// Continue onto the next filter
return next.handle(context, request)
```

The script generates audit events named `OAuth2AccessEvent`, on a topic named `OAuth2AccessTopic`. The events conform to the topic schema.

4. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

5. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/30-custom.json
```

Windows

```
%appdata%\OpenIG\config\routes\30-custom.json
```

```
{
  "name": "30-custom",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/rs-introspect-audit')}",
  "heap": [
    {
      "name": "AuditService-1",
      "type": "AuditService",
      "config": {
        "config": {},
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.json.stdout.JsonStdoutAuditEventHandler",
            "config": {
              "name": "jsonstdout",
              "elasticsearchCompatible": false,
              "topics": [
                "OAuth2AccessTopic"
              ]
            }
          }
        ]
      }
    },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "token-resolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
```

```
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        },
        {
          "type": "ScriptableFilter",
          "config": {
            "type": "application/x-groovy",
            "file": "OAuth2Access.groovy",
            "args": {
              "auditService": "${heap['AuditService-1']}",
              "clock": "${heap['Clock']}"
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></body></
html>"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/rs-introspect-audit`.
- The `accessTokenResolver` uses the token introspection endpoint to validate the access token.
- The HttpBasicAuthenticationClientFilter adds the credentials to the outgoing token introspection request.
- The ScriptableFilter uses the Groovy script `OAuth2Access.groovy` to generate audit events named `OAuth2AccessEvent`, with a topic named `OAuth2AccessTopic`.

- The audit service publishes the custom audit event to the JsonStdoutAuditEventHandler. A single line per audit event is published to standard output.

6. Test the setup

   1. In a terminal window, use a `curl` command similar to the following to retrieve an access token:

      ```
      $ mytoken=$(curl -s \
      --user "client-application:password" \
      --data "grant_type=password&username=demo&password=Ch4ng31t&scope=mail%20employeenumber" \
      http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
      ```

   2. Access the route, with the access_token returned in the previous step:

      ```
      $ curl -v http://ig.example.com:8080/rs-introspect-audit --header "Authorization: Bearer ${mytoken}"
      ```

      Information about the decoded access_token is returned.

   3. Search the standard output for an audit message like the following example, that includes an audit event on the topic `OAuth2AccessTopic`:

      ```
      {
        "_id": "fa2...-14",
        "timestamp": 155...541,
        "eventName": "OAuth2AccessEvent",
        "transactionId": "fa2...-13",
        "accessToken": {
          "scopes": ["employeenumber", "mail"],
          "expiresAt": 155...000,
          "subname": "demo"
        },
        "resource": {
          "path": "/rs-introspect-audit",
          "method": "GET"
        },
        "source": "audit",
        "topic": "OAuth2AccessTopic",
        "level": "INFO"
      }
      ```

Upgrade

This guide shows you how to upgrade IG software.

Read the Release notes before you upgrade.

Plan the upgrade

Do these planning tasks **before** you start an upgrade:

Planning task	Description
Find the upgrade path	Refer to Upgrade to see if you need a drop-in upgrade or a major upgrade.
Find out what changed	Read the release notes for all releases between the current version and the new version. Understand the new features and changes in the new version compared to the current version.
Check the requirements	Make sure you meet all the requirements in the release notes for the new version. In particular, make sure you have a recent, supported Java version.
Plan for server downtime	At least one of your IG servers will be down during upgrade. Plan to route client applications to another server until the upgrade process is complete and you have validated the result. Make sure the owners of client application are aware of the change, and let them know what to expect. If you have a single IG server, make sure the downtime happens in a low-usage window, and make sure you let client application owners plan accordingly.
Back up	The IG configuration is a set of files, including `admin.json`, `config.json`, `logback.xml`, routes, and scripts. Back up the IG configuration and store it in version control, so that you can roll back if something goes wrong. Back up any tools scripts you have edited for your deployment and any trust stores used to connect securely.
Plan for rollback	Sometimes even a well-planned upgrade fails to go smoothly. In such cases, you need a plan to roll back smoothly to the pre-upgrade version. For IG servers, roll back by restoring a backed-up configuration.
Prepare a test environment	Before applying the upgrade in your production environment, always try to upgrade IG in a test environment. This will help you gauge the amount of work required, without affecting your production environment, and will help smooth out unforeseen problems. The test environment should resemble your production environment as closely as possible.

Upgrade

Learn about upgrade between supported versions of IG in Product Support Lifecycle Policy | PingGateway and Agents.

Learn about upgrade of routes in Studio from Upgrade from an earlier version of Studio.

This section describes how to upgrade a single IG instance. The most straightforward option when upgrading sites with multiple IG instances is to upgrade in place. One by one, stop, upgrade, and then restart each server individually, leaving the service running during the upgrade.

IG supports the following types of upgrade:

Drop-in software update

Usually, an update from a version of IG to a newer minor version. For example, the update from 2023.2 to 2023.4.

Drop-in software updates can introduce additional functionality and fix bugs or security issues. Consider the following restrictions for drop-in software updates:

- Do not require any update to the configuration
- Cannot cause feature regression
- Can change default or previously configured behavior **only** for bug fixes and security issues
- Can deprecate **but not remove** existing functionality

Major upgrade

Usually, an upgrade from a version of IG to a newer major version. For example, the upgrade from 7.2 to 2023.2.

Major upgrades can introduce additional functionality and fix bugs or security issues. Major upgrades do not have the restrictions of drop-in software update. Consider the following features of major upgrades:

- Can require code or configuration changes
- Can cause feature regression
- Can change default or previously configured behavior
- Can deprecate **and** remove existing functionality

Drop-in software update with binaries

1. Read and act on Plan the upgrade.
2. Back up the IG configuration and store it in version control so that you can roll back if something goes wrong.
3. Download IG
4. Stop IG.
5. Make the new configuration available on the file system.

   By default, IG configuration files are located under `$HOME/.openig` (on Windows `%appdata%\OpenIG`). For information about how to use a different location, refer to Configuration location.
6. Restart IG from the new installation directory.
7. In a test environment that simulates your production environment, validate that the upgraded service performs as expected with the new configuration. Check the logs for new or unexpected notifications or errors.
8. Allow client application traffic to flow to the upgraded site.

Drop-in software update with Docker files

1. Read and act on Plan the upgrade.
2. Back up the IG configuration and store it in version control so that you can roll back if something goes wrong.
3. Stop the Docker image.
4. Build the new base image for IG.
5. Run the Docker image.
6. In a test environment that simulates your production environment, validate that the upgraded service performs as expected with the new configuration. Check the logs for new or unexpected notifications or errors.
7. Allow client application traffic to flow to the upgraded site.

Major upgrade with binaries

1. Read and act on Plan the upgrade.
2. Use the release notes for **all** releases between the version you currently use and the new version, and create a new configuration as follows:
   - Review all incompatible changes and removed functionality, and adjust your configuration as necessary.
   - Switch to the replacement settings for deprecated functionality. Although deprecated objects continue to work, they add to the notifications in the logs and are eventually removed.
   - Check the lists of fixes, limitations, and known issues to find out if they impact your deployment.
   - Recompile your Java extensions. The method signature or imports for supported and evolving APIs can change in each version.
   - Read the documentation updates for new examples and information that can help with your configuration.
3. Back up the IG configuration and store it in version control so that you can roll back if something goes wrong.
4. Download IG
5. Stop IG.
6. Make the new configuration available on the file system.

   By default, IG configuration files are located under `$HOME/.openig` (on Windows `%appdata%\OpenIG`). For information about how to use a different location, refer to Configuration location.
7. Restart IG from the new installation directory.
8. In a test environment that simulates your production environment, validate that the upgraded service performs as expected with the new configuration. Check the logs for new or unexpected notifications or errors.
9. Allow client application traffic to flow to the upgraded site.

Major upgrade with Docker files

1. Read and act on Plan the upgrade.
2. Use the release notes for **all** releases between the version you currently use and the new version, and create a new configuration as follows:
    - Review all incompatible changes and removed functionality, and adjust your configuration as necessary.
    - Switch to the replacement settings for deprecated functionality. Although deprecated objects continue to work, they add to the notifications in the logs and are eventually removed.
    - Check the lists of fixes, limitations, and known issues to find out if they impact your deployment.
    - Recompile your Java extensions. The method signature or imports for supported and evolving APIs can change in each version.
    - Read the documentation updates for new examples and information that can help with your configuration.
3. Back up the IG configuration and store it in version control so that you can roll back if something goes wrong.
4. Stop the Docker image.
5. Build the new base image for IG.
6. Run the Docker image.
7. In a test environment that simulates your production environment, validate that the upgraded service performs as expected with the new configuration. Check the logs for new or unexpected notifications or errors.
8. Allow client application traffic to flow to the upgraded site.

Post upgrade tasks

After upgrade, review the what's new section in the release notes and consider activating new features and functionality.

Rollback

> **Important**
>
> Before you roll back to a previous version of IG, consider whether any change to the configuration during or since upgrade could be incompatible with the previous version.

Roll back with binaries

1. Plan for server downtime

   Plan to route client applications to another server until the rollback process is complete and you have validated the result. Make sure the owners of client application are aware of the change, and let them know what to expect.
2. Stop IG
3. Download the replacement IG .zip file
4. Make the new configuration available on the file system.

By default, IG configuration files are located under `$HOME/.openig` (on Windows `%appdata%\OpenIG`). For information about how to use a different location, refer to Configuration location.

5. Restart IG.

Roll back with Dockerfiles

1. Plan for server downtime

   Plan to route client applications to another server until the rollback process is complete and you have validated the result. Make sure the owners of client application are aware of the change, and let them know what to expect.

2. Stop the Docker image.
3. Build the new base image for IG.
4. Run the Docker image.

Migrate from web container mode to standalone mode

An IG .war file isn't created or delivered from IG 2024.3. Consider these points when migrating from a .war delivery to a .zip delivery.

Session replication between IG instances

High-availability of sessions isn't supported by IG in the .zip delivery.

Streaming asynchronous responses and events

In ClientHandler and ReverseProxyHandler, use only the default mode of `asyncBehavior:non_streaming`; responses are processed when the entity content is entirely available.

If the property is set to `streaming`, the setting is ignored.

Connection reuse when client certificates are used for authentication

In ClientHandler and ReverseProxyHandler, use only the default mode of `stateTrackingEnabled:true`; when a client certificate is used for authentication, connections can't be reused.

If the property is set to `false`, the setting is ignored.

Replacement settings for migration from web container mode with Tomcat

Feature	Setting for web container mode with Tomcat	Replacement setting
Port number	Configure in the `Connector` element of `/path/to/tomcat/conf/server.xml`: `<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443" />`	Configure the `connectors` property of admin.json.
HTTPS server-side configuration	Create a keystore, and set up the SSL port in the `Connector` element of `/path/to/tomcat/conf/server.xml`.	Create a keystore, set up secrets, and configure secrets stores, ports, and ServerTlsOptions in admin.json. For information, refer to Configure IG for HTTPS (server-side).
Session cookie name	Configure `WEB-INF/web.xml` when you unpack the IG .war file.	Configure the `session` property of admin.json.
Access logs	Configure with `AccessLogValve`.	Configure in the Audit framework. For information, refer to Audit the deployment and Audit framework.
JDBC datasource	Configure in the `GlobalNamingResources` element of `/path/to/tomcat/conf/server.xml`.	Configure with the JdbcDataSource object. For information, refer to JdbcDataSource. For an example, refer to Password replay from a database.
Environment variables	Configure in `/path/to/tomcat/bin/setenv.sh`.	Configure in `$HOME/.openig/bin/env.sh`, where `$HOME/.openig` is the instance directory.
Jar files	Add to to web container classpath; for example `/path/to/tomcat/webapps/ROOT/WEB-INF/lib`.	Add to `$HOME/.openig/extra`, where `$HOME/.openig` is the instance directory.

Deploy with Docker

This guide shows you how to build an evaluation-only Docker image by using the Dockerfile provided inside `IG-2024.3.0.zip`.

> **Warning**
>
> ForgeRock provides no commercial support for production deployments that use ForgeRock's evaluation-only Docker images. When deploying the ForgeRock Identity Platform using Docker images, you must build and use your own images for production deployments.

This guide is for ForgeRock Identity Platform developers who want an easy-to-use example of containerized deployment, and for Identity Gateway developers who want to configure a production environment for containerized deployment.

For information about deploying ForgeRock Identity Platform by using DevOps techniques, refer to ForgeOps' Start here.

This guide assumes that you are familiar with the following topics:

- IG, to edit the basic configuration and test the changes.
- Docker, to build and run and Docker images.

The examples in this guide use some of the following third-party tools:

- `curl`: https://curl.haxx.se
- `HTTPie`: https://httpie.org
- `jq`: https://stedolan.github.io/jq/
- `keytool`: https://docs.oracle.com/en/java/javase/11/tools/keytool.html

Build and run a Docker image

{forgerock_name} delivers a Dockerfile inside `IG-2024.3.0.zip`, to help you build an evaluation-only, base Docker image for IG. After building and running the Docker image, add a configuration as described in Add configuration to a Docker image.

> **Warning**
>
> ForgeRock provides no commercial support for production deployments that use ForgeRock's evaluation-only Docker images. When deploying the ForgeRock Identity Platform using Docker images, you must build and use your own images for production deployments.

The Docker image has the following characteristics:

- The Docker image runs on Linux and Mac operating systems.
- IG binaries are delivered in `/opt/ig`.
- The environment variable `$IG_INSTANCE_DIR` has the value `/var/ig`.
- A ForgeRock user with username: `forgerock` and uid: `11111`, runs the IG process and owns the configuration files.

Build the base image for IG

1. Download `IG-2024.3.0.zip` from the Backstage download site, and unzip. The directory /path/to/identity-gateway-2024.3.0 is created.
2. Go to /path/to/identity-gateway-2024.3.0.
3. With a Docker daemon running, build a base Docker image:

```
$ docker build . -f docker/Dockerfile -t ig-image

Sending build context to Docker daemon
Step 1/7 : FROM gcr.io/forgerock-io/...:latest
latest: Pulling from forgerock-io/...
...
Successfully tagged ig-image:latest
```

4. Make sure the Docker image is available:

```
$ docker image list

REPOSITORY                  TAG        IMAGE ID
ig-image                    latest
gcr.io/forgerock-io/...  latest
```

Run the Docker image

The following steps run the Docker image on port `8080`. Make sure the port is not being used, or use a different port as described in the procedure.

1. With a Docker daemon running, run the Docker image:

```
$ docker run -p 8080:8080 ig-image
```

   IG starts up, and the console displays the message log.

2. Go to http://localhost:8080 to view the IG welcome page.

Stop the Docker image

1. List the Docker containers that are running:

```
$ docker container ls
```

2. For a container with the status `Up`, use the container ID to stop the container:

```
$ docker container stop CONTAINER_ID
```

Run options

Consider using the following options when you run the Docker image:

`-e IG_OPTS=-Dig.pid.file.mode=value ig-image`

Allow startup if there is an existing PID file. IG removes the existing PID file and creates a new one during startup. The following example passes an environment variable with the value `override` as a Java runtime option:

```
$ docker run -e "IG_OPTS=-Dig.pid.file.mode=override" ig-image
```

To prevent restart if there is an existing PID file, set the value to the default `fail`.

`-p port:port`

The default ports `8080:8080` equate to `local-machine-port:internal-container-port`. IG can run on a different port, but the container must always run on `8080`. The following example runs IG on port `8090`:

```
$ docker run -p 8090:8080 ig-image
```

`-v configuration directory`

The default configuration directory is `/var/ig/`. The following example sets the configuration directory to `$HOME/.openig`:

```
$ docker run -v $HOME/.openig:/var/ig/ ig-image
```

`-user user`

Run the image using the provided Forgerock user. The following example uses the ID `11111`:

```
$ docker run --user 11111 ig-image
```

`it`

Run the image in interactive mode:

```
$ docker run -it ig-image
```

`sh`

Run the image in sh shell:

```
$ docker run ig-image sh
```

Add configuration to a Docker image

The following sections describe how to add configuration to your Docker image. Before working through this section, complete the procedures in Build and run a Docker image.

Run an image with a mutable configuration

This section describes how to add a basic route to your local IG configuration folder, and mount the configuration to the Docker container.

If you change your configuration in a way that doesn't require IG to restart, you see the change in your running Docker image without restarting it or rebuilding it. For information changes that require restart, refer to When to restart IG after changing the configuration.

Use this procedure to manage configuration externally to the Docker image. For example, use it when developing routes.

1. Add the following route to your local IG configuration as `$HOME/.openig/config/routes/hello.json`:

   ```
   {
     "name": "hello",
     "handler": {
       "type": "StaticResponseHandler",
       "config": {
         "status": 200,
         "headers": {
           "Content-Type": [ "text/plain; charset=UTF-8" ]
         },
         "entity": "Hello world!"
       }
     },
     "condition": "${find(request.uri.path, '^/hello')}"
   }
   ```

   The configuration contains a static response handler to return a "Hello world!" statement when the URI of a request finishes with `/hello`.

2. Run the Docker image, using the option to mount the local IG configuration directory:

   ```
   $ docker run -p 8080:8080 -v $HOME/.openig:/var/ig/ ig-image
   ```

3. Go to http://localhost:8080/hello to access the route in the mounted configuration.

   The "Hello world!" statement is displayed.

4. Edit `hello.json` to change the "Hello world!" statement, and save the file.

Go again to http://localhost:8080/hello to see that the message changed without changing your Docker image.

Run an image with an immutable configuration

This section describes how to add a basic route to your local IG configuration folder, copy it into a new Docker image, and run that Docker image.

Unlike the previous example, the Docker image is immutable. If you change your configuration locally, the Docker image is not changed.

Use this procedure to manage configuration within the Docker image. For example, use it when you want to deploy the same configuration multiple times.

1. Add the following route to your local IG configuration as `$HOME/.openig/config/routes/hello.json`:

   ```
   {
     "name": "hello",
     "handler": {
       "type": "StaticResponseHandler",
       "config": {
         "status": 200,
         "headers": {
           "Content-Type": [ "text/plain; charset=UTF-8" ]
         },
         "entity": "Hello world!"
       }
     },
     "condition": "${find(request.uri.path, '^/hello')}"
   }
   ```

   The configuration contains a static response handler to return a "Hello world!" statement when the URI of a request finishes with `/hello`.

2. Add the following file to your local IG configuration as `$HOME/.openig/Dockerfile`, where `$HOME/.openig` is the instance directory:

   ```
   FROM ig-image
   COPY config/routes/hello.json "$IG_INSTANCE_DIR"/config/routes/hello.json
   ```

   The Dockerfile copies `hello.json` into the Docker image. The `$IG_INSTANCE_DIR` environment variable is defined in the IG base image.

3. Build the Docker image:

   ```
   $ docker build . -t ig-custom

   Sending build context to Docker daemon
    Step 1/2 : FROM ig-image
    Step 2/2 : COPY config/routes/hello.json "$IG_INSTANCE_DIR"/config/routes/hello.json
    Successfully tagged ig-custom:latest
   ```

4. Make sure the Docker image is available:

```
$ docker image list

REPOSITORY               TAG               IMAGE ID
ig-custom                image_tag         51b...3b7
gcr.io/forgerock-io/ig   image_tag         404...a2b
```

5. Run the Docker image on port `8080`:

```
$ docker run -p 8080:8080 ig-custom
```

6. Go to http://localhost:8080/hello. The "Hello world!" statement is displayed.

Gateway guide

This guide shows you how to set up examples that use IG. It is for access management designers and administrators who develop, build, deploy, and maintain IG for their organizations.

This guide assumes familiarity with the following topics:

- Hypertext Transfer Protocol (HTTP), including how clients and servers exchange messages, and the role that a reverse proxy (gateway) plays
- JavaScript Object Notation (JSON), which is the format for IG configuration files
- Managing services on operating systems and application servers
- Configuring network connections on operating systems
- Managing Public Key Infrastructure (PKI) used to establish HTTPS connections
- Access management for web applications

Depending on the features you use, you should also have basic familiarity with the following topics:

- Structured Query Language (SQL) if you use IG with relational databases
- Configuring AM if you use password capture and replay, or if you plan to follow the OAuth 2.0 or SAML 2.0 tutorials
- The Groovy programming language if you plan to extend IG with scripts
- The Java programming language if you plan to extend IG with plugins, and Apache Maven for building plugins

Example installation for this guide

Unless otherwise stated, the examples in this guide assume the following installation:

- IG accessible on `http://ig.example.com:8080` and `https://ig.example.com:8443`, as described in Quick install.
- Sample application installed on http://app.example.com:8081, as described in Use the sample application.
- AM installed on http://am.example.com:8088/openam, with the default configuration.

If you use a different configuration, substitute in the procedures accordingly.

Set up AM

This documentation contains procedures for setting up items in AM that you can use with IG. For more information about setting up AM, refer to the Access Management docs.

Authenticate an IG agent to AM

Important

From AM 7.3

When AM 7.3 is installed with a default configuration, as described in Evaluation, IG is automatically authenticated to AM by an authentication tree. Otherwise, IG is authenticated to AM by an AM authentication module.
Authentication chains and modules were deprecated in AM 7. When they are removed in a future release of AM, it will be necessary to configure an appropriate authentication tree when you are not using the default configuration.
For more information, refer to AM's Authentication Nodes and Trees.

This section describes how to create an authentication tree to authenticate an IG agent to AM. The tree has the following requirements:

- It must be called `Agent`
- Its nodes must pass the agent credentials to the Agent Data Store Decision node.

When you define a tree in AM, that same tree is used for all instances of IG, Java agent, and Web agent. Consider this point if you change the tree configuration.

1. On the **Realms** page of the AM admin UI, choose the realm in which to create the authentication tree.
2. On the **Realm Overview** page, click **Authentication** > **Trees** > **+ Create tree**.
3. Create a tree named `Agent`.

   The authentication tree designer is displayed, with the `Start` entry point connected to the `Failure` exit point, and a `Success` node.

   The authentication tree designer provides the following features on the toolbar:

Button	Usage
	Lay out and align nodes according to the order they are connected.
	Toggle the designer window between normal and full-screen layout.
	Remove the selected node. Note that the `Start` entry point cannot be deleted.

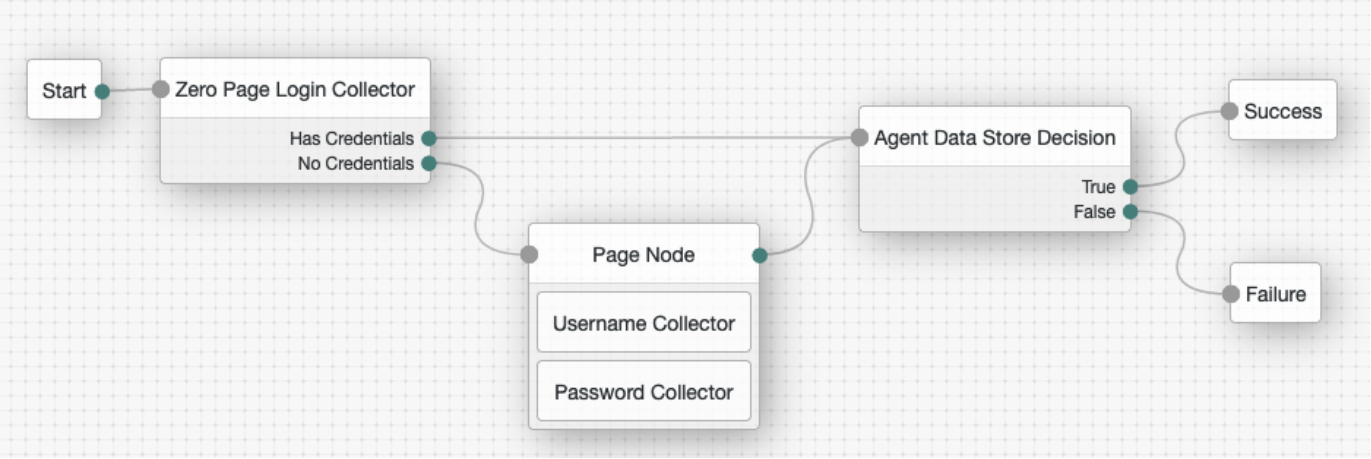
4. Using the **Filter** bar, find and then drag the following nodes from the **Components** panel into the designer area:
   - Zero Page Login Collector node to check whether the agent credentials are provided in the incoming authentication request, and use their values in the following nodes.

     This node is required for compatibility with Java agent and Web agent.

- Page node to collect the agent credentials if they are not provided in the incoming authentication request, and use their values in the following nodes.
- Agent Data Store Decision node to verify the agent credentials match the registered IG agent profile.

> **Important**
>
> Many nodes can be configured in the panel on the right side of the page. Unless otherwise stated, do not configure the nodes and use only the default values.

5. Drag the following nodes from the **Components** panel into the Page node:
   - Username Collector node to prompt the user to enter their username.
   - Password Collector node to prompt the user to enter their password.
6. Connect the nodes as follows and save the tree:



Register an IG agent in AM

In AM 7 and later versions, follow these steps to register an agent that acts on behalf of IG.

1. In the AM admin UI, select the top-level realm, and then select **Applications** > **Agents** > **Identity Gateway**.
2. Add an agent with the following configuration, leaving other options blank or with the default value:

For SSO

- **Agent ID** : `ig_agent`
- **Password** : `password`

For CDSSO

- **Agent ID** : `ig_agent`
- **Password** : `password`
- **Redirect URL for CDSSO** : `https://ig.ext.com:8443/home/cdsso/redirect`
- **Login URL Template for CDSSO**: Configure this property to direct login to a custom URL instead of the default AM login page.

3. (Optional - From AM 7.5) Use AM's secret service to manage the agent profile password. If AM finds a matching secret in a secret store, it uses that secret instead of the agent password configured in Step 2.

    1. In the agent profile page, set a label for the agent password in **Secret Label Identifier**.

        AM uses the identifier to generate a secret label for the agent.

        The secret label has the format `am.application.agents.identifier.secret`, where identifier is the **Secret Label Identifier**.

        The **Secret Label Identifier** can only contain characters `a-z`, `A-Z`, `0-9`, and periods (`.`). It can't start or end with a period.

    2. Select **Secret Stores** and configure a secret store.
    3. Map the label to the secret. Learn more from AM's mapping.

    Note the following points for using AM's secret service:

    - Set a **Secret Label Identifier** that clearly identifies the agent.
    - If you update or delete the **Secret Label Identifier**, AM updates or deletes the corresponding mapping for the previous identifier provided no other agent shares the mapping.
    - When you rotate a secret, update the corresponding mapping.

Set up a demo user in AM

AM is provided with a demo user in the top-level realm, with the following credentials:

- ID/username: `demo`
- Last name: `user`
- Password: `Ch4ng31t`
- Email address: `demo@example.com`
- Employee number: `123`

For information about how to manage identities in AM, refer to AM's Identity stores.

Find the AM session cookie name

In routes that use AmService, IG retrieves AM's SSO cookie name from the `ssoTokenHeader` property or from AM's `/serverinfo/*` endpoint.

In other circumstances where you need to find the SSO cookie name, access `http://am-base-url/serverinfo/*`. For example, access the AM endpoint with `curl`:

```
$ curl http://am.example.com:8088/openam/json/serverinfo/*
```

Set up PingOne

This documentation contains procedures for setting up items in PingOne that you can use with IG.

Create a PingOne test environment

Learn more from PingOne's Adding an environment.

In the PingOne console, create a test environment with the following values:

- **Select a solution for your Environment**: Build your own solution
- **Select solution(s) for your Environment**: `PingOne SSO`
- **ENVIRONMENT NAME**: `Test environment`
- **DESCRIPTION**: OIDC `Test environment`
- **ENVIRONMENT TYPE**: `Sandbox`

Add a PingOne test user

Learn more from PingOne's Adding a user.

In the PingOne test environment, select **Directory** > **Users** and add a user with the following values:

- **Given Name**: `demo`
- **Family Name**: `user`
- **Username**: `demo`
- **Email**: `demo@example.com`
- **Password**: `Ch4ng31t`.

You are required to change the password on first login.

External tools used in this guide

The examples in this guide use some of the following third-party tools:

- `curl`: https://curl.haxx.se
- `HTTPie`: https://httpie.org
- `jq`: https://stedolan.github.io/jq/
- `keytool`: https://docs.oracle.com/en/java/javase/11/tools/keytool.html

Authentication

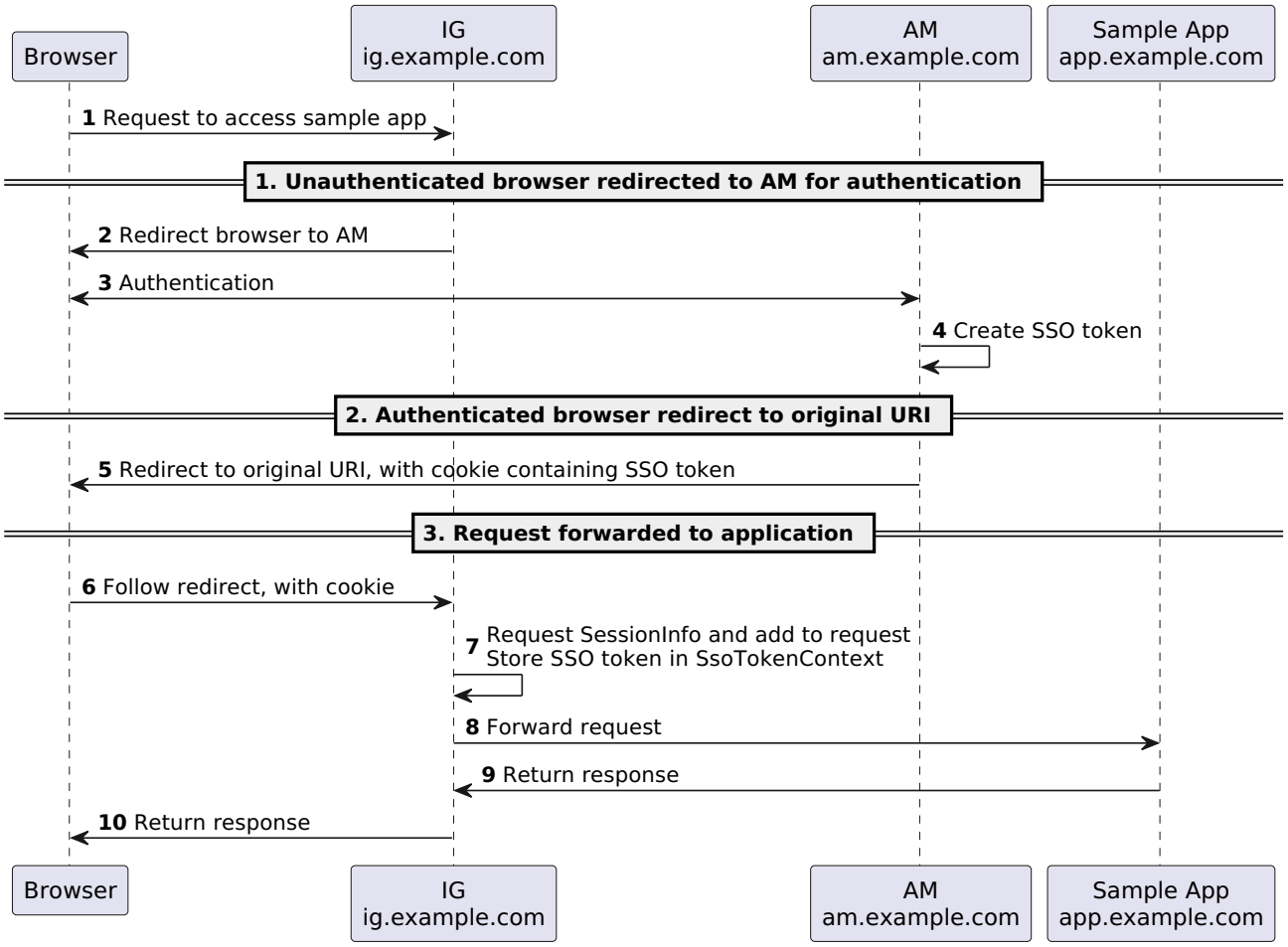
Single sign-on (SSO)

The following sections describe how to set up SSO for requests in the same domain:

> **Important**
>
> To require users to authenticate in the correct realm for security reasons, configure SSO or CDSSO with a PolicyEnforcementFilter, that refers to an AM policy where the realm is enforced. For an example, refer to Require users to authenticate to a specific realm.

In SSO using the SingleSignOnFilter, IG processes a request using authentication provided by AM. IG and the authentication provider must run on the same domain.

The following sequence diagram shows the flow of information during SSO between IG and AM as the authentication provider.



- The browser sends an unauthenticated request to access the sample app.
- IG intercepts the request, and redirects the browser to AM for authentication.
- AM authenticates the user, creates an SSO token.
- AM redirects the request back to the original URI with the token in a cookie, and the browser follows the redirect to IG.
- IG validates the token it gets from the cookie. It then adds the AM session info to the request, and stores the SSO token in the context for use by downstream filters and handlers.
- IG forwards the request to the sample app, and the sample app returns the requested resource to the browser.

SSO through the default AM authentication tree

This section gives an example of how to authenticate by using SSO and the default authentication service provided in AM.

Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

1. Set up AM:
    1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
        - `https://ig.example.com:8443/*`
        - `https://ig.example.com:8443/*?*`

2. Register an IG agent with the following values, as described in Register an IG agent in AM:
   - **Agent ID**: `ig_agent`
   - **Password**: `password`

   > **Important**
   >
   > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

3. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

   > **Important**
   >
   > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

4. Select **Configure** > **Global Services** > **Platform**, and add `example.com` as an AM cookie domain.

   By default, AM sets host-based cookies. After authentication with AM, requests can be redirected to AM instead of to the resource.

2. Set up IG:
   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
   2. Set an environment variable for the IG agent password, and then restart IG:

      ```
      $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
      ```

      The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.
   3. Add the following route to IG to serve the sample application .css and other static resources:

      Linux

      ```
      $HOME/.openig/config/routes/00-static-resources.json
      ```

      Windows

      ```
      %appdata%\OpenIG\config\routes\00-static-resources.json
      ```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

4. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/sso.json
```

Windows

```
%appdata%\OpenIG\config\routes\sso.json
```

```
{
  "name": "sso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/sso$')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the IG route in Studio, refer to Policy enforcement in Structured Editor or Protecting a web app with Freeform Designer.

3. Test the setup:

    1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/sso.
    2. Log in to AM as user `demo`, password `Ch4ng31t`.

       The SingleSignOnFilter passes the request to sample application, which returns the sample application home page.

SSO through a specified AM authentication tree

This section gives an example of how to authenticate by using SSO and the example authentication tree provided in AM, instead of the default authentication tree.

1. Set up the example in Authenticate with SSO through the default authentication service.

2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/sso-authservice.json
```

Windows

```
%appdata%\OpenIG\config\routes\sso-authservice.json
```

```
{
  "name": "sso-authservice",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/sso-authservice')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1",
            "authenticationService": "Example"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the features of the route compared to `sso.json`:

- The route matches requests to `/home/sso-authservice`.
- The `authenticationService` property of SingleSignOnFilter refers to `Example`, the name of the example authentication tree in AM. This authentication tree is used for authentication instead of the AM admin UI.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/sso-authservice.
   2. If you see warnings that the site isn't secure, respond to the warnings to access the site.
   3. Note that the login page is different to that returned in Authenticate with SSO through the default authentication service.

Cross-domain single sign-on (CDSSO)

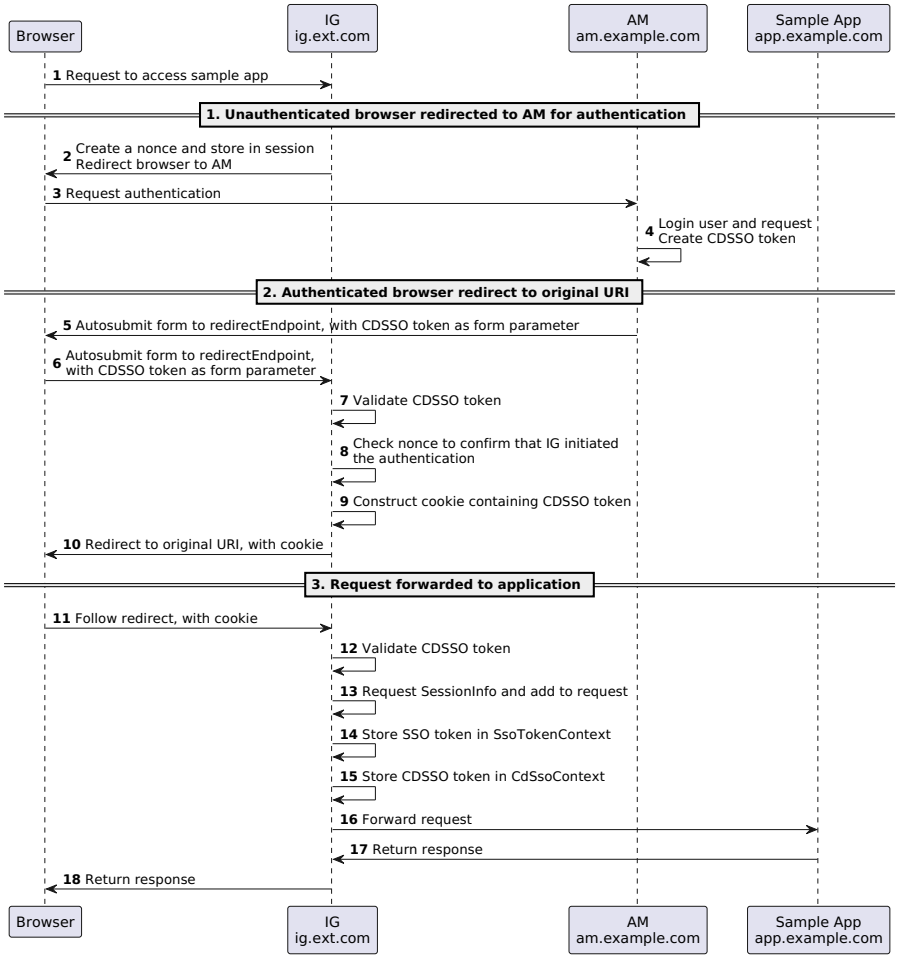
The following sections describe how to set up CDSSO for requests in a different domain:

> **Important**
>
> To require users to authenticate in the correct realm for security reasons, configure SSO or CDSSO with a PolicyEnforcementFilter, that refers to an AM policy where the realm is enforced. For an example, refer to Require users to authenticate to a specific realm.

The SSO mechanism described in Authenticating with SSO can be used when IG and AM are running in the same domain. When IG and AM are running in different domains, AM cookies are not visible to IG because of the same-origin policy.

CDSSO using the CrossDomainSingleSignOnFilter provides a mechanism to push tokens issued by AM to IG running in a different domain.

The following sequence diagram shows the flow of information between IG, AM, and the sample application during CDSSO. In this example, AM is running on `am.example.com`, and IG is running on `ig.ext.com`.



1. The browser sends an unauthenticated request to access the sample app.

2-3. IG intercepts the request, and redirects the browser to AM for authentication.

4. AM authenticates the user and creates a CDSSO token.

5. AM responds to a successful authentication with an HTML autosubmit form containing the issued token.

6. The browser loads the HTML and autosubmit form parameters to the IG callback URL for the redirect endpoint.

7. When `verificationSecretId` in CrossDomainSingleSignOnFilter is configured, IG uses it to verify signature of AM session tokens.

When `verificationSecretId` isn't configured, IG discovers and uses the AM JWK set to verify the signature of AM session tokens.

If that fails, the CrossDomainSingleSignOnFilter fails to load.

8. IG checks the nonce found inside the CDSSO token to confirm that the callback comes from an authentication initiated by IG.

9. IG constructs a cookie, and fulfills it with a cookie name, path, and domain, using the CrossDomainSingleSignOnFilter property `authCookie`. The domain must match that set in the AM IG agent.

10-11. IG redirects the request back to the original URI, with the cookie, and the browser follows the redirect back to IG.

12. IG validates the SSO token inside of the CDSSO token

13-15. IG adds the AM session info to the request, and stores the SSO token and CDSSO token in the contexts for use by downstream filters and handlers.

16-18. IG forwards the request to the sample application, and the sample application returns the requested resource to the browser.

Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

1. Set up AM:
    1. Register an IG agent with the following values, as described in Register an IG agent in AM:
        - **Agent ID**: `ig_agent_cdsso`
        - **Password**: `password`
        - **Redirect URL for CDSSO**: `https://ig.ext.com:8443/home/cdsso/redirect`

        > **Important**
        >
        > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

    2. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

        > **Important**
        >
        > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

    3. Select **Services** > **Add a Service**, and add a **Validation Service** with the following **Valid goto URL Resources**:
        - `https://ig.ext.com:8443/*`
        - `https://ig.ext.com:8443/*?*`

4. Select **Configure** > **Global Services** > **Platform**, and add `example.com` as an AM cookie domain.

   By default, AM sets host-based cookies. After authentication with AM, requests can be redirected to AM instead of to the resource.

2. Set up IG:

   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

   2. Add the following `session` configuration to `admin.json`, to ensure that the browser passes the session cookie in the form-POST to the redirect endpoint (step 6 of Information flow during CDSSO):

      ```
      {
        "connectors": […],
        "session": {
          "cookie": {
            "sameSite": "none",
            "secure": true
          }
        },
        "heap": […]
      }
      ```

      This step is required for the following reasons:

      - When `sameSite` is `strict` or `lax`, the browser does not send the session cookie, which contains the nonce used in validation. If IG doesn't find the nonce, it assumes that the authentication failed.
      - When `secure` is `false`, the browser is likely to reject the session cookie.

        For more information, refer to admin.json.

   3. Set an environment variable for the IG agent password, and then restart IG:

      ```
      $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
      ```

      The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   4. Add the following route to IG to serve the sample application .css and other static resources:

      Linux

      ```
      $HOME/.openig/config/routes/00-static-resources.json
      ```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

5. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/cdsso.json
```

Windows

```
%appdata%\OpenIG\config\routes\cdsso.json
```

```
{
  "name": "cdsso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/cdsso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "agent": {
          "username": "ig_agent_cdsso",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "sessionCache": {
          "enabled": false
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "CrossDomainSingleSignOnFilter-1",
          "type": "CrossDomainSingleSignOnFilter",
          "config": {
            "redirectEndpoint": "/home/cdsso/redirect",
            "authCookie": {
              "path": "/home",
              "name": "ig-token-cookie"
            },
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/home/cdsso`.
- The agent password for AmService is provided by a SystemAndEnvSecretStore in the heap.
- Because the CrossDomainSingleSignOnFilter's `verificationSecretId` isn't configured, IG discovers and uses the AM JWK set to verify the signature of AM session tokens. If that fails, the CrossDomainSingleSignOnFilter fails to load.

3. Test the setup:

    1. In your browser's privacy or incognito mode, go to https://ig.ext.com:8443/home/cdsso.

       The CrossDomainSingleSignOnFilter redirects the request to AM for authentication.

    2. Log in to AM as user `demo`, password `Ch4ng31t`.

       When you have authenticated, AM calls `/home/cdsso/redirect`, and includes the CDSSO token. The CrossDomainSingleSignOnFilter passes the request to sample app, which returns the home page.

Password replay from AM

Use IG with AM's password capture and replay to bring SSO to legacy web applications, without the need to edit, upgrade, or recode. This feature helps you to integrate legacy web applications with other applications using the same user identity.

The following figure illustrates the flow of requests when an unauthenticated user accesses a protected application. After authenticating with AM, the user is logged into the application with the username and password from the AM login session.

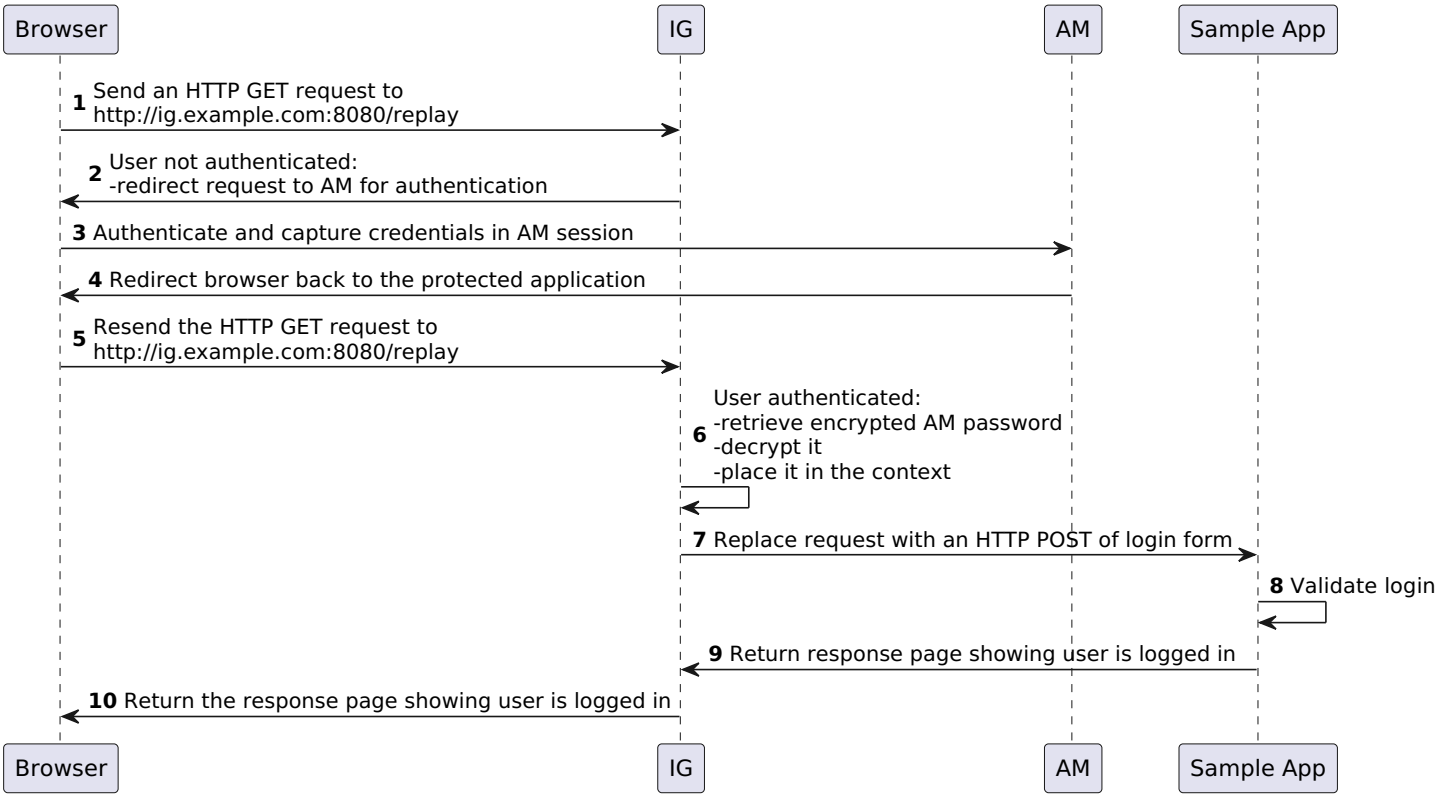


Figure 1. Data flow to log in to a protect appl

- IG intercepts the browser's HTTP GET request.
- Because the user is not authenticated, the SingleSignOnFilter redirects the user to AM for authentication.
- AM authenticates the user, capturing the login credentials, and storing the encrypted password in the user's AM session.
- AM redirects the browser back to the protected application.

- IG intercepts the browser's HTTP GET request again:
  - The user is now authenticated, so IG's SingleSignOnFilter passes the request to the CapturedUserPasswordFilter.
  - The CapturedUserPasswordFilter checks that the SessionInfoContext `${contexts.amSession.properties.sunIdentityUserPassword}` is available and not `null`. It then decrypts the password and stores it in the CapturedUserPasswordContext, at `${contexts.capturedPassword}`.
- The PasswordReplayFilter uses the username and decrypted password in the context to replace the request with an HTTP POST of the login form.
- The sample application validates the credentials.
- The sample application responds with the user's profile page.
- IG then passes the response from the sample application to the browser.

Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

> **Note**
>
> In Identity Cloud and from AM 7.5, the password capture and replay feature can optionally manage the replay password through AM's secret service. The secret label for the replay password must be `am.authentication.replaypassword.key`.
> For backward compatibility, if a secret isn't defined, is empty, or can't be resolved, AM manages the replay password through the AM system property `am.authentication.replaypassword.key`.

1. Generate an AES 256-bit key:

```
$ openssl rand -base64 32
loH...UFQ=
```

2. Set up AM:
   1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
      - `https://ig.example.com:8443/*`
      - `https://ig.example.com:8443/*?*`
   2. Register an IG agent with the following values, as described in Register an IG agent in AM:
      - **Agent ID**: `ig_agent`
      - **Password**: `password`

      > **Important**
      >
      > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   3. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

> **Important**
>
> IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

4. Update the Authentication Post Processing Classes for password replay:

    1. Select **Authentication** > **Settings** > **Post Authentication Processing**.
    2. In **Authentication Post Processing Classes**, add `com.sun.identity.authentication.spi.JwtReplayPassword`.

5. Add the AES 256-bit key to AM:

    1. Select **DEPLOYMENT** > **Servers**, and then select the AM server name, `http://am.example.com:8088/openam`.

        In earlier version of AM, select **Configuration** > **Servers and Sites**.

    2. Select **Advanced**, and add the following property:

        - **PROPERTY NAME** : `com.sun.am.replaypasswd.key`
        - **PROPERTY VALUE** : The value of the AES 256-bit key from step 1.

6. Select **Configure** > **Global Services** > **Platform**, and add `example.com` as an AM cookie domain.

    By default, AM sets host-based cookies. After authentication with AM, requests can be redirected to AM instead of to the resource.

3. Set up IG:

    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
    2. Set environment variables for the value of the AES 256-bit key in step 1, and the IG agent password, and then restart IG:

```
$ export AES_KEY='AES 256-bit key'
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

    3. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

4. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/04-replay.json
```

Windows

```
%appdata%\OpenIG\config\routes\04-replay.json
```

```
{
  "name": "04-replay",
  "condition": "${find(request.uri.path, '^/replay')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore",
      "config": {
        "mappings": [
          {
            "secretId": "aes.key",
            "format": {
              "type": "SecretKeyPropertyFormat",
              "config": {
                "format": "BASE64",
                "algorithm": "AES"
              }
            }
          }
        ]
      }
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    },
    {
      "name": "CapturedUserPasswordFilter",
      "type": "CapturedUserPasswordFilter",
      "config": {
        "ssoToken": "${contexts.ssoToken.value}",
        "keySecretId": "aes.key",
        "keyType": "AES",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "amService": "AmService-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${true}",
```

```
            "credentials": "CapturedUserPasswordFilter",
            "request": {
              "method": "POST",
              "uri": "http://app.example.com:8081/login",
              "form": {
                "username": [
                  "${contexts.ssoToken.info.uid}"
                ],
                "password": [
                  "${contexts.capturedPassword.value}"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/replay`.
- The agent password for AmService is provided by a SystemAndEnvSecretStore in the heap.
- If the request does not have a valid AM session cookie, the SingleSignOnFilter redirects the request to AM for authentication.

  After authentication, the SingleSignOnFilter passes the request to the next filter, storing the cookie value in an `SsoTokenContext`.
- The PasswordReplayFilter uses the CapturedUserPasswordFilter declared in the heap to retrieve the AM password from AM session properties. The CapturedUserPasswordFilter uses the AES 256-bit key to decrypt the password, and then makes it available in a CapturedUserPasswordContext.

  The value of the AES 256-bit key is provided by the SystemAndEnvSecretStore.

  The PasswordReplayFilter retrieves the username and password from the context. It replaces the browser's original HTTP GET request with an HTTP POST login request containing the credentials to authenticate to the sample application.

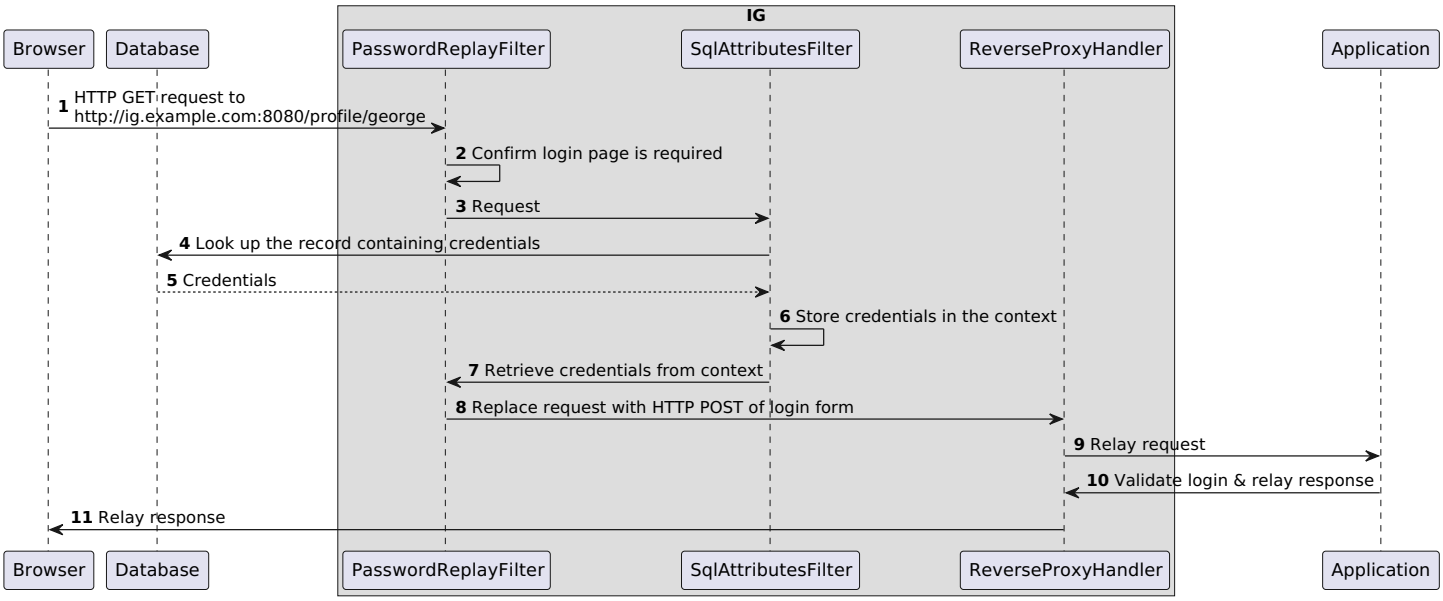
4. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/replay. The SingleSignOnFilter redirects the request to AM for authentication.
   2. If you see warnings that the site isn't secure, respond to the warnings to access the site.
   3. Log in to AM as user `demo`, password `Ch4ng31t`. The request is redirected to the sample application.

Password replay from a database

This section describes how to configure IG to get credentials from a database. This example is tested with H2 1.4.197.

The following figure illustrates the flow of requests when IG uses credentials from a database to log a user in to the sample application:



- IG intercepts the browser's HTTP GET request.
- The PasswordReplayFilter confirms that a login page is required, and passes the request to the SqlAttributesFilter.
- The SqlAttributesFilter uses the email address to look up credentials in H2, and stores them in the request context attributes map.
- The PasswordReplayFilter retrieves the credentials from the attributes map, builds the login form, and performs the HTTP POST request to the sample app.
- The sample application validates the credentials, and responds with a profile page.

Before you start, prepare IG and the sample application as described in the Quick install.

1. Set up the database:
    1. On your system, add the following data in a comma-separated value file:

Linux

```
/tmp/userfile.txt
```

Windows

```
C:\Temp\userfile.txt
```

```
username,password,fullname,email
george,C0stanza,George Costanza,george@example.com
kramer,N3wman12,Kramer,kramer@example.com
bjensen,H1falutin,Babs Jensen,bjensen@example.com
demo,Ch4ng31t,Demo User,demo@example.com
kvaughan,B5ibery12,Kirsten Vaughan,kvaughan@example.com
scarter,S9rain12,Sam Carter,scarter@example.com
```

2. Download and unpack the H2 database, and then start H2:

```
$ sh /path/to/h2/bin/h2.sh
```

H2 starts, listening on port 8082, and opens the H2 Console in a browser.

3. In the H2 Console, select the following options, and then select Connect to access the console:

- **Saved Settings** : `Generic H2 (Server)`
- **Setting Name** : `Generic H2 (Server)`
- **Driver Class**: `org.h2.Driver`
- **JDBC URL**: `jdbc:h2:~/ig-credentials`
- **User Name**: `sa`
- **Password** : `password`

> **Tip**
>
> If you have run this example before but can't access the console now, try deleting your local `~/ig-credentials` files and starting H2 again.

4. In the console, add the following text, and then run it to create the user table:

```
DROP TABLE IF EXISTS USERS;
CREATE TABLE USERS AS SELECT * FROM CSVREAD('/tmp/userfile.txt');
```

5. In the console, add the following text, and then run it to verify that the table contains the same users as the file:

```
SELECT * FROM users;
```

6. Add the .jar file `/path/to/h2/bin/h2-*.jar` to the IG configuration:

- Create the directory `$HOME/.openig/extra`, where `$HOME/.openig` is the instance directory, and add .jar files to the directory.

2. Set up IG:

    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

    2. Set an environment variable for the database password, and then restart IG:

    ```
    $ export DATABASE_PASSWORD='cGFzc3dvcmQ='
    ```

    The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

    3. Add the following route to IG to serve the sample application .css and other static resources:

    Linux

    ```
    $HOME/.openig/config/routes/00-static-resources.json
    ```

    Windows

    ```
    %appdata%\OpenIG\config\routes\00-static-resources.json
    ```

    ```
    {
      "name" : "00-static-resources",
      "baseURI" : "http://app.example.com:8081",
      "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
    or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
      "handler": "ReverseProxyHandler"
    }
    ```

    4. Add the following route to IG:

    Linux

    ```
    $HOME/.openig/config/routes/03-sql.json
    ```

Windows

```
%appdata%\OpenIG\config\routes\03-sql.json
```

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "JdbcDataSource-1",
      "type": "JdbcDataSource",
      "config": {
        "driverClassName": "org.h2.Driver",
        "jdbcUrl": "jdbc:h2:tcp://localhost/~/ig-credentials",
        "username": "sa",
        "passwordSecretId": "database.password",
        "secretsProvider": "SystemAndEnvSecretStore-1"
      }
    }
  ],
  "name": "sql",
  "condition": "${find(request.uri.path, '^/profile')}",
  "handler": {
    "type": "Chain",
    "baseURI": "http://app.example.com:8081",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${find(request.uri.path, '^/profile/george') and (request.method == 'GET')}",
            "credentials": {
              "type": "SqlAttributesFilter",
              "config": {
                "dataSource": "JdbcDataSource-1",
                "preparedStatement":
                "SELECT username, password FROM users WHERE email = ?;",
                "parameters": [
                  "george@example.com"
                ],
                "target": "${attributes.sql}"
              }
            },
            "request": {
              "method": "POST",
              "uri": "http://app.example.com:8081/login",
              "form": {
                "username": [
                  "${attributes.sql.USERNAME}"
                ],
                "password": [
                  "${attributes.sql.PASSWORD}"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/profile`.
- The PasswordReplayFilter specifies a loginPage page property:
- When a request is an HTTP GET, and the request URI path is `/profile/george`, the expression resolves to `true`. The request is directed to a login page.

  The `SqlAttributesFilter` specifies the data source to access, a prepared statement to look up the user's record, a parameter to pass into the statement, and where to store the search results in the request context attributes map.

  The `request` object retrieves the username and password from the context, and replaces the browser's original HTTP GET request with an HTTP POST login request, containing the credentials to authenticate.

  The request is for `username, password`, but H2 returns the fields as `USERNAME` and `PASSWORD`. The configuration reflects this difference.
- For other requests, the expression resolves to `false`. The request passes to the ReverseProxyHandler, which directs it to the profile page of the sample app.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/profile.

      If you see warnings that the site isn't secure, respond to the warnings to access the site.

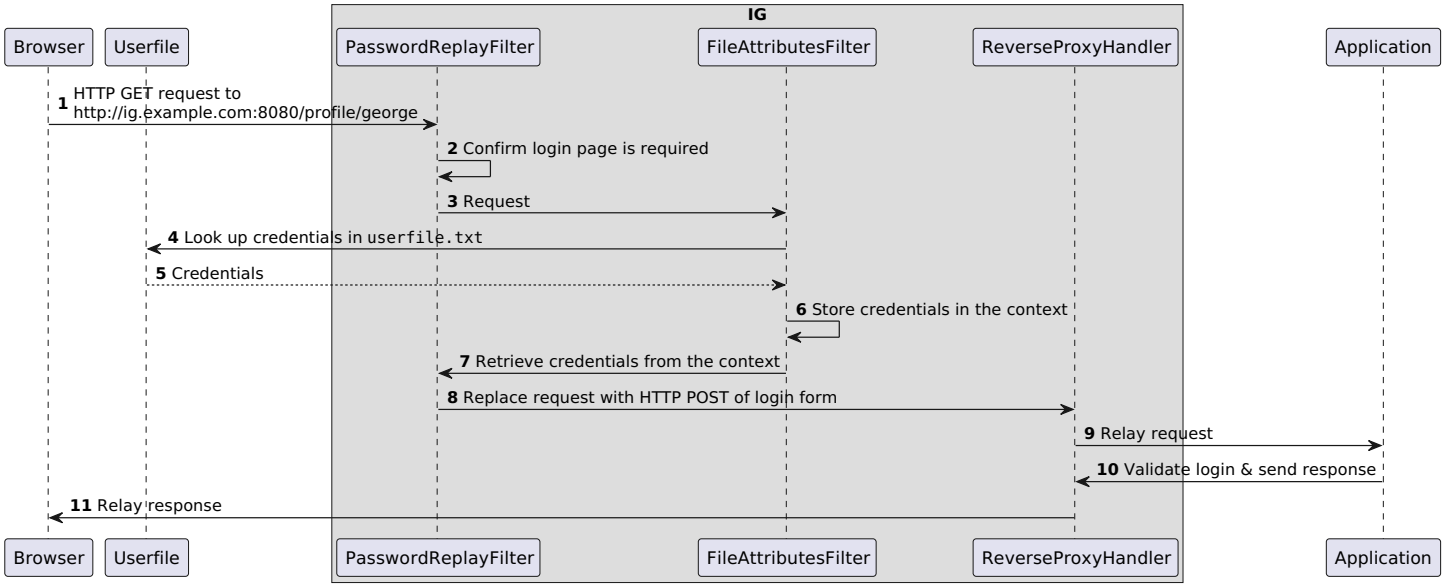
      Because the property `loginPage` resolves to `false`, the PasswordReplayFilter passes the request directly to the ReverseProxyHandler. The sample app returns the login page.

   2. Go to https://ig.example.com:8443/profile/george.

      Because the property `loginPage` resolves to `true`, the PasswordReplayFilter processes the request to obtain the login credentials. The sample app returns the profile page for George.

Password replay from a file

The following figure illustrates the flow of requests when IG uses credentials in a file to log a user in to the sample application:



- IG intercepts the browser's HTTP GET request, which matches the route condition.
- The PasswordReplayFilter confirms that a login page is required, and
- The FileAttributesFilter uses the email address to look up the user credentials in a file, and stores the credentials in the request context attributes map.
- The PasswordReplayFilter retrieves the credentials from the attributes map, builds the login form, and performs the HTTP POST request to the sample app.
- The sample application validates the credentials, and responds with a profile page.
- The ReverseProxyHandler passes the response to the browser.

Before you start, prepare IG and the sample application as described in the Quick install.

1. On your system, add the following data in a comma-separated value file:

Linux

```
/tmp/userfile.txt
```

Windows

```
C:\Temp\userfile.txt
```

```
username,password,fullname,email
george,C0stanza,George Costanza,george@example.com
kramer,N3wman12,Kramer,kramer@example.com
bjensen,H1falutin,Babs Jensen,bjensen@example.com
demo,Ch4ng31t,Demo User,demo@example.com
kvaughan,B5ibery12,Kirsten Vaughan,kvaughan@example.com
scarter,S9rain12,Sam Carter,scarter@example.com
```

2. Set up IG:

    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

    2. Add the following route to IG to serve the sample application .css and other static resources:

    Linux

    ```
    $HOME/.openig/config/routes/00-static-resources.json
    ```

    Windows

    ```
    %appdata%\OpenIG\config\routes\00-static-resources.json
    ```

    ```
    {
      "name" : "00-static-resources",
      "baseURI" : "http://app.example.com:8081",
      "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
    or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
      "handler": "ReverseProxyHandler"
    }
    ```

    3. Add the following route to IG:

    Linux

    ```
    $HOME/.openig/config/routes/02-file.json
    ```

Windows

```
%appdata%\OpenIG\config\routes\02-file.json
```

```
{
  "name": "02-file",
  "condition": "${find(request.uri.path, '^/profile')}",
  "capture": "all",
  "handler": {
    "type": "Chain",
    "baseURI": "http://app.example.com:8081",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${find(request.uri.path, '^/profile/george') and (request.method == 'GET')}",
            "credentials": {
              "type": "FileAttributesFilter",
              "config": {
                "file": "/tmp/userfile.txt",
                "key": "email",
                "value": "george@example.com",
                "target": "${attributes.credentials}"
              }
            },
            "request": {
              "method": "POST",
              "uri": "http://app.example.com:8081/login",
              "form": {
                "username": [
                  "${attributes.credentials.username}"
                ],
                "password": [
                  "${attributes.credentials.password}"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/profile`.
- The `PasswordReplayFilter` specifies a `loginPage` page property:
  - When a request is an HTTP GET, and the request URI path is `/profile/george`, the expression resolves to `true`. The request is directed to a login page.

The `FileAttributesFilter` looks up the key and value in `/tmp/userfile.txt`, and stores them in the context.

The `request` object retrieves the username and password from the context, and replaces the browser's original HTTP GET request with an HTTP POST login request, containing the credentials to authenticate.

- For other requests, the expression resolves to `false`. The request passes to the ReverseProxyHandler, which directs it to the profile page of the sample app.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/profile/george.
   2. If you see warnings that the site isn't secure, respond to the warnings to access the site.

      Because the property `loginPage` resolves to `true`, the PasswordReplayFilter processes the request to obtain the login credentials. The sample app returns the profile page for George.

   3. Go to https://ig.example.com:8443/profile/bob, or to any other URI starting with `https://ig.example.com:8443/profile`.

      Because the property `loginPage` resolves to `false`, the PasswordReplayFilter passes the request directly to the ReverseProxyHandler. The sample app returns the login page.

Session cache eviction

When WebSocket notifications are enabled in IG, IG receives notifications when the following events occur:

- A user logs out of AM
- An AM session is modified, closed, or times out
- An AM admin forces logout of user sessions (from AM 7.3)

The following procedure gives an example of how to change the configurations in Single sign-on and Cross-domain single sign-on to receive WebSocket notifications for session logout, and to evict entries related to the session from the cache. For information about WebSocket notifications, refer to WebSocket notifications.

Before you start, set up and test the example in Single sign-on (SSO).

1. Websocket notifications are enabled by default. If they are disabled, enable them by adding the following configuration to the AmService in your route:

```
"notifications": {
  "enabled": true
}
```

2. Enable the session cache by adding the following configuration to the AmService in your route:

```
"sessionCache": {
  "enabled": true
}
```

3. In `logback.xml` add the following logger for WebSocket notifications, and then restart IG:

   ```
   <logger name="org.forgerock.openig.tools.notifications.ws" level="TRACE" />
   ```

   For information, refer to Changing the log level for different object types.

4. On the AM console, log the demo user out of AM to end the AM session.

5. Note that the IG system logs are updated with Websocket notifications about the logout:

   ```
   ... | TRACE | vert.x-eventloop-thread-4 | o.f.o.t.n.w.l.DirectAmLink | @system | Received a message:
   { "topic": ... "eventType": "LOGOUT" } }
   ... | TRACE | vert.x-eventloop-thread-4 | o.f.o.t.n.w.SubscriptionService | @system | Notification received...
   "eventType": "LOGOUT" }}
   ... | TRACE | vert.x-eventloop-thread-4 | o.f.o.t.n.w.SubscriptionService | @system | Notification sent to a
   [/agent/session.v2] listener
   ```

Policy enforcement

About policy enforcement

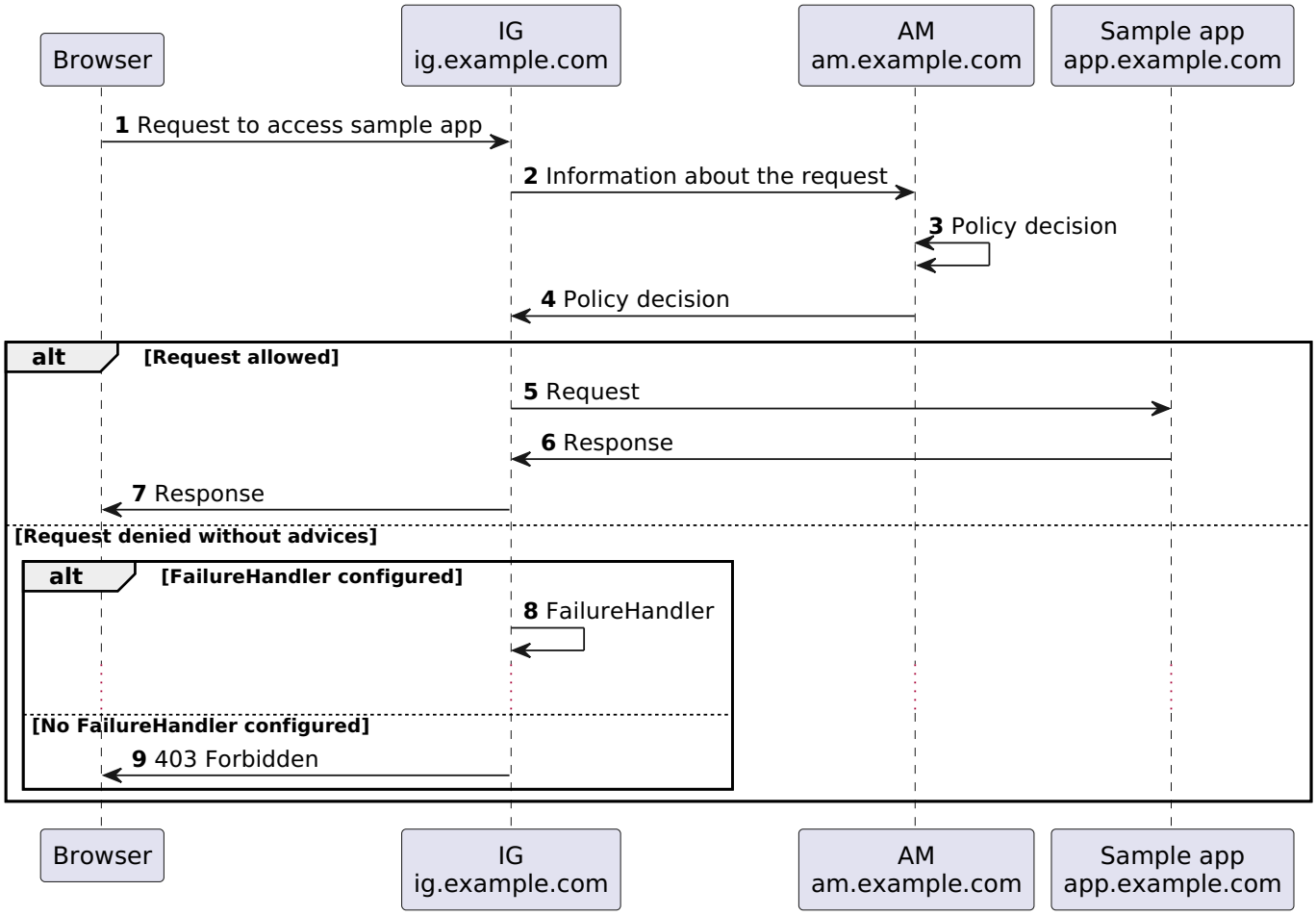
IG as a policy enforcement point (PEP) uses the PolicyEnforcementFilter to intercept requests for a resource and provide information about the request to AM.

AM as a policy decision point (PDP) evaluates requests based on their context and the configured policies. AM then returns decisions that indicate what actions are allowed or denied, as well as any advices, subject attributes, or static attributes for the specified resources.

For more information, refer to the PolicyEnforcementFilter and AM's Authentication and SSO guide.

Deny requests without advices

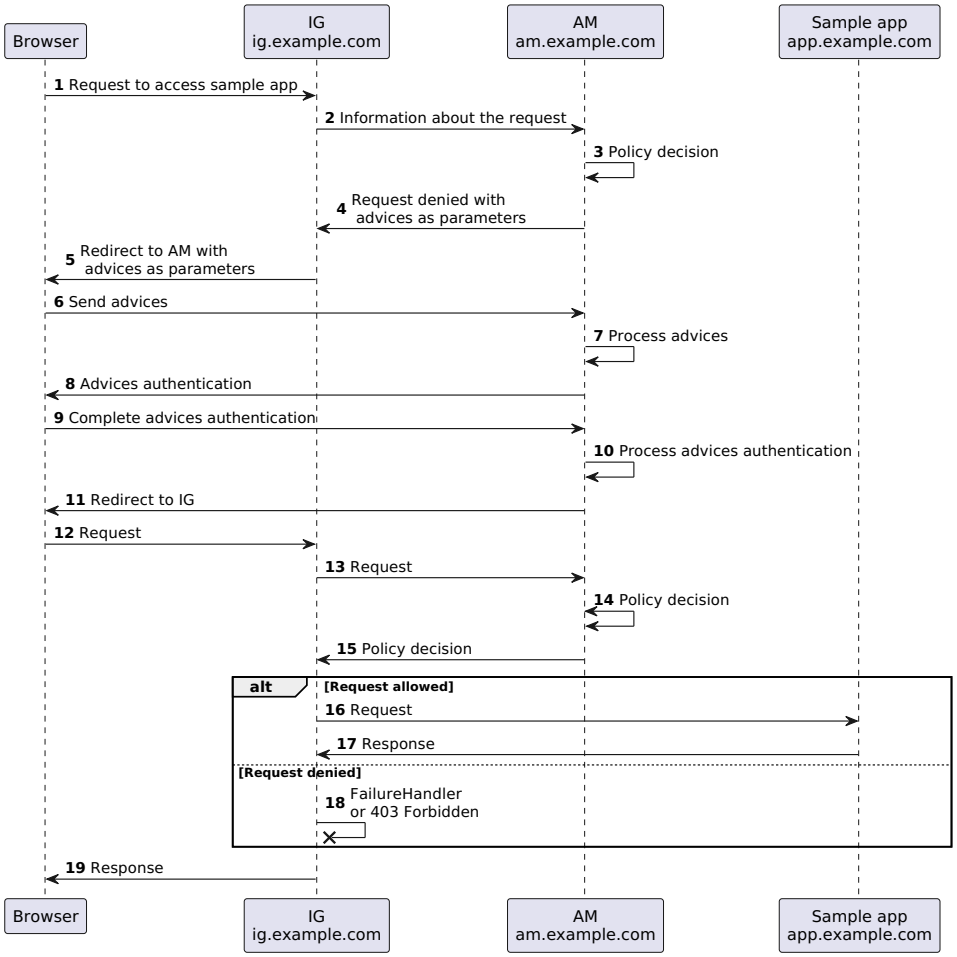
The following image shows a simplified flow of information when AM denies a request without advices.



Deny requests with advices as parameters in a redirect response

The following image shows a simplified flow of information when AM denies a request with advices and IG returns the advices as parameters in a redirect response.

This is the default flow, most used for web applications.



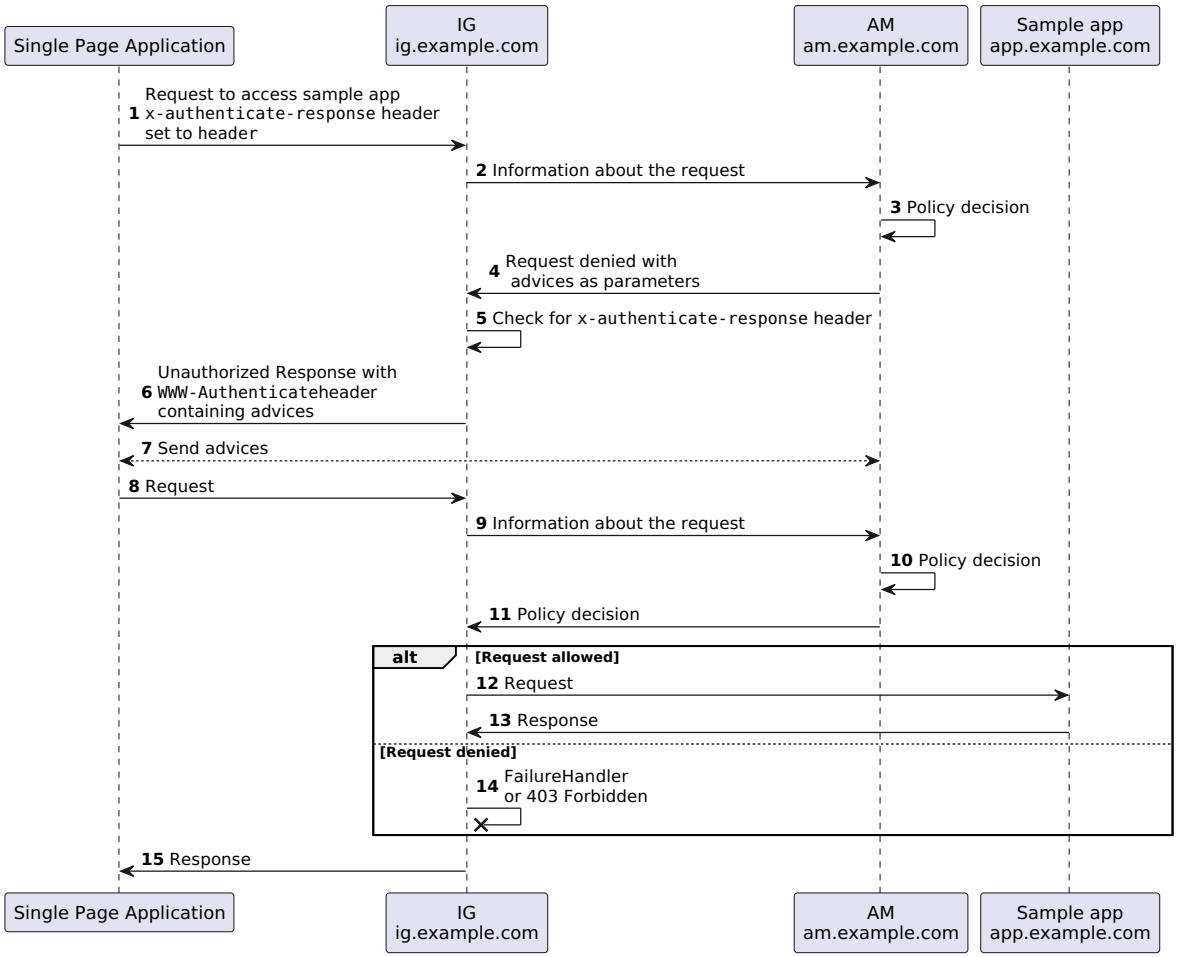
Deny requests with advices in a header

The following image shows a simplified flow of information when the request to IG includes an `x-authenticate-response` header with the value `header`. If the header has any other value, the flow in Deny requests with advices as parameters in a redirect response takes place.

To change the name of the `x-authenticate-response` header, refer to the `authenticateResponseRequestHeader` property of the PolicyEnforcementFilter.

In this flow, AM denies the request with advices, and IG sends the response with the advices in the `WWW-authenticate` header.

Use this method for SDKs and single page applications. Placing advices in a header gives these applications more options for handling the advices.



Consider the following example GET with an `x-authenticate-response` header with the value `HEADER`:

```
[CONTINUED]GET https://ig.example.com:8443/home HTTP/1.1
[CONTINUED]accept-encoding: gzip, deflate
[CONTINUED]Connection: close
[CONTINUED]cookie: iPlanetDirectoryPro=0Dx...e3A.*....; amlbcookie=01
[CONTINUED]Host: ig.example.com:8443
[CONTINUED]x-authenticate-response: HEADER
```

IG returns a `WWW-Authenticate` header containing advices, as follows:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: SSOADVICE realm="/",advices="eyJ...XX0=",am_uri="http://openam.example.com:8080/am/"
transfer-encoding: chunked
connection: close
```

The advice decodes to a transaction condition advice:

```
{"TransactionConditionAdvice":["493...3c4"]}
```

Enforce policy decisions from AM

The following sections describe how to set up single sign on for requests in the same domain and in a different domain.

Enforce AM policy decisions in the same domain

The following procedure gives an example of how to create a policy in AM and configure an agent that can request policy decisions, when IG and AM are in the same domain.

Before you start, set up and test the example in Authenticate with SSO through the default authentication service.

1. Set up AM:
    1. Select 🔑 **Authorization** > **Policy Sets** > **New Policy Set**, and add a policy set with the following values:
        - **Id** : `PEP-SSO`
        - **Resource Types** : `URL`
    2. In the policy set, add a policy with the following values:
        - **Name** : `PEP-SSO`
        - **Resource Type** : `URL`
        - **Resource pattern** : `*://*:*/*`
        - **Resource value** : `http://app.example.com:8081/home/pep-sso*`

          This policy protects the home page of the sample application.
    3. On the **Actions** tab, add an action to allow HTTP `GET`.
    4. On the **Subjects** tab, remove any default subject conditions, add a subject condition for all `Authenticated Users`.
2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/04-pep.json
```

Windows

```
%appdata%\OpenIG\config\routes\04-pep.json
```

```
{
  "name": "pep-sso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/pep-sso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "PolicyEnforcementFilter-1",
          "type": "PolicyEnforcementFilter",
          "config": {
            "application": "PEP-SSO",
            "ssoTokenSubject": "${contexts.ssoToken.value}",
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the IG route in Studio, refer to Policy enforcement in Structured Editor or Protecting a web app with Freeform Designer.

For an example route that uses `claimsSubject` instead of `ssoTokenSubject` to identify the subject, refer to Example policy enforcement using claimsSubject.

3. Test the setup:

    1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/pep-sso.

2. If you see warnings that the site isn't secure, respond to the warnings to access the site.

   Because you haven't previously authenticated to AM, the request does not contain a cookie with an SSO token. The SingleSignOnFilter redirects you to AM for authentication.

3. Log in to AM as user `demo`, password `Ch4ng31t`.

   When you have authenticated, AM redirects you back to the request URL, and IG requests a policy decision using the AM session cookie.

   AM returns a policy decision that grants access to the sample application.

Require users to authenticate to a specific realm

This example creates a policy that requires users to authenticate in a specific realm.

To reduce the attack surface on the top level realm, ForgeRock advises you to create federation entities, agent profiles, authorizations, OAuth2/OIDC, and STS services in a subrealm. For this reason, the AM policy, AM agent, and services are in a subrealm.

1. Set up AM:

   1. In the AM admin UI, click **Realms**, and add a realm named `alpha`. Leave all other values as default.

      For the rest of the steps in this procedure, make sure you are managing the alpha realm by checking that the ☁ **alpha** icon is displayed on the top left.

   2. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:

      - `https://ig.example.com:8443/*`
      - `https://ig.example.com:8443/*?*`

   3. Register an IG agent with the following values, as described in Register an IG agent in AM:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

      > **Important**
      >
      > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   4. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

      > **Important**
      >
      > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

5. Add a policy:

    1. Select 🔑 **Authorization** > **Policy Sets** > **New Policy Set**, and add a policy set with the following values:

        - **Id** : `PEP-SSO-REALM`
        - **Resource Types** : `URL`

    2. In the policy set, add a policy with the following values:

        - **Name** : `PEP-SSO-REALM`
        - **Resource Type** : `URL`
        - **Resource pattern** : `*://*:*/*`
        - **Resource value** : `http://app.example.com:8081/home/pep-sso-realm`

          This policy protects the home page of the sample application.

    3. On the **Actions** tab, add an action to allow HTTP `GET`.

    4. On the **Subjects** tab, remove any default subject conditions, add a subject condition for all `Authenticated Users`.

    5. On the **Environments** tab, add an environment condition that requires the user to authenticate to the ☁ **alpha** realm:

        - **Type** : `Authentication to a Realm`
        - **Authenticate to a Realm** : `/alpha`

2. Set up IG:

    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

    2. Set an environment variable for the IG agent password, and then restart IG:

    ```
    $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
    ```

    The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

    3. Add the following route to IG to serve the sample application .css and other static resources:

    Linux

    ```
    $HOME/.openig/config/routes/00-static-resources.json
    ```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

4. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/04-pep-sso-realm.json
```

Windows

```
%appdata%\OpenIG\config\routes\04-pep-sso-realm.json
```

```
{
  "name": "pep-sso-realm",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/pep-sso-realm')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/",
        "realm": "/alpha"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "PolicyEnforcementFilter-1",
          "type": "PolicyEnforcementFilter",
          "config": {
            "application": "PEP-SSO-REALM",
            "ssoTokenSubject": "${contexts.ssoToken.value}",
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following differences compared to `04-pep-sso.json`:

- The AmService is in the `alpha` realm. That means that the user authenticates to AM in that realm.
- The PolicyEnforcementFilter realm is not specified, so it takes the same value as the AmService realm. If refers to a policy in the AM `alpha` realm.

3. Test the setup:

    1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/pep-sso-realm.
    2. If you see warnings that the site isn't secure, respond to the warnings to access the site.
    3. Log in to AM as user `demo`, password `Ch4ng31t`.

       Because you are authenticating in the `alpha` realm, AM returns a policy decision that grants access to the sample application.

       If you were to send the request from a different realm, AM would redirect the request with an `AuthenticateToRealmConditionAdvice`.

Enforce AM policy decisions in different domains

The following procedure gives an example of how to create a policy in AM and configure an agent that can request policy decisions, when IG and AM are in different domains.

Before you start, set up and test the example in Cross-domain single sign-on.

1. Set up AM:

    1. In the AM admin UI, select **Applications** > **Agents** > **Identity Gateway**, and change the redirect URL for `ig_agent_cdsso`:

        - **Redirect URL for CDSSO** : `https://ig.ext.com:8443/home/pep-cdsso/redirect`

    2. Select 🔑 **Authorization** > **Policy Sets** > **New Policy Set**, and add a policy set with the following values:

        - **Id** : `PEP-CDSSO`
        - **Resource Types** : `URL`
            - In the new policy set, add a policy with the following values:
        - **Name** : `CDSSO`
        - **Resource Type** : `URL`
        - **Resource pattern** : `*://*:*/*`
        - **Resource value** : `http://app.example.com:8081/home/pep-cdsso*`

          This policy protects the home page of the sample application.
        - On the **Actions** tab, add an action to allow HTTP `GET`.
        - On the **Subjects** tab, remove any default subject conditions, add a subject condition for all `Authenticated Users`.

2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/04-pep-cdsso.json
```

Windows

```
%appdata%\OpenIG\config\routes\04-pep-cdsso.json
```

```
{
  "name": "pep-cdsso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/pep-cdsso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent_cdsso",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "CrossDomainSingleSignOnFilter-1",
          "type": "CrossDomainSingleSignOnFilter",
          "config": {
            "redirectEndpoint": "/home/pep-cdsso/redirect",
            "authCookie": {
              "path": "/home",
              "name": "ig-token-cookie"
            },
            "amService": "AmService-1"
          }
        },
        {
          "name": "PolicyEnforcementFilter-1",
          "type": "PolicyEnforcementFilter",
          "config": {
            "application": "PEP-CDSSO",
            "ssoTokenSubject": "${contexts.cdsso.token}",
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Warning

When `verificationSecretId` isn't configured, IG discovers and uses the AM JWK set to verify the signature of AM session tokens. If the JWK set isn't available, IG doesn't verify the tokens.

3. Test the setup:

    1. In your browser's privacy or incognito mode, go to to https://ig.ext.com:8443/home/pep-cdsso.
    2. If you see warnings that the site isn't secure, respond to the warnings to access the site.

       IG redirects you to AM for authentication.
    3. Log in to AM as user `demo`, password `Ch4ng31t`.

       When you have authenticated, AM redirects you back to the request URL, and IG requests a policy decision. AM returns a policy decision that grants access to the sample application.

Enforce policy decisions using claimsSubject

This example extends Enforce AM policy decisions in the same domain to enforce a policy decision from AM, using `claimsSubject` instead of `ssoTokenSubject` to identify the subject.

Before you start, set up and test the example in Enforce AM policy decisions in the same domain.

1. Set up AM:

    1. Select the policy `PEP-SSO` and add a new resource:

        - **Resource Type**: `URL`
        - **Resource pattern**: `*://*:*/*`
        - **Resource value**: `http://app.example.com:8081/home/pep-claims`
    2. In the same policy, add the following subject condition:

        - `Any of`
        - **Type** : `OpenID Connect/JwtClaim`
        - **claimName** : `iss`
        - **claimValue** : `am.example.com`
2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/04-pep-claims.json
```

Windows

```
%appdata%\OpenIG\config\routes\04-pep-claims.json
```

```
{
  "name": "pep-claims",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/pep-claims')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "PolicyEnforcementFilter-1",
          "type": "PolicyEnforcementFilter",
          "config": {
            "application": "PEP-SSO",
            "claimsSubject": {
              "sub": "${contexts.ssoToken.info.uid}",
              "iss": "am.example.com"
            },
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

3. Test the setup:

    1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/pep-claims.
    2. If you see warnings that the site isn't secure, respond to the warnings to access the site.
    3. Log in to AM as user `demo`, password `Ch4ng31t`.

AM returns a policy decision that grants access to the sample application.

Using WebSocket notifications to evict the policy cache

When WebSocket notifications are enabled, IG receives notifications whenever AM creates, deletes, or changes a policy.

The following procedure gives an example of how to change the configuration in Enforce AM policy decisions in the same domain and Enforce AM policy decisions in different domains to evict outdated entries from the policy cache. For information about WebSocket notifications, refer to WebSocket notifications.

Before you start, set up and test the example in Enforce AM policy decisions in the same domain.

1. Websocket notifications are enabled by default. If they are disabled, enable them by adding the following configuration to the AmService in your route:

```
"notifications": {
  "enabled": true
}
```

2. Enable policy cache in the PolicyEnforcementFilter in your route:

```
"cache": {
  "enabled": true
}
```

3. In `logback.xml` add the following logger for WebSocket notifications, and then restart IG:

```
<logger name="org.forgerock.openig.tools.notifications.ws" level="TRACE" />
```

   For information, refer to Changing the log level for different object types.

4. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.ext.com:8443/home/pep-sso.
   2. If you see warnings that the site isn't secure, respond to the warnings to access the site.
   3. Log in to AM as user `demo`, password `Ch4ng31t`.
   4. In a separate terminal, log on to AM as admin, and change the PEP-SSO policy. For example, in the **Actions** tab, add an action to allow HTTP `DELETE`.
   5. Note that the IG system logs are updated with Websocket notifications about the change:

```
... | TRACE | vert.x-eventloop-thread-14 | o.f.o.t.n.w.l.DirectAmLink | @system | Received a
message: ... "policy": "PEP-SSO", "policySet": "PEP-SSO", "eventType": "UPDATE" } }
... | TRACE | vert.x-eventloop-thread-14 | o.f.o.t.n.w.SubscriptionService | @system | Notification
received, ... "policy": "PEP-SSO", "policySet": "PEP-SSO", "eventType": "UPDATE" }}
... | TRACE | vert.x-eventloop-thread-14 | o.f.o.t.n.w.SubscriptionService | @system | Notification
sent to a [/agent/policy] listener
```

Harden authorization with advice from AM

To protect sensitive resources, AM policies can be configured with additional conditions to harden the authorization. When AM communicates these policy decisions to IG, the decision includes advices to indicate what extra conditions the user must meet.

Conditions can include requirements to access the resource over a secure channel, access during working hours, or to authenticate again at a higher authentication level. For more information, refer to AM's Authorization guide.

The following sections build on the policies in Enforce policy decisions from AM to step up the authentication level:

Step up the authentication level for an AM session

When you step up the authentication level for an AM session, the authorization is verified and then captured as part of the AM session, and the user agent is authorized to that authentication level for the duration of the AM session.

This section uses the policies you created in Enforce AM policy decisions in the same domain and Enforce AM policy decisions in different domains, adding an authorization policy with a Authentication by Service environment condition. Except for the paths where noted, procedures for single domain and cross-domain are the same.

After the user agent redirects the user to AM, if the user is not already authenticated they are presented with a login page. If the user is already authenticated, or after they authenticate, they are presented with a second page asking for a verification code to meet the `AuthenticateToService` environment condition.

Before you start, set up one of the following examples in Enforce AM policy decisions in the same domain or Enforce AM policy decisions in different domains.

1. In the AM admin UI, add an environment condition to the policy:
    1. Select a policy set:
        - For SSO, select 🔑 **Authorization** > **Policy Sets** > **PEP-SSO**.
        - For CDSSO, select 🔑 **Authorization** > **Policy Sets** > **PEP-CDSSO**.
    2. In the policy, select **Environments**, and add the following environment condition:
        - `All of`
        - **Type** : `Authentication by Service`
        - **Authenticate to Service** : `VerificationCodeLevel1`
2. Set up client-side and server-side scripts:
    1. Select </> **Scripts** > **Scripted Module - Client Side**, and replace the default script with the following script:

```
autoSubmitDelay = 60000;

function callback() {
    var parent = document.createElement("div");
    parent.className = "form-group";

    var label = document.createElement("label");
    label.className = "sr-only separator";
    label.setAttribute("for", "answer");
    label.innerText = "Verification Code";
    parent.appendChild(label);

    var input = document.createElement("input");
    input.className = "form-control input-lg";
    input.type = "text";
    input.placeholder = "Enter your verification code";
    input.name = "answer";
    input.id = "answer";
    input.value = "";
    input.oninput = function(event) {
        var element = document.getElementById("clientScriptOutputData");
        if (!element.value || element.value == "clientScriptOutputData") element.value = "{}";
        var json = JSON.parse(element.value);
        json["answer"] = event.target.value;
        element.value = JSON.stringify(json);
    };
    parent.appendChild(input);

    var fieldset = document.forms[0].getElementsByTagName("fieldset")[0];
    fieldset.prepend(parent);
}

if (document.readyState !== 'loading') {
    callback();
} else {
    document.addEventListener("DOMContentLoaded", callback);
}
```

Leave all other values as default.

This client-side script adds a field to the AM form, in which the user is required to enter a verification code. The script formats the entered code as a JSON object, as required by the server-side script.

2. Select **</> Scripts** > **Scripted Module - Server Side**, and replace the default script with the following script:

```
username = 'demo'
logger.error('username: ' + username)

// Test whether the user 'demo' enters the correct validation code
data = JSON.parse(clientScriptOutputData);
answer = data.answer;

if (answer !== '123456') {
  logger.error('Authentication Failed !!')
  authState = FAILED;
} else {
  logger.error('Authenticated !!')
  authState = SUCCESS;
}
```

Leave all other values as default.

This server-side script tests that the user `demo` has entered `123456` as the verification code.

3. Add an authentication module:
    1. Select **Authentication** > **Modules**, and add a module with the following settings:
        - **Name** : `VerificationCodeLevel1`
        - **Type** : `Scripted Module`
    2. In the authentication module, enable the option for client-side script, and select the following options:
        - **Client-side Script** : `Scripted Module - Client Side`
        - **Server-side Script** : `Scripted Module - Server Side`
        - **Authentication Level** : `1`
    3. Add the authentication module to an authentication chain:
        1. Select **Authentication** > **Chains**, and add a chain called `VerificationCodeLevel1` .
        2. Add a module with the following settings:
            - **Select Module** : `VerificationCodeLevel1`
            - **Select Criteria** : `Required`
4. Test the setup:
    1. Log out of AM.
    2. Access the route:
        - For SSO, go to https://ig.example.com:8443/home/pep-sso.
        - For CDSSO, go to https://ig.ext.com:8443/home/pep-cdsso.

          If you haven't previously authenticated to AM, the SingleSignOnFilter redirects the request to AM for authentication.

3. Log in to AM as user `demo`, password `Ch4ng31t`.

   AM creates a session with the default authentication level `0`, and IG requests a policy decision.

   The updated policy requires authentication level `1`, which is higher than the AM session's current authentication level. AM issues a redirect with a `AuthenticateToServiceConditionAdvice` to authenticate at level `1`.

4. In the session upgrade window, enter the verification code `123456`.

   AM upgrades the authentication level for the session to 1, and grants access to the sample application. If you try to access the sample application again in the same session, you don't need to provide the verification code.

Increase authorization for a single transaction

Transactional authorization improves security by requiring a user to perform additional actions when trying to access a resource protected by an AM policy. For example, they must reauthenticate to an authentication module or respond to a push notification on their mobile device.

Performing the additional action successfully grants access to the protected resource, but only once. Additional attempts to access the resource require the user to perform the configured actions again.

This section builds on the example in Step up the authentication level for an AM session, adding a simple authorization policy with a `Transaction` environment condition. Each time the user agent tries to access the protected resource, the user must reauthenticate to an authentication module by providing a verification code.

Before you start, configure AM as described in Step up the authentication level for an AM session. The IG configuration is not changed.

1. In the AM admin UI, add a new environment condition:

   1. Select the policy set:

      - For SSO, select **Authorization** > **Policy Sets** > **PEP-SSO**.
      - For CDSSO, select **Authorization** > **Policy Sets** > **PEP-CDSSO**.

   2. In the IG policy, select Environments and add another environment condition:

      - `All of`
      - **Type** : `Transaction`
      - **Authentication strategy** : `Authenticate To Module`
      - **Strategy specifier** : `TxVerificationCodeLevel5`

2. Set up client-side and server-side scripts:

   1. Select **</> Scripts** > **New Script**, and add the following client-side script:

      - **Name** : `Tx Scripted Module - Client Side`
      - **Script Type** : `Client-side Authentication`

```
autoSubmitDelay = 60000;

function callback() {
    var parent = document.createElement("div");
    parent.className = "form-group";

    var label = document.createElement("label");
    label.className = "sr-only separator";
    label.setAttribute("for", "answer");
    label.innerText = "Verification Code";
    parent.appendChild(label);

    var input = document.createElement("input");
    input.className = "form-control input-lg";
    input.type = "text";
    input.placeholder = "Enter your TX code";
    input.name = "answer";
    input.id = "answer";
    input.value = "";
    input.oninput = function(event) {
        var element = document.getElementById("clientScriptOutputData");
        if (!element.value || element.value == "clientScriptOutputData") element.value = "{}";
        var json = JSON.parse(element.value);
        json["answer"] = event.target.value;
        element.value = JSON.stringify(json);
    };
    parent.appendChild(input);

    var fieldset = document.forms[0].getElementsByTagName("fieldset")[0];
    fieldset.prepend(parent);
}

if (document.readyState !== 'loading') {
    callback();
} else {
    document.addEventListener("DOMContentLoaded", callback);
}
```

This client-side script adds a field to the AM form, in which the user is required to enter a TX code. The script formats the entered code as a JSON object, as required by the server-side script.

2. Select **</> Scripts** > **New Script**, and add the following server-side script:

   - **Name** : `Tx Scripted Module - Server Side`
   - **Script Type** : `Server-side Authentication`

```
username = 'demo'
logger.error('username: ' + username)

// Test whether the user 'demo' enters the correct validation code
data = JSON.parse(clientScriptOutputData);
answer = data.answer;

if (answer !== '789') {
  logger.error('Authentication Failed !!')
  authState = FAILED;
} else {
  logger.error('Authenticated !!')
  authState = SUCCESS;
}
```

This server-side script tests that the user `demo` has entered `789` as the verification code.

3. Add an authentication module:

    1. Select **Authentication** > **Modules**, and add a module with the following settings:

        - **Name** : `TxVerificationCodeLevel5`
        - **Type** : `Scripted Module`

    2. In the authentication module, enable the option for client-side script, and select the following options:

        - **Client-side Script** : `Tx Scripted Module - Client Side`
        - **Server-side Script** : `Tx Scripted Module - Server Side`
        - **Authentication Level** : `5`

4. Test the setup:

    1. Log out of AM.
    2. In your browser's privacy or incognito mode, access your route:

        - For SSO, go to https://ig.example.com:8443/home/pep-sso.
        - For CDSSO, go to https://ig.ext.com:8443/home/pep-cdsso.

    3. If you see warnings that the site isn't secure, respond to the warnings to access the site.

        If you haven't previously authenticated to AM, the SingleSignOnFilter redirects the request to AM for authentication.

    4. Log in to AM as user `demo`, password `Ch4ng31t`.

        AM creates a session with the default authentication level `0`, and IG requests a policy decision.

    5. Enter the verification code `123456` to upgrade the authorization level for the session to `1`.

        The authentication module you configured for transactional authorization requires authentication level `5`, so AM issues a `TransactionConditionAdvice`.

6. In the transaction upgrade window, enter the verification code `789`.

   AM upgrades the authentication level for this policy evaluation to `5`, and then returns a policy decision that grants a one-time access to the sample application. If you try to access the sample application again, you must enter the code again.

OAuth 2.0

OAuth 2.0 includes the following entities:

- *Resource owner* : A user who owns protected resources on a resource server. For example, a resource owner can store photos in a web service.
- *Resource server* : A service that gives authorized client applications access to the resource owner's protected resources. In OAuth 2.0, an Authorization Server grants authorization to a client application, based on the resource owner's consent. For example, a resource server can be a web service that holds a user's photos.
- *Client* : An application that requests access to the resource owner's protected resources, on behalf of the resource owner. For example, a client can be a photo printing service requesting access to a resource owner's photos stored on a web service, after the resource owner gives the client consent to download the photos.
- *Authorization server* : A service responsible for authenticating resource owners, and obtaining their consent to allow client applications to access their resources. For example, AM can act as the OAuth 2.0 Authorization Server to authenticate resource owners and obtain their consent. Other services, such as Google and Facebook can provide OAuth 2.0 authorization services.

IG as an OAuth 2.0 client

IG as an OAuth 2.0 client supports the OAuth 2.0 filters and flows in the following table:

Filter	OAuth 2.0 flow	Description
AuthorizationCodeOAuth2ClientFilter (previously named OAuth2ClientFilter)	Authorization Code Grant	This filter requires the user agent to authorize the request interactively to obtain an access token and optional ID token. The access token is maintained only for the OAuth 2.0 session, and is valid only for the configured scopes. This filter can act as an OpenID Connect relying party or as an OAuth 2.0 client. Use for Web applications running on a server.

Filter	OAuth 2.0 flow	Description
ResourceOwnerOAuth2ClientFilter	Resource Owner Password Credentials Grant	According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and use other grant types when possible. This filter supports the transformation of client credentials and user credentials to obtain an access token from the Authorization Server. It injects the access token into the inbound request as a Bearer Authorization header. The access token is valid only for the configured scopes. Use for clients trusted with the resource owner credentials.
ClientCredentialsOAuth2ClientFilter	Client Credentials Grant	This filter is similar to the Resource Owner Password Credentials grant type, but the resource owner is not part of the flow and the client accesses only information relevant to itself. Use when the client is the resource owner, or the client does not act on behalf of the resource owner.

IG as an OAuth 2.0 resource server

The following image illustrates the steps for a client application to access a user's protected resources, with AM as the Authorization Server and IG as the resource server:

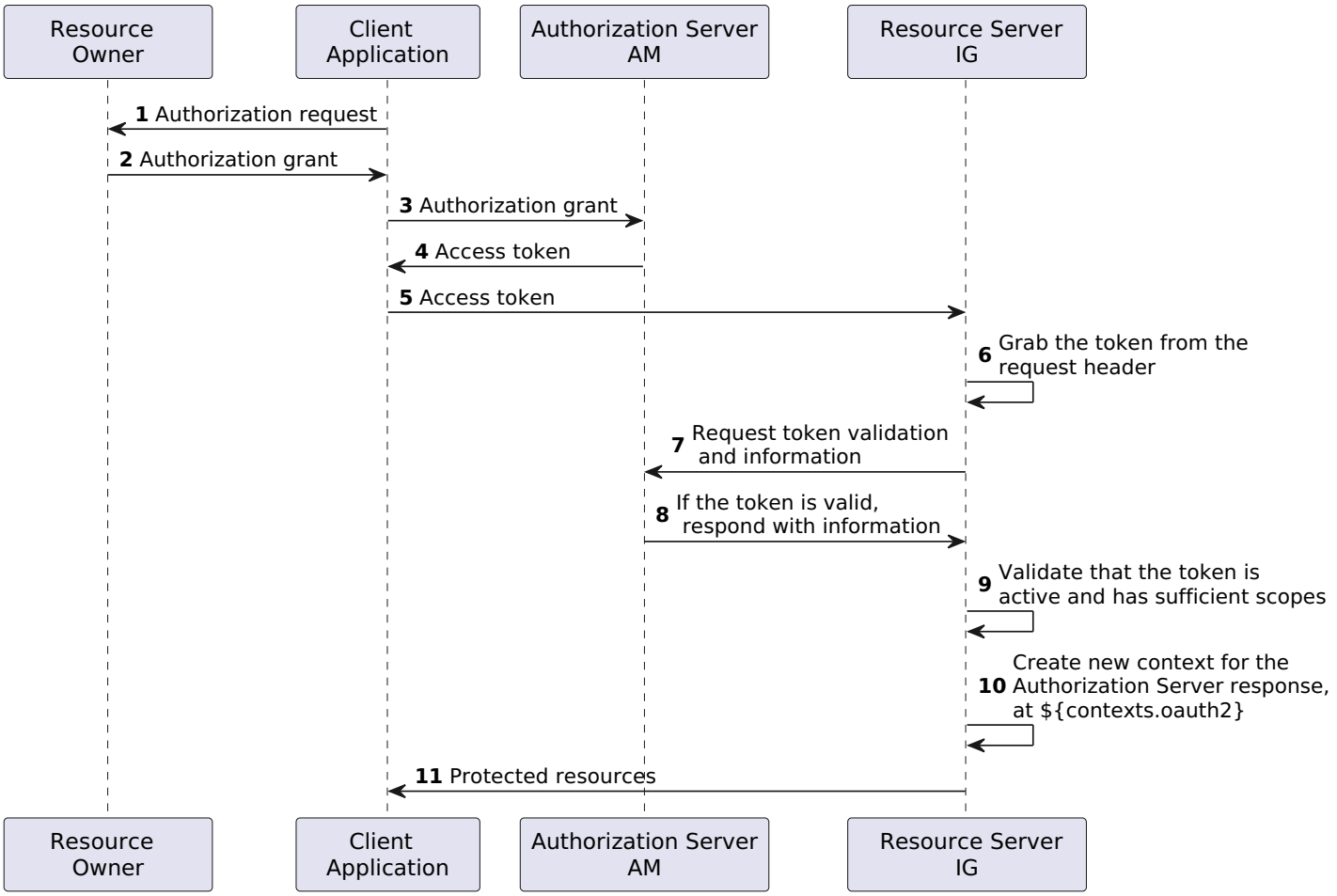


Figure 1. IG as an OAuth 2.0 resource server handling OAuth 2.0 requests

- The application obtains an *authorization grant*, representing the resource owner's consent. For information about the different OAuth 2.0 grant mechanisms supported by AM, refer to OAuth 2.0 grant flows in AM's *OAuth 2.0 guide*.
- The application authenticates to the Authorization Server and requests an *access token*. The Authorization Server returns an access token to the application.

  An OAuth 2.0 access token is an opaque string issued by the authorization server. When the client interacts with the resource server, the client presents the access token in the `Authorization` header. For example:

  ```
  Authorization: Bearer 7af...da9
  ```

  Access tokens are the credentials to access protected resources. The advantage of access tokens over passwords or other credentials is that access tokens can be granted and revoked without exposing the user's credentials.

  The access token represents the authorization to access protected resources. Because an access token is a bearer token, anyone who has the access token can use it to get the resources. Access tokens must therefore be protected, so that requests involving them go over HTTPS.

  In OAuth 2.0, the token scopes are strings that identify the scope of access authorized to the client, but can also be used for other purposes.

- The application supplies the access token to the resource server, which then resolves and validates the access token by using an access token resolver, as described in Access token resolvers.

  If the access token is valid, the resource server permits the client access to the requested resource.

The OAuth2ResourceServerFilter grants access to a resource by using an OAuth 2.0 access token from the HTTP Authorization header of a request.

When auditing is enabled, OAuth 2.0 token tracking IDs can be logged in access audit events for routes that contain an OAuth2ResourceServerFilter. For information, refer to Audit the deployment and Audit framework.

Validate stateful or stateless access tokens through the introspection endpoint

This section sets up IG as an OAuth 2.0 resource server, using the introspection endpoint.

For more information about configuring AM as an OAuth 2.0 authorization service, refer to AM's OAuth 2.0 guide.

> **Important**
>
> This procedure uses the *Resource Owner Password Credentials* grant type. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize other grant types whenever possible.

Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

1. Set up AM:

   1. Select **Applications** > **Agents** > **Identity Gateway**, and register an IG agent with the following values:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`
      - **Token Introspection**: `Realm Only`

      > **Important**
      >
      > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   2. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

      > **Important**
      >
      > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

   3. Create an OAuth 2.0 Authorization Server:

      1. Select **Services** > **Add a Service** > **OAuth2 Provider**.
      2. Add a service with the default values.

4. Create an OAuth 2.0 Client to request OAuth 2.0 access tokens:
    1. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:
        - **Client ID**: `client-application`
        - **Client secret**: `password`
        - **Scope(s)**: `mail`, `employeenumber`
    2. On the **Advanced** tab, select the following value:
        - **Grant Types**: `Resource Owner Password Credentials`

2. Set up IG
    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
    2. Set an environment variable for the IG agent password, and then restart IG:

    ```
    $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
    ```

    The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

    3. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/rs-introspect.json
```

Windows

```
%appdata%\OpenIG\config\routes\rs-introspect.json
```

```
{
  "name": "rs-introspect",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/rs-introspect$')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "TokenIntrospectionAccessTokenResolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        }
```

```
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></
body></html>"
        }
      }
    }
  }
}
```

For information about how to set up the IG route in Studio, see Token validation using the introspection endpoint in Structured Editor.

Notice the following features of the route:

- The route matches requests to `/rs-introspect`.
- The `OAuth2ResourceServerFilter` expects an OAuth 2.0 access token in the `authorization` header of the incoming authorization request, with the scopes `mail` and `employeenumber`.

  The `accessTokenResolver` uses the AM server declared in the heap. The introspection endpoint to validate the access token is extrapolated from the URL of the AM server.

  For convenience in this test, `requireHttps` is false. In production environments, set it to true.
- After the filter validates the access token, it creates a new context from the Authorization Server response. The context is named `oauth2`, and can be reached at `contexts.oauth2` or `contexts['oauth2']`.

  The context contains information about the access token, which can be reached at `contexts.oauth2.accessToken.info`. Filters and handlers further down the chain can access the token info through the context.

  If there is no access token in the request, or token validation does not complete successfully, the filter returns an HTTP error status to the user agent, and IG does not continue processing the request. This is done as specified in the RFC, The OAuth 2.0 Authorization Framework: Bearer Token Usage.
- The HttpBasicAuthenticationClientFilter adds the credentials to the outgoing token introspection request.
- The StaticResponseHandler returns the content of the access token from the context `${contexts.oauth2.accessToken.info}`.

3. Test the setup:

   1. In a terminal window, use a `curl` command similar to the following to retrieve an access token:

```
$ mytoken=$(curl -s \
--user "client-application:password" \
--data "grant_type=password&username=demo&password=Ch4ng31t&scope=mail%20employeenumber" \
http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
```

2. Validate the access token returned in the previous step:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/rs-introspect

{
  active = true,
  scope = employeenumber mail,
  realm=/,
  client_id = client-application,
  user_id = demo,
  token_type = Bearer,
  exp = 158...907,
  ...
}
```

Define required scopes with a script

This example builds on the example in Validate access tokens through the introspection endpoint to use a script to define the scopes that a request requires in an access token.

- If the request path is `/rs-tokeninfo`, the request requires only the scope `mail`.
- If the request path is `/rs-tokeninfo/employee`, the request requires the scopes `mail` and `employeenumber`.

Before you start, set up and test the example in Validate access tokens through the introspection endpoint.

1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/rs-dynamicscope.json
```

Windows

```
%appdata%\OpenIG\rs-dynamicscope.json
```

```
{
  "name": "rs-dynamicscope",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/rs-dynamicscope')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": {
              "name": "myscript",
              "type": "ScriptableResourceAccess",
              "config": {
                "type": "application/x-groovy",
                "source": [
                  "// Minimal set of required scopes",
                  "def scopes = [ 'mail' ] as Set",
                  "if (request.uri.path =~ /employee$/) {",
                  "  // Require another scope to access this resource",
                  "  scopes += 'employeenumber'",
                  "}",
                  "return scopes"
                ]
              }
            },
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "token-resolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
```

```
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></body></
html>"
        }
      }
    }
  }
}
```

2. Test the setup with the `mail` scope only:

    1. In a terminal, use a `curl` command to retrieve an access token with the scope `mail`:

    ```
    $ mytoken=$(curl -s \
    --user "client-application:password" \
    --data "grant_type=password&username=demo&password=Ch4ng31t&scope=mail" \
    http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
    ```

    2. Confirm that the access token is returned for the `/rs-dynamicscope` path:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/rs-dynamicscope

{
  active = true,
  scope = mail,
  client_id = client-application,
  user_id = demo,
  token_type = Bearer,
  exp = 158...907,
  sub = demo,
  iss = http://am.example.com:8088/openam/oauth2, ...
  ...
}
```

3. Confirm that the access token **is not** returned for the `/rs-dynamicscope/employee` path:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/rs-dynamicscope/employee
```

3. Test the setup with the scopes `mail` and `employeenumber`:

1. In a terminal window, use a `curl` command similar to the following to retrieve an access token with the scopes `mail` and `employeenumber`:

```
$ mytoken=$(curl -s \
--user "client-application:password" \
--data "grant_type=password&username=demo&password=Ch4ng31t&scope=mail%20employeenumber" \
http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
```

2. Confirm that the access token **is** returned for the `/rs-dynamicscope/employee` path:

```
$ curl -v
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}"
https://ig.example.com:8443/rs-dynamicscope/employee
```

Validate stateless access tokens with the StatelessAccessTokenResolver

The StatelessAccessTokenResolver confirms that stateless access tokens provided by AM are well-formed, have a valid issuer, have the expected access token name, and have a valid signature.

After the StatelessAccessTokenResolver resolves an access token, the OAuth2ResourceServerFilter checks that the token is within the expiry time, and that it provides the required scopes. For more information, refer to StatelessAccessTokenResolver.

The following sections provide examples of how to validate signed and encrypted access tokens:

Validate signed access tokens with the StatelessAccessTokenResolver and JwkSetSecretStore

This section provides examples of how to validate signed access tokens with the StatelessAccessTokenResolver, using a JwkSetSecretStore. For more information about JwkSetSecretStore, refer to JwkSetSecretStore.

> **Important**
>
> This procedure uses the *Resource Owner Password Credentials* grant type. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize other grant types whenever possible.

1. Set up AM:
    1. Configure an OAuth 2.0 Authorization Provider:
        1. Select **Services**, and add an OAuth 2.0 Provider.
        2. Accept the default values and select **Create**. The service is added to the **Services** list.
        3. On the **Core** tab, select the following option:
            - **Use Client-Based Access & Refresh Tokens** : `on`
        4. On the **Advanced** tab, select the following options:
            - **Client Registration Scope Allowlist** : `myscope`
            - **OAuth2 Token Signing Algorithm** : `RS256`
            - **Encrypt Client-Based Tokens** : Deselected
    2. Create an OAuth2 Client to request OAuth 2.0 access tokens:
        1. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:
            - **Client ID** : `client-application`
            - **Client secret** : `password`
            - **Scope(s)** : `myscope`
        2. On the **Advanced** tab, select the following values:
            - **Grant Types** : `Resource Owner Password Credentials`
            - **Response Types** : `code token`
        3. On the **Signing and Encryption** tab, include the following setting:
            - **ID Token Signing Algorithm** : `RS256`
2. Set up IG:
    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
    2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/rs-stateless-signed.json
```

Windows

```
%appdata%\OpenIG\config\routes\rs-stateless-signed.json
```

```
{
  "name": "rs-stateless-signed",
  "condition": "${find(request.uri.path, '/rs-stateless-signed')}",
  "heap": [
    {
      "name": "SecretsProvider-1",
      "type": "SecretsProvider",
      "config": {
        "stores": [
          {
            "type": "JwkSetSecretStore",
            "config": {
              "jwkUrl": "http://am.example.com:8088/openam/oauth2/connect/jwk_uri"
            }
          }
        ]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": ["myscope"],
            "requireHttps": false,
            "accessTokenResolver": {
              "type": "StatelessAccessTokenResolver",
              "config": {
                "secretsProvider": "SecretsProvider-1",
                "issuer": "http://am.example.com:8088/openam/oauth2",
                "verificationSecretId": "any.value.in.regex.format"
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></
body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/rs-stateless-signed`.

- A SecretsProvider in the heap declares a JwkSetSecretStore to manage secrets for signed access tokens.
- The JwkSetSecretStore specifies the URL to a JWK set on AM, that contains the signing keys.
- The OAuth2ResourceServerFilter expects an OAuth 2.0 access token in the header of the incoming authorization request, with the scope `myscope`.
- The StatelessAccessTokenResolver uses the SecretsProvider to verify the signature of the provided access token.
- After the OAuth2ResourceServerFilter validates the access token, it creates the OAuth2Context context. For more information, refer to OAuth2Context.
- If there is no access token in a request, or token validation does not complete successfully, the filter returns an HTTP error status to the user agent, and IG does not continue processing the request. This is done as specified in the RFC The OAuth 2.0 Authorization Framework: Bearer Token Usage.
- The StaticResponseHandler returns the content of the access token from the context.

3. Test the setup for a signed access token:

   1. Get an access token for the demo user, using the scope `myscope`:

      ```
      $ mytoken=$(curl -s \
      --user "client-application:password" \
      --data "grant_type=password&username=demo&password=Ch4ng31t&scope=myscope" \
      http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
      ```

   2. Display the token:

      ```
      $ echo ${mytoken}
      ```

      Note that the token is structured as a signed token.

   3. Access the route by providing the token returned in the previous step:

      ```
      $ curl -v \
      --cacert /path/to/secrets/ig.example.com-certificate.pem \
      --header "Authorization: Bearer ${mytoken}" \
      https://ig.example.com:8443/rs-stateless-signed

      ...
          Decoded access_token: {
          sub=(usr!demo),
          cts=OAUTH2_STATELESS_GRANT,
          ...
      ```

Validate signed access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore

This section provides examples of how to validate signed access tokens with the StatelessAccessTokenResolver, using a KeyStoreSecretStore. For more information about KeyStoreSecretStore, refer to KeyStoreSecretStore.

Set up keys to sign access tokens

1. Locate the following directories for keys, keystores, and certificates, and in a terminal create variables for them:
   - Directory where the keystore is created: `keystore_directory`
   - AM keystore directory: `am_keystore_directory`
   - IG keystore directory: `ig_keystore_directory`
2. Set up the keystore for signing keys:
   1. Generate a private key called `signature-key`, and a corresponding public certificate called `x509certificate.pem`:

      ```
      $ openssl req -x509 \
      -newkey rsa:2048 \
      -nodes \
      -subj "/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
      -keyout $keystore_directory/signature-key.key \
      -out $keystore_directory/x509certificate.pem \
      -days 365

      ...
      writing new private key to '$keystore_directory/signature-key.key'
      ```

   2. Convert the private key and certificate files into a PKCS#12 file, called `signature-key`, and store them in a keystore named `keystore.p12`:

      ```
      $ openssl pkcs12 \
      -export \
      -in $keystore_directory/x509certificate.pem \
      -inkey $keystore_directory/signature-key.key \
      -out $keystore_directory/keystore.p12 \
      -passout pass:password \
      -name signature-key
      ```

   3. List the keys in `keystore.p12`:

      ```
      $ keytool -list \
      -v \
      -keystore "$keystore_directory/keystore.p12" \
      -storepass "password" \
      -storetype PKCS12

      ...
      Your keystore contains 1 entry
      Alias name: signature-key
      ```

3. Set up keys for AM:
   1. Copy the signing key `keystore.p12` to AM:

```
$ cp $keystore_directory/keystore.p12 $am_keystore_directory/AM_keystore.p12
```

2. List the keys in the AM keystore:

```
$ keytool -list \
-v \
-keystore "$am_keystore_directory/AM_keystore.p12" \
-storepass "password" \
-storetype PKCS12

...
Your keystore contains 1 entry
Alias name: signature-key
```

3. Add a file called `keystore.pass`, containing the store password `password`:

```
$ cd $am_keystore_directory
$ echo -n 'password' > keystore.pass
```

> **Note**
>
> Make sure the password file contains only the password, with no trailing spaces or carriage returns.

The filename corresponds to the secret ID of the store password and entry password for the KeyStoreSecretStore.

4. Restart AM.

4. Set up keys for IG:

1. Import the public certificate to the IG keystore, with the alias `verification-key`:

```
$ keytool -import \
-trustcacerts \
-rfc \
-alias verification-key \
-file "$keystore_directory/x509certificate.pem" \
-keystore "$ig_keystore_directory/IG_keystore.p12" \
-storetype PKCS12 \
-storepass "password"

...
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

2. List the keys in the IG keystore:

```
$ keytool -list \
-v \
-keystore "$ig_keystore_directory/IG_keystore.p12" \
-storepass "password" \
-storetype PKCS12

...
Your keystore contains 1 entry
Alias name: verification-key
```

3. In the IG configuration, set an environment variable for the keystore password:

```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='
```

4. Restart IG.

Validate signed access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore

> **Important**
>
> This procedure uses the *Resource Owner Password Credentials* grant type. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize other grant types whenever possible.

1. Set up AM:
    1. Create a KeyStoreSecretStore to manage the new AM keystore:
        1. In AM, select **Secret Stores**, and then add a secret store with the following values:
            - **Secret Store ID** : `keystoresecretstore`
            - **Store Type** : `Keystore`
            - **File** : `am_keystore_directory/AM_keystore.p12`
            - **Keystore type** : `PKCS12`
            - **Store password secret ID** : `keystore.pass`
            - **Entry password secret ID** : `keystore.pass`
        2. Select the **Mappings** tab, and add a mapping with the following values:
            - **Secret ID** : `am.services.oauth2.stateless.signing.RSA`
            - **Aliases** : `signature-key`

              The mapping sets `signature-key` as the active alias to use for signature generation.
    2. Create a FileSystemSecretStore to manage secrets for the KeyStoreSecretStore:
        1. Select **Secret Stores**, and then create a secret store with the following configuration:
            - **Secret Store ID** : `filesystemsecretstore`

- **Store Type** : `File System Secret Volumes`
- **Directory** : `am_keystore_directory`
- **File format** : `Plain text`

3. Configure an OAuth 2.0 Authorization Provider:
    1. Select **Services**, and add an OAuth 2.0 Provider.
    2. Accept all of the default values, and select **Create**. The service is added to the **Services** list.
    3. On the **Core** tab, select the following option:
        - **Use Client-Based Access & Refresh Tokens** : `on`
    4. On the **Advanced** tab, select the following options:
        - **Client Registration Scope Allowlist** : `myscope`
        - **OAuth2 Token Signing Algorithm** : `RS256`
        - **Encrypt Client-Based Tokens** : Deselected
4. Create an OAuth2 Client to request OAuth 2.0 access tokens:
    1. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:
        - **Client ID** : `client-application`
        - **Client secret** : `password`
        - **Scope(s)** : `myscope`
    2. On the **Advanced** tab, select the following values:
        - **Grant Types** : `Resource Owner Password Credentials`
        - **Response Types** : `code token`
    3. On the **Signing and Encryption** tab, include the following setting:
        - **ID Token Signing Algorithm** : `RS256`

2. Set up IG:
    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
    2. Add the following route to IG, replacing ig_keystore_directory:

Linux

```
$HOME/.openig/config/routes/rs-stateless-signed-ksss.json
```

Windows

```
%appdata%\OpenIG\config\routes\rs-stateless-signed-ksss.json
```

```
{
  "name": "rs-stateless-signed-ksss",
  "condition" : "${find(request.uri.path, '/rs-stateless-signed-ksss')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "KeyStoreSecretStore-1",
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "<ig_keystore_directory>/IG_keystore.p12",
        "storeType": "PKCS12",
        "storePasswordSecretId": "keystore.secret.id",
        "entryPasswordSecretId": "keystore.secret.id",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "mappings": [
          {
            "secretId": "stateless.access.token.verification.key",
            "aliases": [ "verification-key" ]
          }
        ]
      }
    }
  ],
  "handler" : {
    "type" : "Chain",
    "capture" : "all",
    "config" : {
      "filters" : [ {
        "name" : "OAuth2ResourceServerFilter-1",
        "type" : "OAuth2ResourceServerFilter",
        "config" : {
          "scopes" : [ "myscope" ],
          "requireHttps" : false,
          "accessTokenResolver": {
            "type": "StatelessAccessTokenResolver",
            "config": {
              "secretsProvider": "KeyStoreSecretStore-1",
              "issuer": "http://am.example.com:8088/openam/oauth2",
              "verificationSecretId": "stateless.access.token.verification.key"
            }
          }
        }
      } ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></
body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/rs-stateless-signed-ksss`.
- The keystore password is provided by the SystemAndEnvSecretStore in the heap.
- The OAuth2ResourceServerFilter expects an OAuth 2.0 access token in the header of the incoming authorization request, with the scope `myscope`.
- The `accessTokenResolver` uses a `StatelessAccessTokenResolver` to resolve and verify the authenticity of the access token. The secret is provided by the KeyStoreSecretStore in the heap.
- After the OAuth2ResourceServerFilter validates the access token, it creates the `OAuth2Context` context. For more information, refer to OAuth2Context.
- If there is no access token in a request, or if the token validation does not complete successfully, the filter returns an HTTP error status to the user agent, and IG stops processing the request, as specified in the RFC, The OAuth 2.0 Authorization Framework: Bearer Token Usage.
- The StaticResponseHandler returns the content of the access token from the context.

3. Test the setup for a signed access token:

   1. Get an access token for the demo user, using the scope `myscope`:

      ```
      $ mytoken=$(curl -s \
      --user "client-application:password" \
      --data "grant_type=password&username=demo&password=Ch4ng31t&scope=myscope" \
      http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
      ```

   2. Display the token:

      ```
      $ echo ${mytoken}
      ```

   3. Access the route by providing the token returned in the previous step:

      ```
      $ curl -v \
      --cacert /path/to/secrets/ig.example.com-certificate.pem \
      --header "Authorization: Bearer ${mytoken}" \
      https://ig.example.com:8443/rs-stateless-signed-ksss

      ...
      Decoded access_token: {
      sub=(usr!demo),
      cts=OAUTH2_STATELESS_GRANT,
      ...
      ```

Validating encrypted access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore

Set up keys to encrypt access tokens

1. Locate the following directories for keys, keystores, and certificates, and in a terminal create variables for them:
   - Directory where the keystore is created: `keystore_directory`
   - AM keystore directory: `am_keystore_directory`
   - IG keystore directory: `ig_keystore_directory`
2. Set up keys for AM:
   1. Generate the encryption key:

      ```
      $ keytool -genseckey \
      -alias encryption-key \
      -dname "CN=ig.example.com, OU=example, O=com, L=fr, ST=fr, C=fr" \
      -keystore "$am_keystore_directory/AM_keystore.p12" \
      -storetype PKCS12 \
      -storepass "password" \
      -keyalg AES \
      -keysize 256
      ```

   2. List the keys in the AM keystore:

      ```
      $ keytool -list \
      -v \
      -keystore "$am_keystore_directory/AM_keystore.p12" \
      -storepass "password" \
      -storetype PKCS12

      ...
      Your keystore contains 1 entry
      Alias name: encryption-key
      ```

   3. Add a file called `keystore.pass`, with the content `password`:

      ```
      $ cd $am_keystore_directory
      $ echo -n 'password' > keystore.pass
      ```

      > **Note**
      >
      > Make sure the password file contains only the password, with no trailing spaces or carriage returns.

      The filename corresponds to the secret ID of the store password and entry password for the KeyStoreSecretStore.

   4. Restart AM.

3. Set up keys for IG:

    1. Import `encryption-key` into the IG keystore, with the alias `decryption-key`:

    ```
    $ keytool -importkeystore \
    -srcalias encryption-key \
    -srckeystore "$am_keystore_directory/AM_keystore.p12" \
    -srcstoretype PKCS12 \
    -srcstorepass "password" \
    -destkeystore "$ig_keystore_directory/IG_keystore.p12" \
    -deststoretype PKCS12 \
    -destalias decryption-key \
    -deststorepass "password" \
    -destkeypass "password"
    ```

    2. List the keys in the IG keystore:

    ```
    $ keytool -list \
    -v \
    -keystore "$ig_keystore_directory/IG_keystore.p12" \
    -storepass "password" \
    -storetype PKCS12

    ...
    Your keystore contains 1 entry
    Alias name: decryption-key
    ```

    3. In the IG configuration, set an environment variable for the keystore password:

    ```
    $ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='
    ```

    4. Restart IG.

Validate encrypted access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore

1. Set up AM:

    1. Set up AM as described in Validate signed access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore.

    2. Add a mapping for the encryption keystore:

        1. Select **Secret Stores** > `keystoresecretstore`.

        2. Select the **Mappings** tab, and add a mapping with the following values:

            - **Secret ID**: `am.services.oauth2.stateless.token.encryption`
            - **Alias**: `encryption-key`

    3. Enable token encryption on the OAuth 2.0 Authorization Provider:

        1. Select **Services** > **OAuth2 Provider**.

2. On the **Advanced** tab, select **Encrypt Client-Side Tokens**.

2. Set up IG:

   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

   2. Add the following route to IG, replacing ig_keystore_directory:

Linux

```
$HOME/.openig/config/routes/rs-stateless-encrypted.json
```

Windows

```
%appdata%\OpenIG\config\routes\rs-stateless-encrypted.json
```

```
{
  "name": "rs-stateless-encrypted",
  "condition": "${find(request.uri.path, '/rs-stateless-encrypted')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "KeyStoreSecretStore-1",
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "<ig_keystore_directory>/IG_keystore.p12",
        "storeType": "PKCS12",
        "storePasswordSecretId": "keystore.secret.id",
        "entryPasswordSecretId": "keystore.secret.id",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "mappings": [
          {
            "secretId": "stateless.access.token.decryption.key",
            "aliases": [ "decryption-key" ]
          }
        ]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [ {
        "name": "OAuth2ResourceServerFilter-1",
        "type": "OAuth2ResourceServerFilter",
        "config": {
          "scopes": [ "myscope" ],
          "requireHttps": false,
          "accessTokenResolver": {
            "type": "StatelessAccessTokenResolver",
            "config": {
              "secretsProvider": "KeyStoreSecretStore-1",
              "issuer": "http://am.example.com:8088/openam/oauth2",
              "decryptionSecretId": "stateless.access.token.decryption.key"
            }
          }
        }
      } ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></
body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route compared to `rs-stateless-signed.json`, used in: Validate signed access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore:

- The route matches requests to `/rs-stateless-encrypted`.
- The OAuth2ResourceServerFilter and KeyStoreSecretStore refer to the configuration for a decryption key instead of a verification key.

3. Test the setup

1. Get an access token for the demo user, using the scope `myscope`:

```
$ mytoken=$(curl -s \
--user "client-application:password" \
--data "grant_type=password&username=demo&password=Ch4ng31t&scope=myscope" \
http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
```

2. Display the token:

```
$ echo ${mytoken}
```

Note that the token is structured as an encrypted token.

3. Access the route by providing the token returned in the previous step:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/rs-stateless-encrypted

...
Decoded access_token: {
sub=demo,
cts=OAUTH2_STATELESS_GRANT,
...
```

Validate certificate-bound access tokens

Clients can authenticate to AM through mutual TLS (mTLS) and X.509 certificates. Certificates must be self-signed or use public key infrastructure (PKI), as described in version 12 of the draft OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens.

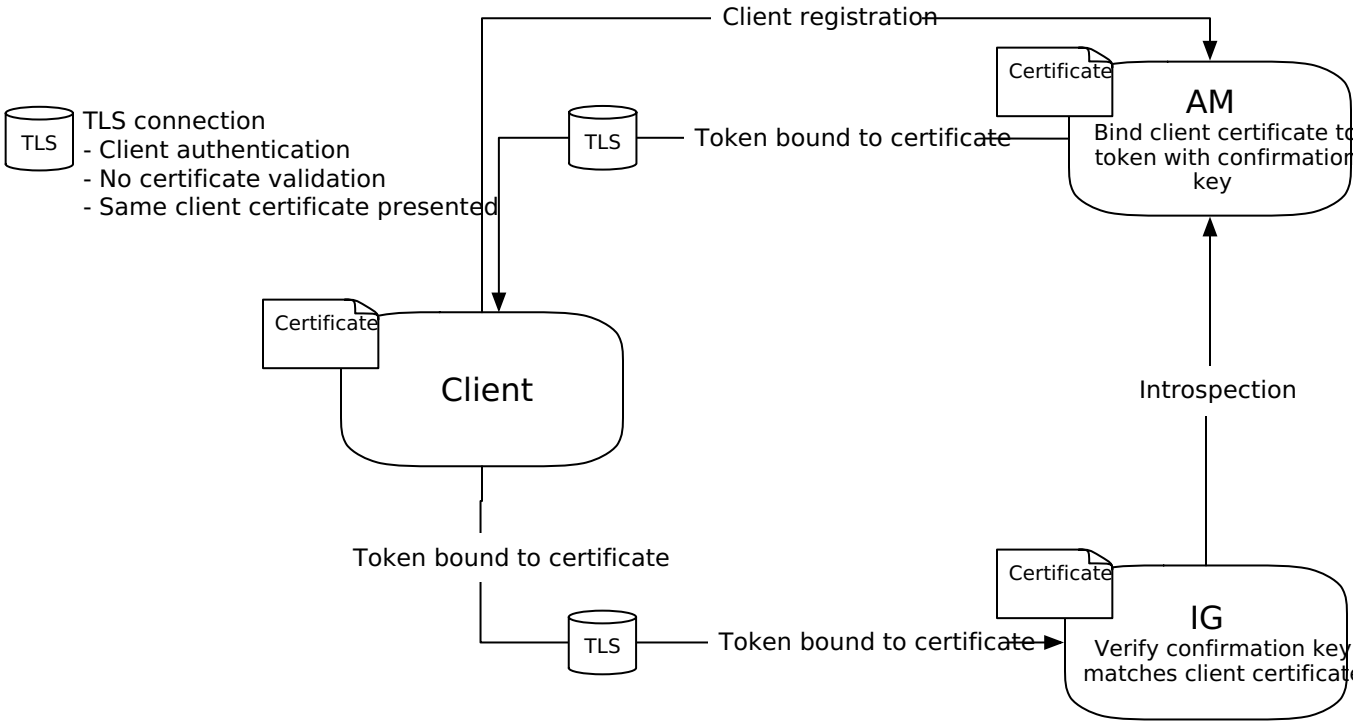
When a client requests an access token from AM through mTLS, AM can use a *confirmation key* to bind the access token to the presented client certificate. The confirmation key is the certificate *thumbprint*, computed as `base64url-encode(sha256(der(certificate)))`. The access token is then *certificate-bound*. For more information, refer to Mutual TLS in AM's *OAuth 2.0 guide*.

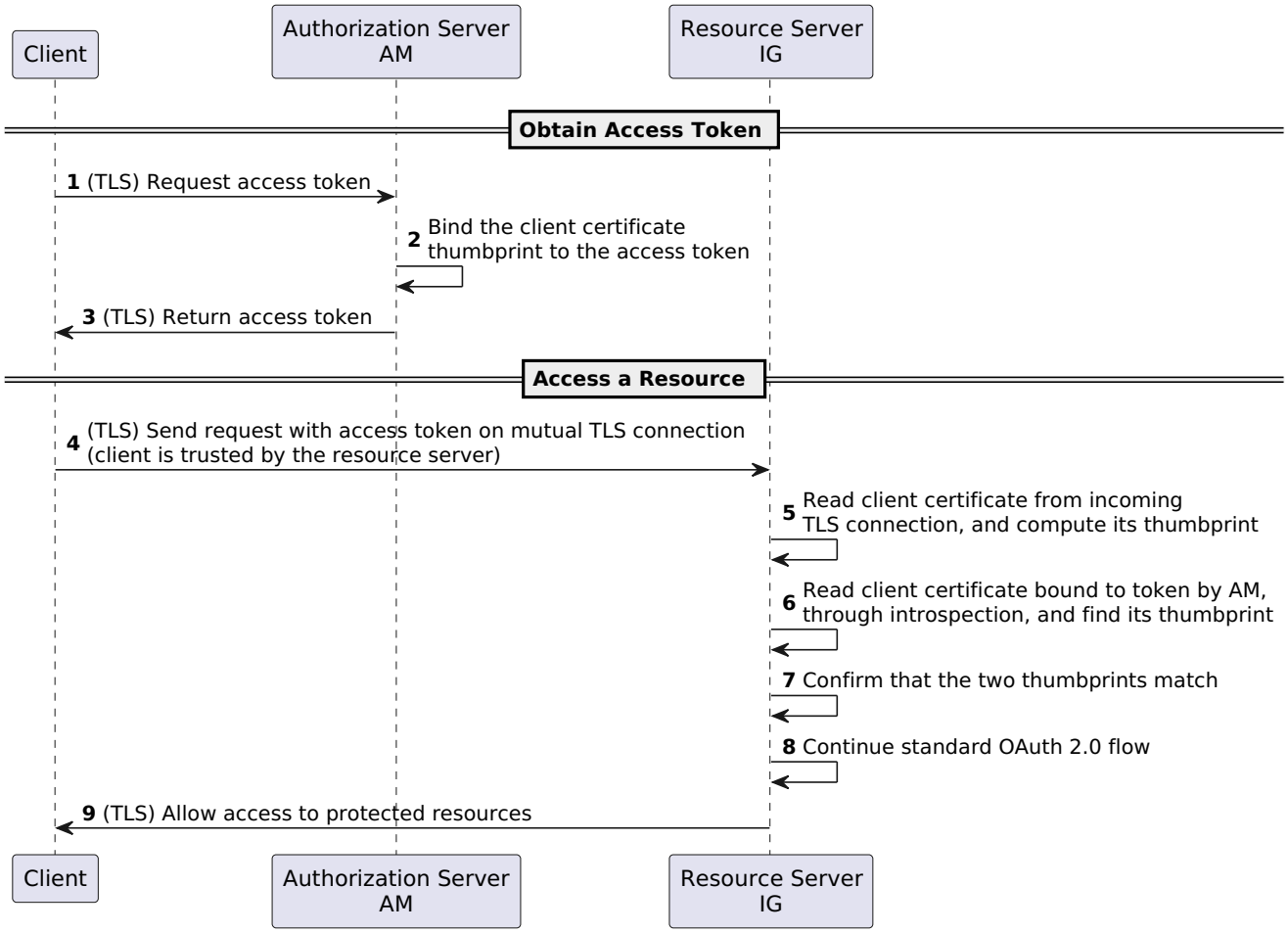
When the client connects to IG by using that certificate, IG can verify that the confirmation key corresponds to the presented certificate. This proof-of-possession interaction ensures that only the client in possession of the key corresponding to the certificate can use the access token to access protected resources.

mTLS using standard TLS client certificate authentication

IG can validate the thumbprint of certificate-bound access tokens by reading the client certificate from the TLS connection.

For this example, the client must be connected directly to IG through a TLS connection, for which IG is the TLS termination point. If TLS is terminated at a reverse proxy or load balancer before IG, use the example in mTLS Using Trusted Headers.





Perform the procedures in this section to set up and test mTLS using standard TLS client certificate authentication:

Set up keystores and truststores

1. Locate the following keystore directories, and in a terminal create variables for them:
   - `oauth2_client_keystore_directory`
   - `am_keystore_directory`
   - `ig_keystore_directory`
2. Create self-signed RSA key pairs for AM, IG, and the client:

```
$ keytool -genkeypair \
-alias openam-server \
-keyalg RSA \
-keysize 2048 \
-keystore $am_keystore_directory/keystore.p12 \
-storepass changeit \
-storetype PKCS12 \
-keypass changeit \
-validity 360 \
-dname CN=am.example.com,O=Example,C=FR
```

```
$ keytool -genkeypair \
-alias openig-server \
-keyalg RSA \
-keysize 2048 \
-keystore $ig_keystore_directory/keystore.p12 \
-storepass changeit \
-storetype PKCS12 \
-keypass changeit \
-validity 360 \
-dname CN=ig.example.com,O=Example,C=FR
```

```
$ keytool -genkeypair \
-alias oauth2-client \
-keyalg RSA \
-keysize 2048 \
-keystore $oauth2_client_keystore_directory/keystore.p12 \
-storepass changeit \
-storetype PKCS12 \
-keypass changeit \
-validity 360 \
-dname CN=test
```

3. Export the certificates to .pem so that the `curl` client can verify the identity of the AM and IG servers:

```
$ keytool -export \
-rfc \
-alias openam-server \
-keystore $am_keystore_directory/keystore.p12 \
-storepass changeit \
-storetype PKCS12 \
-file $am_keystore_directory/openam-server.cert.pem

Certificate stored in file .../openam-server.cert.pem
```

```
$ keytool -export \
-rfc \
-alias openig-server \
-keystore $ig_keystore_directory/keystore.p12 \
-storepass changeit \
-storetype PKCS12 \
-file $ig_keystore_directory/openig-server.cert.pem

Certificate stored in file openig-server.cert.pem
```

4. Extract the certificate and client private key to .pem so that the `curl` command can identity itself as the client for the HTTPS connection:

```
$ keytool -export \
-rfc \
-alias oauth2-client \
-keystore $oauth2_client_keystore_directory/keystore.p12 \
-storepass changeit \
-storetype PKCS12 \
-file $oauth2_client_keystore_directory/client.cert.pem

Certificate stored in file .../client.cert.pem
```

```
$ openssl pkcs12 \
-in $oauth2_client_keystore_directory/keystore.p12 \
-nocerts \
-nodes \
-passin pass:changeit \
-out $oauth2_client_keystore_directory/client.key.pem

...verified OK
```

You can now delete the client keystore.

5. Create the CACerts truststore so that AM can validate the client identity:

```
$ keytool -import \
-noprompt \
-trustcacerts \
-file $oauth2_client_keystore_directory/client.cert.pem \
-keystore $oauth2_client_keystore_directory/cacerts.p12 \
-storepass changeit \
-storetype PKCS12 \
-alias client-cert

Certificate was added to keystore
```

Set up AM for HTTPS (server-side) in Tomcat

This procedure sets up AM for HTTPS in Tomcat. For more information, see Secure connections to the AM container in AM's *Installation guide*.

1. Add a connector configuration for port `8445` to AM's Tomcat `server.xml`, replacing the values for the keystore directories with your path. If the file already contains a connector for the port, edit that connector or replace it:

```
<Connector port="8445" protocol="HTTP/1.1" SSLEnabled="true" scheme="https" secure="true">
  <SSLHostConfig protocols="+TLSv1.2,-TLSv1.1,-TLSv1,-SSLv2Hello,-SSLv3"
                 certificateVerification="optionalNoCA"
                 truststoreFile="oauth2_client_keystore_directory/cacerts.p12"
                 truststorePassword="changeit"
                 truststoreType="PKCS12">
    <Certificate certificateKeystoreFile="am_keystore_directory/keystore.p12"
                 certificateKeystorePassword="changeit"
                 certificateKeystoreType="PKCS12"/>
  </SSLHostConfig>
</Connector>
```

The `optionalNoCA` property allows the presentation of client certificates to be optional. Tomcat does not check them against the list of trusted CAs.

2. In AM, export an environment variable for the base64-encoded value of the password (`changeit`) for the `cacerts.p12` truststore:

```
$ export PASSWORDSECRETID='Y2hhbmdlaXQ='
```

3. Restart AM, and make sure you can access it on the secure port `https://am.example.com:8445/openam` .

Set up IG for HTTPS (server-side)

This procedure sets up IG for HTTPS. Before you start, install IG as described in the Install.

1. In ig_keystore_directory, add a file called `keystore.pass` containing the keystore password:

```
$ cd $ig_keystore_directory
$ echo -n 'changeit' > keystore.pass
```

> **Note**
>
> Make sure the password file contains only the password, with no trailing spaces or carriage returns.

2. Add the following configuration to IG, replacing instances of ig_keystore_directory and oauth2_client_keystore_directory with your path:

Linux

```
$HOME/.openig/config/admin.json
```

Windows

```
%appdata%\OpenIG\config\admin.json
```

```
{
  "mode": "DEVELOPMENT",
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443,
      "tls": {
        "type": "ServerTlsOptions",
        "config": {
          "alpn": {
            "enabled": true
          },
          "clientAuth": "REQUEST",
          "keyManager": {
            "type": "SecretsKeyManager",
            "config": {
              "signingSecretId": "key.manager.secret.id",
              "secretsProvider": {
                "type": "KeyStoreSecretStore",
                "config": {
                  "file": "<ig_keystore_directory>/keystore.p12",
                  "storePasswordSecretId": "keystore.pass",
                  "secretsProvider": "SecretsPasswords",
                  "mappings": [
                    {
                      "secretId": "key.manager.secret.id",
                      "aliases": [
                        "openig-server"
                      ]
                    }
                  ]
                }
              }
            }
          },
          "trustManager": {
            "name": "SecretsTrustManager-1",
            "type": "SecretsTrustManager",
            "config": {
              "verificationSecretId": "trust.manager.secret.id",
              "secretsProvider": {
                "type": "KeyStoreSecretStore",
                "config": {
                  "file": "<oauth2_client_keystore_directory>/cacerts.p12",
                  "storePasswordSecretId": "keystore.pass",
                  "secretsProvider": "SecretsPasswords",
                  "mappings": [
                    {
                      "secretId": "trust.manager.secret.id",
                      "aliases": [
                        "client-cert"
                      ]
                    }
                  ]
                }
              }
            }
          }
```

```
        }
      }
    }
  ],
  "heap": [
    {
      "name": "SecretsPasswords",
      "type": "FileSystemSecretStore",
      "config": {
        "directory": "<ig_keystore_directory>",
        "format": "PLAIN"
      }
    }
  ]
}
```

Notice the following features of the configuration:

- IG starts on port `8080`, and on `8443` over TLS.
- IG's private keys for TLS are managed by the SecretsKeyManager, which references the KeyStoreSecretStore that holds the keys.
- The password of the KeyStoreSecretStore is provided by the FileSystemSecretStore.
- The KeyStoreSecretStore maps the keystore alias to the secret ID for retrieving the private signing keys.

3. Start IG:

Linux

```
$ /path/to/identity-gateway-2024.3.0/bin/start.sh

...
... started in 1234ms on ports : [8080 8443]
```

Windows

```
C:\path\to\identity-gateway-2024.3.0\bin\start.bat
```

By default, IG configuration files are located under `$HOME/.openig` (on Windows `%appdata%\OpenIG`). For information about how to use a different location, refer to Configuration location.

Set up AM as an Authorization Server with mTLS

1. In a the AM admin UI, select **Applications** > **Agents** > **Identity Gateway**, and register an IG agent with the following values:
   - **Agent ID**: `ig_agent`
   - **Password**: `password`
   - **Token Introspection**: `Realm Only`

   > **Important**
   >
   > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

2. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

   > **Important**
   >
   > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

3. Configure an OAuth 2.0 Authorization Server:
   1. Select **Services** > **Add a Service** > **OAuth2 Provider**, and add a service with the default values.
   2. On the **Advanced** tab, select the following value:
      - **Support TLS Certificate-Bound Access Tokens**: enabled
4. Configure an OAuth 2.0 client to request access tokens:
   1. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:
      - **Client ID**: `client-application`
      - **Client secret**: `password`
      - **Scope(s)**: `test`
   2. On the **Advanced** tab, select the following values:
      - **Grant Types**: `Client Credentials`

        The `password` is the only grant type used by the client in the example.
      - **Token Endpoint Authentication Method**: `tls_client_auth`
   3. On the **signing and Encryption** tab, select the following values:
      - **mTLS Subject DN**: `CN=test`

        When this option is set, AM requires the subject DN in the client certificate to have the same value. This ensures that the certificate is from the client, and not just any valid certificate trusted by the trust manager.
      - **Use Certificate-Bound Access Tokens**: Enabled

5. Set up AM secret stores to trust the client certificate:

    1. Select **Secret Stores**, and add a store with the following values:

        - **Secret Store ID**: `trusted-ca-certs`
        - **Store Type**: `Keystore`
        - **File**: `$oauth2_client_keystore_directory/cacerts.p12`
        - **Keystore type**: `PKCS12`
        - **Store password secret ID**: `passwordSecretId`

    2. Select **Mappings** and add the following mapping:

        - **Secret ID**: `am.services.oauth2.tls.client.cert.authentication`
        - **Aliases**: `client-cert`

    When the token endpoint authentication method is `tls_client_auth`, this secret is used to validate the client certificate. Add an alias in this list for each client that uses `tls_client_auth`. For certificates signed by a CA, add the CA certificate to the list.

Set up IG as a resource server with mTLS

1. Set an environment variable for the IG agent password, and then restart IG:

    ```
    $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
    ```

    The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

2. Add the following route to IG:

    Linux

    ```
    $HOME/.openig/config/routes/mtls-certificate.json
    ```

    Windows

    ```
    %appdata%\OpenIG\config\routes\mtls-certificate.json
    ```

```
{
  "name": "mtls-certificate",
  "condition": "${find(request.uri.path, '/mtls-certificate')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "test"
            ],
            "requireHttps": false,
            "accessTokenResolver": {
              "type": "ConfirmationKeyVerifierAccessTokenResolver",
              "config": {
                "delegate": {
                  "name": "token-resolver-1",
                  "type": "TokenIntrospectionAccessTokenResolver",
                  "config": {
                    "amService": "AmService-1",
                    "providerHandler": {
                      "type": "Chain",
                      "config": {
                        "filters": [
                          {
                            "type": "HttpBasicAuthenticationClientFilter",
                            "config": {
                              "username": "ig_agent",
                              "passwordSecretId": "agent.secret.id",
                              "secretsProvider": "SystemAndEnvSecretStore-1"
                            }
                          }
                        ],
                        "handler": "ForgeRockClientHandler"
                      }
                    }
                  }
                }
              }
```

```
              }
            }
          }
        }
      ],
      "handler": {
        "name": "StaticResponseHandler-1",
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/plain; charset=UTF-8" ]
          },
          "entity": "mTLS\n Valid token: ${contexts.oauth2.accessToken.token}\n Confirmation keys: $
{contexts.oauth2}"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/mtls-certificate`.
- The OAuth2ResourceServerFilter uses the ConfirmationKeyVerifierAccessTokenResolver to validate the certificate thumbprint against the thumbprint from the resolved access token, provided by AM.

  The ConfirmationKeyVerifierAccessTokenResolver then delegates token resolution to the TokenIntrospectionAccessTokenResolver.
- The `providerHandler` adds an authorization header to the request, containing the username and password of the OAuth 2.0 client with the scope to examine (introspect) access tokens.
- The OAuth2ResourceServerFilter checks that the resolved token has the required scopes, and injects the token info into the context.
- The StaticResponseHandler returns the content of the access token from the context.

Test the setup

1. Get an access token from AM, over TLS:

```
$ mytoken=$(curl --request POST \
--cacert $am_keystore_directory/openam-server.cert.pem \
--cert $oauth2_client_keystore_directory/client.cert.pem \
--key $oauth2_client_keystore_directory/client.key.pem \
--header 'cache-control: no-cache' \
--header 'content-type: application/x-www-form-urlencoded' \
--data 'client_id=client-application&grant_type=client_credentials&scope=test' \
https://am.example.com:8445/openam/oauth2/access_token | jq -r .access_token)
```

2. Introspect the access token on AM:

```
$ curl --request POST \
-u ig_agent:password \
--header 'content-type: application/x-www-form-urlencoded' \
--data token=${mytoken} \
http://am.example.com:8088/openam/oauth2/realms/root/introspect | jq

{
  "active": true,
  "scope": "test",
  "realm": "/",
  "client_id": "client-application",
  "user_id": "client-application",
  "token_type": "Bearer",
  "exp": 155...833,
  "sub": "(age!client-application)",
  "subname": "client-application",
  "iss": "http://am.example.com:8088/openam/oauth2",
  "cnf": {
    "x51...156": "T4u...R9Q"
  },
  "authGrantId": "dfE...2vk",
  "auditTrackingId": "e36...524"
}
```

The `cnf` property indicates the value of the confirmation code, as follows:

- `x5` : X509 certificate
- `t` : thumbprint
- `#` : separator
- `S256` : algorithm used to hash the raw certificate bytes

3. Access the IG route to validate the token's confirmation thumbprint with the ConfirmationKeyVerifierAccessTokenResolver:

```
$ curl --request POST \
--cacert $ig_keystore_directory/openig-server.cert.pem \
--cert $oauth2_client_keystore_directory/client.cert.pem \
--key $oauth2_client_keystore_directory/client.key.pem \
--header "authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/mtls-certificate

mTLS
  Valid token: 2Bp...s_k
  Confirmation keys: {
  ...
  }
```

The validated token and confirmation keys are displayed.

mTLS using trusted headers

IG can validate the thumbprint of certificate-bound access tokens by reading the client certificate from a configured, trusted HTTP header.

Use this method when TLS is terminated at a reverse proxy or load balancer before IG. IG cannot authenticate the client through the TLS connection's client certificate because:

- If the connection is over TLS, the connection presents the certificate of the TLS termination point before IG.
- If the connection is not over TLS, the connection presents no client certificate.

If the client is connected directly to IG through a TLS connection, for which IG is the TLS termination point, use the example in mTLS Using Standard TLS Client Certificate Authentication.

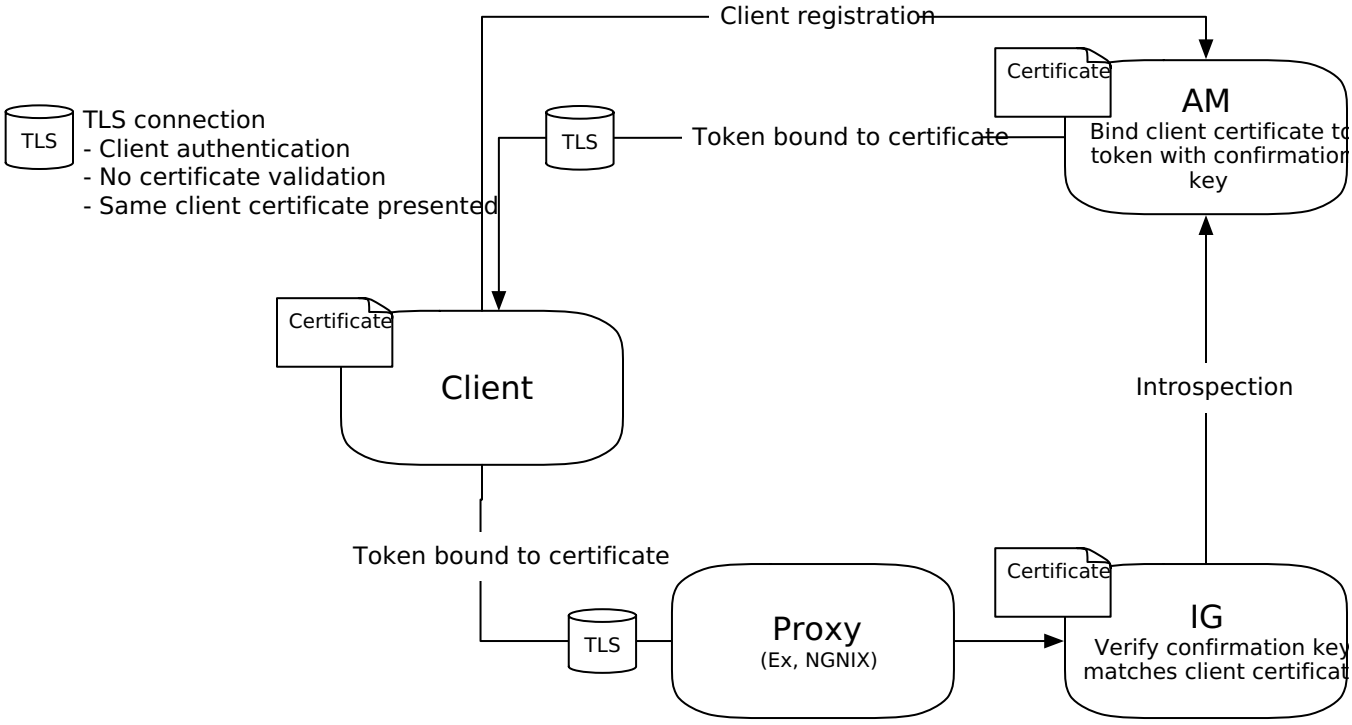
Configure the proxy or load balancer to:

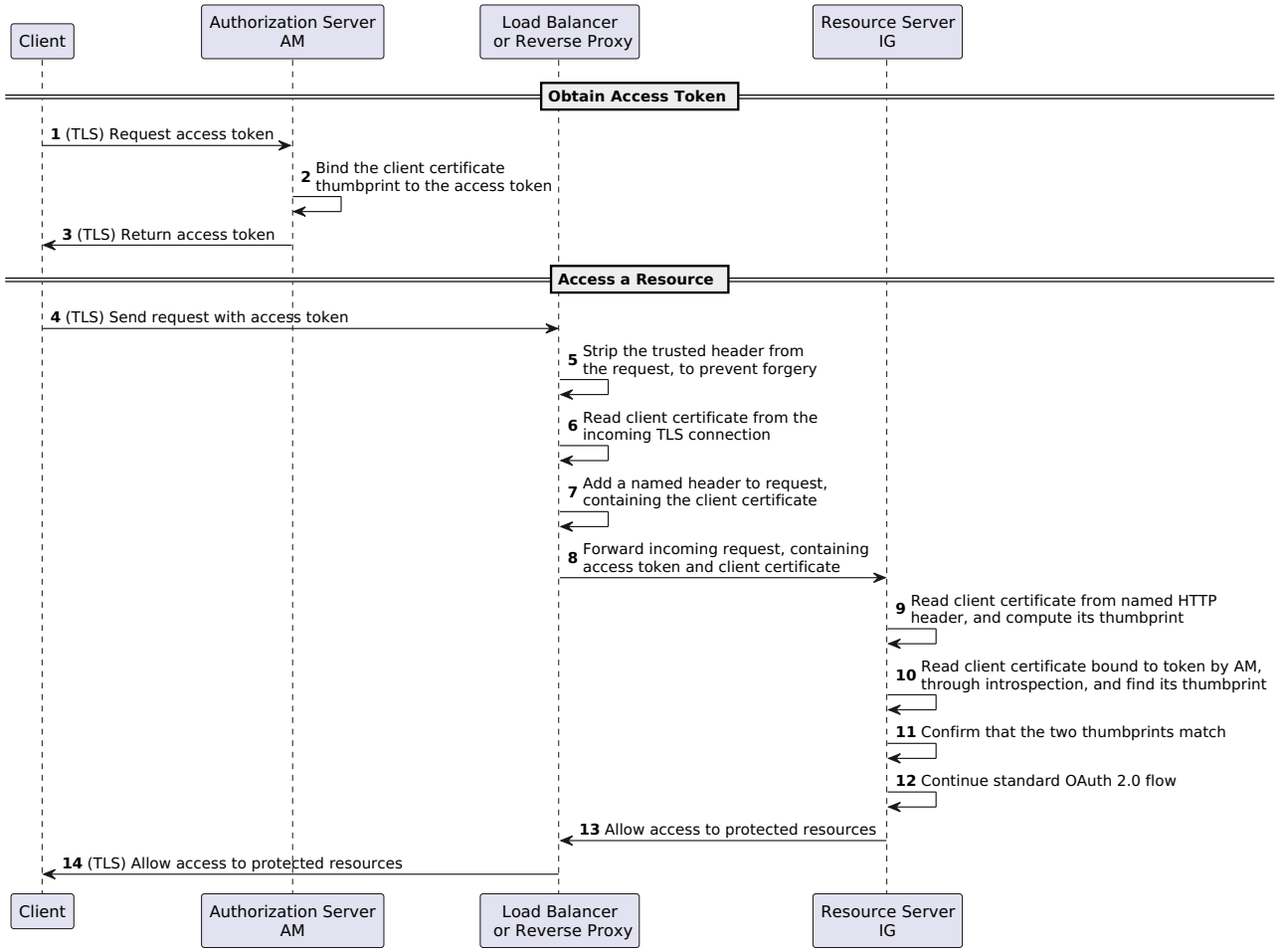
- Forward the encoded certificate to IG in the trusted header. Encode the certificate in an HTTP-header compatible format that can convey a full certificate, so that IG can rebuild the certificate.
- Strip the trusted header from incoming requests, and change the default header name to something an attacker can't guess.

Because there is a trust relationship between IG and the TLS termination point, IG doesn't authenticate the contents of the trusted header. IG accepts any value in a header from a trusted TLS termination point.

Use this example when the IG instance is running behind a load balancer or other ingress point. If the IG instance is running behind the TLS termination point, consider the example in mTLS Using Standard TLS Client Certificate Authentication.

The following image illustrates the connections and certificates required by the example:





Set up mTLS using trusted headers

1. Set up the keystores, truststores, AM, and IG as described in mTLS Using Standard TLS Client Certificate Authentication.
2. Base64-encode the value of `$oauth2_client_keystore_directory/client.cert.pem`. The value is used in the final POST.
3. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/mtls-header.json
```

Windows

```
%appdata%\OpenIG\config\routes\mtls-header.json
```

```
{
  "name": "mtls-header",
  "condition": "${find(request.uri.path, '/mtls-header')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [
        {
          "name": "CertificateThumbprintFilter-1",
          "type": "CertificateThumbprintFilter",
          "config": {
            "certificate": "${pemCertificate(decodeBase64(request.headers['ssl_client_cert'][0]))}",
            "failureHandler": {
              "type": "ScriptableHandler",
              "config": {
                "type": "application/x-groovy",
                "source": [
                  "def response = new Response(Status.TEAPOT);",
                  "response.entity = 'Failure in CertificateThumbprintFilter'",
                  "return response"
                ]
              }
            }
          }
        },
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "test"
            ],
            "requireHttps": false,
            "accessTokenResolver": {
              "type": "ConfirmationKeyVerifierAccessTokenResolver",
              "config": {
                "delegate": {
                  "name": "token-resolver-1",
                  "type": "TokenIntrospectionAccessTokenResolver",
                  "config": {
                    "amService": "AmService-1",
```

```
                    "providerHandler": {
                      "type": "Chain",
                      "config": {
                        "filters": [
                          {
                            "type": "HttpBasicAuthenticationClientFilter",
                            "config": {
                              "username": "ig_agent",
                              "passwordSecretId": "agent.secret.id",
                              "secretsProvider": "SystemAndEnvSecretStore-1"
                            }
                          }
                        ],
                        "handler": "ForgeRockClientHandler"
                      }
                    }
                  }
                }
              }
            }
          }
        }
      ],
      "handler": {
        "name": "StaticResponseHandler-1",
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/plain; charset=UTF-8" ]
          },
          "entity": "mTLS\n Valid token: ${contexts.oauth2.accessToken.token}\n Confirmation keys: $
{contexts.oauth2}"
        }
      }
    }
  }
}
```

Notice the following features of the route compared to `mtls-certificate.json`:

- The route matches requests to `/mtls-header`.
- The CertificateThumbprintFilter extracts a Java certificate from the trusted header, computes the SHA-256 thumbprint of that certificate, and makes the thumbprint available for the ConfirmationKeyVerifierAccessTokenResolver.

4. Test the setup:

    1. Get an access token from AM, over TLS:

```
$ mytoken=$(curl --request POST \
--cacert $am_keystore_directory/openam-server.cert.pem \
--cert $oauth2_client_keystore_directory/client.cert.pem \
--key $oauth2_client_keystore_directory/client.key.pem \
--header 'cache-control: no-cache' \
--header 'content-type: application/x-www-form-urlencoded' \
--data 'client_id=client-application&grant_type=client_credentials&scope=test' \
https://am.example.com:8445/openam/oauth2/access_token | jq -r .access_token)
```

2. Introspect the access_token on AM:

```
$ curl --request POST \
-u ig_agent:password \
--header 'content-type: application/x-www-form-urlencoded' \
--data token=${mytoken} \
http://am.example.com:8088/openam/oauth2/realms/root/introspect | jq

{
  "active": true,
  "scope": "test",
  "realm": "/",
  "client_id": "client-application",
  "user_id": "client-application",
  "token_type": "Bearer",
  "exp": 157...994,
  "sub": "(age!client-application)",
  "subname": "client-application",
  "iss": "http://am.example.com:8088/openam/oauth2",
  "cnf": {
    "x51...156": "1QG...Wgc"
  },
  "authGrantId": "lto...8vw",
  "auditTrackingId": "119...480"
}
```

The `cnf` property indicates the value of the confirmation code, as follows:

- `x5` : X509 certificate
- `t` : thumbprint
- `#` : separator
- `S256` : algorithm used to hash the raw certificate bytes

3. Access the IG route to validate the confirmation key, using the base64-encoded value of `$oauth2_client_keystore_directory/client.cert.pem` :

```
$ curl --request POST \
--header "authorization:Bearer $mytoken" \
--header 'ssl_client_cert:base64-encoded-cert'
http://ig.example.com:8080/mtls-header

Valid token: zw5...Sj1
  Confirmation keys: {
  ...
  }
```

The validated token and confirmation keys are displayed.

Use the OAuth 2.0 context to log in to the sample application

This section contains an example route that retrieves scopes from a token introspection, assigns them as the IG session username and password, and uses them to log the user directly in to the sample application.

For information about the context, refer to OAuth2Context.

Before you start, set up and test the example in Validate access tokens through the introspection endpoint.

1. Set up AM:
    1. Select **Identities**, and change the email address of the demo user to `demo`.
    2. Select **Scripts** > **OAuth2 Access Token Modification Script**, and replace the default script as follows:

```
import org.forgerock.http.protocol.Request
import org.forgerock.http.protocol.Response
import com.iplanet.sso.SSOException
import groovy.json.JsonSlurper

def attributes = identity.getAttributes(["mail"].toSet())
accessToken.setField("mail", attributes["mail"][0])
accessToken.setField("password", "Ch4ng31t")
```

    The AM script adds user profile information to the access token, and adds a `password` field with the value `Ch4ng31t`.

    > **Warning**
    >
    > Don't use this example in production. If the token is stateless and unencrypted, the password value is easily accessible when you have the token.

2. Set up IG:
    1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/rs-pwreplay.json
```

Windows

```
%appdata%\OpenIG\config\routes\rs-pwreplay.json
```

```
{
  "name" : "rs-pwreplay",
  "baseURI" : "http://app.example.com:8081",
  "condition" : "${find(request.uri.path, '^/rs-pwreplay')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler" : {
    "type" : "Chain",
    "config" : {
      "filters" : [
        {
          "name" : "OAuth2ResourceServerFilter-1",
          "type" : "OAuth2ResourceServerFilter",
          "config" : {
            "scopes" : [ "mail", "employeenumber" ],
            "requireHttps" : false,
            "realm" : "OpenIG",
            "accessTokenResolver": {
              "name": "TokenIntrospectionAccessTokenResolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        },
        {
          "type": "AssignmentFilter",
```

```
            "config": {
              "onRequest": [{
                "target": "${session.username}",
                "value": "${contexts.oauth2.accessToken.info.mail}"
              },
                {
                  "target": "${session.password}",
                  "value": "${contexts.oauth2.accessToken.info.password}"
                }
              ]
            }
          },
          {
            "type": "StaticRequestFilter",
            "config": {
              "method": "POST",
              "uri": "http://app.example.com:8081/login",
              "form": {
                "username": [
                  "${session.username}"
                ],
                "password": [
                  "${session.password}"
                ]
              }
            }
          }
        ],
        "handler": "ReverseProxyHandler"
      }
    }
}
```

Notice the following features of the route compared to `rs-introspect.json`:

- The route matches requests to `/rs-pwreplay`.
- The AssignmentFilter accesses the context, and injects the username and password into the SessionContext, `${Session}`.
- The StaticRequestFilter retrieves the username and password from `session`, and replaces the original HTTP GET request with an HTTP POST login request that contains the credentials to authenticate.

3. Test the setup:

   1. In a terminal window, use a `curl` command similar to the following to retrieve an access token:

      ```
      $ mytoken=$(curl -s \
      --user "client-application:password" \
      --data "grant_type=password&username=demo&password=Ch4ng31t&scope=mail%20employeenumber" \
      http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
      ```

   2. Validate the access token returned in the previous step:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/rs-pwreplay
```

HTML for the sample application is displayed.

Cache access tokens

This section builds on the example in Validate access tokens through the introspection endpoint to cache and then revoke access tokens.

When the access token **is not** cached, IG calls AM to validate the access token. When the access token **is** cached, IG doesn't validate the access token with AM.

When an access token is revoked on AM, the CacheAccessTokenResolver can delete the token from the cache when both of the following conditions are true:

- The `notification` property of AmService is enabled.
- The delegate AccessTokenResolver provides the token metadata required to update the cache.

When a refresh_token is revoked on AM, all associated access tokens are automatically and immediately revoked.

Before you start, set up and test the example in Validate access tokens through the introspection endpoint.

1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/rs-introspect-cache.json
```

Windows

```
%appdata%\OpenIG\config\routes\rs-introspect-cache.json
```

```
{
  "name": "rs-introspect-cache",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/rs-introspect-cache$')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "agent" : {
          "username" : "ig_agent",
          "passwordSecretId" : "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "CacheAccessTokenResolver-1",
              "type": "CacheAccessTokenResolver",
              "config": {
                "enabled": true,
                "defaultTimeout ": "1 hour",
                "maximumTimeToCache": "1 day",
                "amService":"AmService-1",
                "delegate": {
                  "name": "TokenIntrospectionAccessTokenResolver-1",
                  "type": "TokenIntrospectionAccessTokenResolver",
                  "config": {
                    "amService": "AmService-1",
                    "providerHandler": {
                      "type": "Chain",
                      "config": {
                        "filters": [
                          {
                            "type": "HttpBasicAuthenticationClientFilter",
                            "config": {
                              "username": "ig_agent",
                              "passwordSecretId": "agent.secret.id",
                              "secretsProvider": "SystemAndEnvSecretStore-1"
```

```
                            }
                          }
                        ],
                        "handler": {
                          "type": "Delegate",
                          "capture": "all",
                          "config": {
                            "delegate": "ForgeRockClientHandler"
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        ],
        "handler": {
          "type": "StaticResponseHandler",
          "config": {
            "status": 200,
            "headers": {
              "Content-Type": [ "text/html; charset=UTF-8" ]
            },
            "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></body></
html>"
          }
        }
      }
    }
}
```

Notice the following features of the route compared to `rs-introspect.json`, in Validate access tokens through the introspection endpoint:

- The OAuth2ResourceServerFilter uses a CacheAccessTokenResolver to cache the access token, and then delegate token resolution to the TokenIntrospectionAccessTokenResolver.
- The `amService` property in CacheAccessTokenResolver enables WebSocket notifications from AM, for events such as token revocation.
- The TokenIntrospectionAccessTokenResolver uses a ForgeRockClientHandler and a capture decorator to capture IG's interactions with AM.

2. Test token caching:

   1. In a terminal window, use a `curl` command similar to the following to retrieve an access token:

      ```
      $ mytoken=$(curl -s \
      --user "client-application:password" \
      --data "grant_type=password&username=demo&password=Ch4ng31t&scope=mail%20employeenumber" \
      http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
      ```

   2. Access the route, using the access token returned in the previous step:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/rs-introspect-cache

{
 active = true,
 scope = employeenumber mail,
 client_id = client-application,
 user_id = demo,
 token_type = Bearer,
 exp = 158...907,
 ...
}
```

3. In the route log, note that IG calls AM to introspect the access token:

```
POST http://am.example.com:8088/openam/oauth2/realms/root/introspect HTTP/1.1
```

4. Access the route again. In the route log note that this time IG doesn't call AM, because the token is cached.
5. Disable the cache and repeat the previous steps to cause IG to call AM to validate the access token for each request.

3. Test token revocation:

1. In a terminal window, use a `curl` command similar to the following to revoke the access token obtained in the previous step:

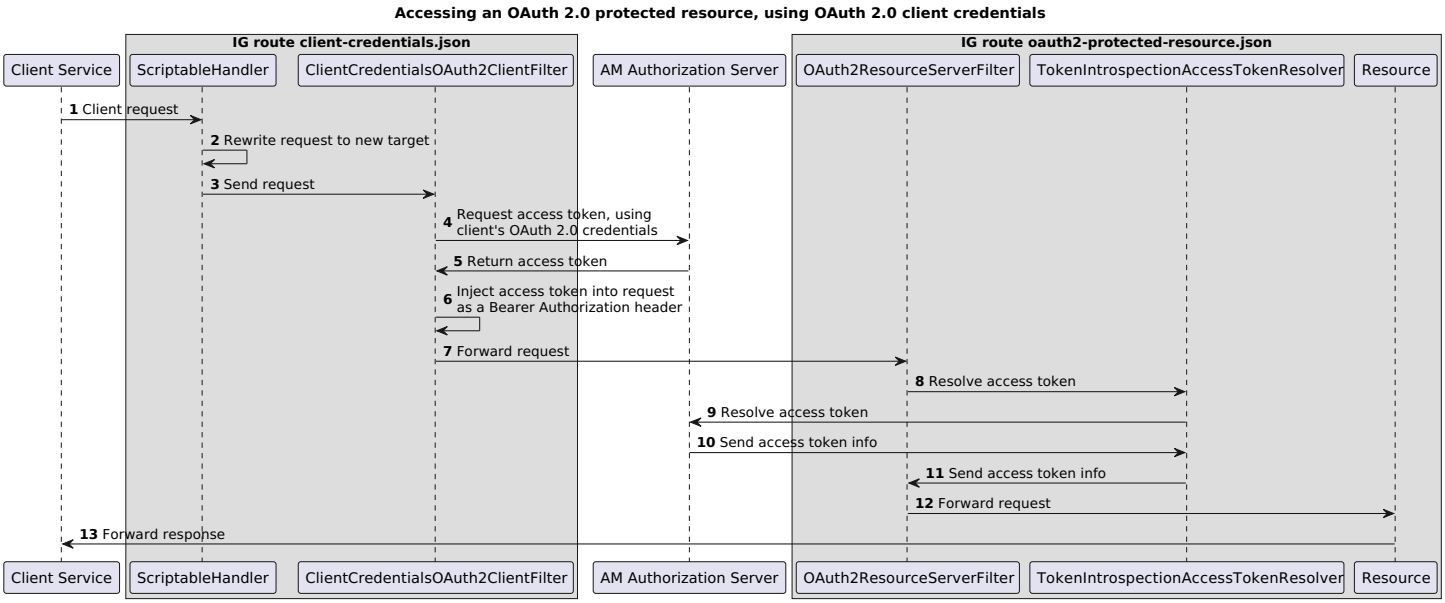
```
$ curl --request POST \
--data "token=${mytoken}" \
--data "client_id=client-application" \
--data "client_secret=password" \
"http://am.example.com:8088/openam/oauth2/realms/root/token/revoke"
```

2. Access the route using the access token and and note that the request isn't authorized because the token is revoked:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/rs-introspect-cache

...
HTTP/1.1 401 Unauthorized
```

Use OAuth 2.0 client credentials

This example shows how a client service accesses an OAuth 2.0-protected resource by using its OAuth 2.0 client credentials.



1. Set up the AM as an Authorization Server:

    1. Register an IG agent with the following values, as described in Register an IG agent in AM:

        - **Agent ID**: `ig_agent`
        - **Password**: `password`
        - **Token Introspection**: `Realm Only`

        > **Important**
        >
        > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

    2. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

        > **Important**
        >
        > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

    3. Create an OAuth 2.0 Authorization Server:

        1. Select **Services** > **Add a Service** > **OAuth2 Provider**.
        2. Add a service with the default values.

    4. Create an OAuth 2.0 client to request access tokens, using client credentials for authentication:

        1. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:

            - **Client ID** : `client-service`
            - **Client secret** : `password`

- **Scope(s)** : `client-scope`

2. On the **Advanced** tab, select the following value:

   - **Grant Types** : `Client Credentials`

2. Set up IG:

   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

   2. Set an environment variable for the IG agent password, and then restart IG:

      ```
      $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
      ```

      The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   3. Add the following route to IG:

      Linux

      ```
      $HOME/.openig/config/routes/oauth2-protected-resource.json
      ```

      Windows

      ```
      %appdata%\OpenIG\config\routes\oauth2-protected-resource.json
      ```

```
{
  "name": "oauth2-protected-resource",
  "condition": "${find(request.uri.path, '^/oauth2-protected-resource')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [ "client-scope" ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "TokenIntrospectionAccessTokenResolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
```

```
          "config": {
            "status": 200,
            "headers": {
              "Content-Type": [ "text/html; charset=UTF-8" ]
            },
            "entity": "<html><body><h2>Access Granted</h2></body></html>"
          }
        }
      }
    }
  }
```

Notice the following features of the route:

- The route matches requests to `/oauth2-protected-resource`.
- The `OAuth2ResourceServerFilter` expects an OAuth 2.0 access token in the header of the incoming request, with the scope `client-scope`.
- The filter uses a TokenIntrospectionAccessTokenResolver to resolve the access token. The introspect endpoint is protected with HTTP Basic Authentication, and the `providerHandler` uses an HttpBasicAuthenticationClientFilter to provide the resource server credentials.
- For convenience in this test, `"requireHttps"` is false. In production environments, set it to true.
- After the filter successfully validates the access token, it creates a new context from the Authorization Server response, containing information about the access token.
- The StaticResponseHandler returns a message that access is granted.

4. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/client-credentials.json
```

Windows

```
%appdata%\OpenIG\config\routes\client-credentials.json
```

```
{
  "name": "client-credentials",
  "baseURI": "http://ig.example.com:8080",
  "condition" : "${find(request.uri.path, '^/client-credentials')}",
  "heap" : [ {
    "name" : "clientSecretAccessTokenExchangeHandler",
    "type" : "Chain",
    "capture" : "all",
    "config" : {
      "filters" : [ {
        "type" : "ClientSecretBasicAuthenticationFilter",
        "config" : {
          "clientId" : "client-service",
          "clientSecretId" : "client.secret.id",
          "secretsProvider" : {
            "type" : "Base64EncodedSecretStore",
            "config" : {
              "secrets" : {
                "client.secret.id" : "cGFzc3dvcmQ="
              }
            }
          }
        }
      } ],
      "handler" : "ForgeRockClientHandler"
    }
  }, {
    "name" : "oauth2EnabledClientHandler",
    "type" : "Chain",
    "capture" : "all",
    "config" : {
      "filters" : [ {
        "type" : "ClientCredentialsOAuth2ClientFilter",
        "config" : {
          "tokenEndpoint" : "http://am.example.com:8088/openam/oauth2/access_token",
          "endpointHandler": "clientSecretAccessTokenExchangeHandler",
          "scopes" : [ "client-scope" ]
        }
      } ],
      "handler" : "ForgeRockClientHandler"
    }
  } ],
  "handler" : {
    "type" : "ScriptableHandler",
    "config" : {
      "type" : "application/x-groovy",
      "clientHandler" : "oauth2EnabledClientHandler",
      "source" : [ "request.uri.path = '/oauth2-protected-resource'", "return http.send(context,
request);" ]
    }
  }
}
```

Note the following features of the route:

- The route matches requests to `/client-credentials`.

- The ScriptableHandler rewrites the request to target it to `/oauth2-protected-resource`, and then calls the HTTP client, that has been redefined to use the oauth2EnabledClientHandler.
- The oauth2EnabledClientHandler calls the ClientCredentialsOAuth2ClientFilter to obtain an access token from AM.
- The ClientCredentialsOAuth2ClientFilter calls the clientSecretAccessTokenExchangeHandler to exchange tokens on the authorization endpoint.
- The clientSecretAccessTokenExchangeHandler calls a ClientSecretBasicAuthenticationFilter to authenticate the client through the HTTP basic access authentication scheme, and a ForgeRockClientHandler to propagate the request.
- The route `oauth2-protected-resource.json` uses the AM introspection endpoint to resolve the access token and display its contents.

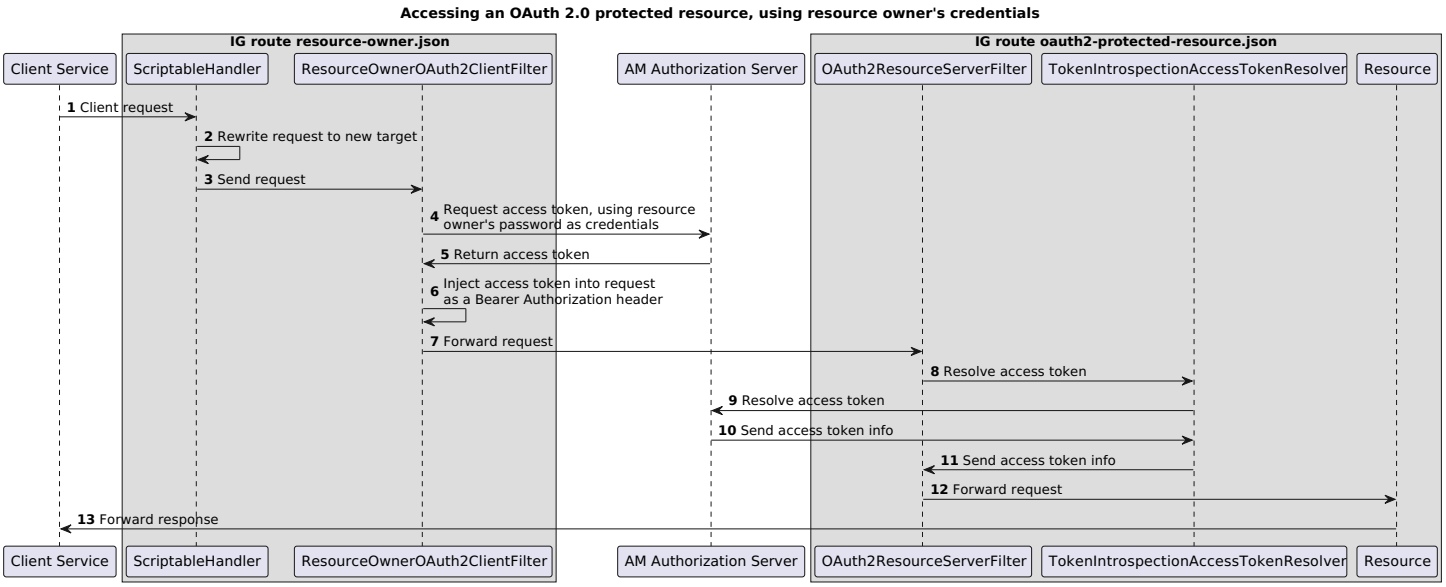
3. Test the setup:

   1. In your browser's privacy or incognito mode, go to to https://ig.example.com:8443/client-credentials.
   2. If you see warnings that the site isn't secure, respond to the warnings to access the site.

      A message shows that access is granted.

Use OAuth 2.0 resource owner password credentials

This example shows how a client service accesses an OAuth 2.0-protected resource by using resource owner password credentials.



> **Important**
>
> This procedure uses the *Resource Owner Password Credentials* grant type. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize other grant types whenever possible.

1. Set up the AM as an Authorization Server:
    1. Register an IG agent with the following values, as described in Register an IG agent in AM:
        - **Agent ID**: `ig_agent`
        - **Password**: `password`
        - **Token Introspection**: `Realm Only`

        > **Important**
        >
        > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

    2. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

        > **Important**
        >
        > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

    3. Create an OAuth 2.0 Authorization Server:
        1. Select **Services** > **Add a Service** > **OAuth2 Provider**.
        2. Add a service with the default values.
    4. Create an OAuth 2.0 client to request access tokens, using the resource owner's password for authentication:
        1. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:
            - **Client ID** : `resource-owner-client`
            - **Client secret** : `password`
            - **Scope(s)** : `client-scope`
        2. On the **Advanced** tab, select the following value:
            - **Grant Types** : `Resource Owner Password Credentials`
2. Set up IG:
    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
    2. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

3. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/oauth2-protected-resource.json
```

Windows

```
%appdata%\OpenIG\config\routes\oauth2-protected-resource.json
```

```
{
  "name": "oauth2-protected-resource",
  "condition": "${find(request.uri.path, '^/oauth2-protected-resource')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [ "client-scope" ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "TokenIntrospectionAccessTokenResolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
```

```
          "config": {
            "status": 200,
            "headers": {
              "Content-Type": [ "text/html; charset=UTF-8" ]
            },
            "entity": "<html><body><h2>Access Granted</h2></body></html>"
          }
        }
      }
    }
  }
```

Notice the following features of the route:

- The route matches requests to `/oauth2-protected-resource`.
- The `OAuth2ResourceServerFilter` expects an OAuth 2.0 access token in the header of the incoming request, with the scope `client-scope`.
- The filter uses a TokenIntrospectionAccessTokenResolver to resolve the access token. The introspect endpoint is protected with HTTP Basic Authentication, and the `providerHandler` uses an HttpBasicAuthenticationClientFilter to provide the resource server credentials.
- For convenience in this test, `"requireHttps"` is false. In production environments, set it to true.
- After the filter successfully validates the access token, it creates a new context from the Authorization Server response, containing information about the access token.
- The StaticResponseHandler returns a message that access is granted.

4. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/resource-owner.json
```

Windows

```
%appdata%\OpenIG\config\routes\resource-owner.json
```

```
{
  "name": "resource-owner",
  "baseURI": "http://ig.example.com:8080",
  "condition" : "${find(request.uri.path, '^/resource-owner')}",
  "heap" : [ {
    "name" : "clientSecretAccessTokenExchangeHandler",
    "type" : "Chain",
    "capture" : "all",
    "config" : {
      "filters" : [ {
        "type" : "ClientSecretBasicAuthenticationFilter",
        "config" : {
          "clientId" : "resource-owner-client",
          "clientSecretId" : "client.secret.id",
          "secretsProvider" : {
            "type" : "Base64EncodedSecretStore",
            "config" : {
              "secrets" : {
                "client.secret.id" : "cGFzc3dvcmQ="
              }
            }
          }
        }
      } ],
      "handler" : "ForgeRockClientHandler"
    }
  }, {
    "name" : "oauth2EnabledClientHandler",
    "type" : "Chain",
    "capture" : "all",
    "config" : {
      "filters" : [ {
        "type" : "ResourceOwnerOAuth2ClientFilter",
        "config" : {
          "tokenEndpoint" : "http://am.example.com:8088/openam/oauth2/access_token",
          "endpointHandler": "clientSecretAccessTokenExchangeHandler",
          "scopes" : [ "client-scope" ],
          "username" : "demo",
          "passwordSecretId" : "user.password.secret.id",
          "secretsProvider" : {
            "type" : "Base64EncodedSecretStore",
            "config" : {
              "secrets" : {
                "user.password.secret.id" : "Q2g0bmczMXQ="
              }
            }
          }
        }
      } ],
      "handler" : "ForgeRockClientHandler"
    }
  } ],
  "handler" : {
    "type" : "ScriptableHandler",
    "config" : {
      "type" : "application/x-groovy",
      "clientHandler" : "oauth2EnabledClientHandler",
```

```
        "source" : [ "request.uri.path = '/oauth2-protected-resource'", "return http.send(context,
request);" ]
      }
    }
}
```

Note the following features of the route:

- The route matches requests to `/resource-owner`.
- The ScriptableHandler rewrites the request to target it to `/oauth2-protected-resource`, and then calls the HTTP client, that has been redefined to use the oauth2EnabledClientHandler.
- The oauth2EnabledClientHandler calls the ResourceOwnerOAuth2ClientFilter to obtain an access token from AM.
- The ResourceOwnerOAuth2ClientFilter calls the clientSecretAccessTokenExchangeHandler to exchange tokens on the authorization endpoint. The demo user authenticates with their username and password.
- The clientSecretAccessTokenExchangeHandler calls a ClientSecretBasicAuthenticationFilter to authenticate the client through the HTTP basic access authentication scheme, and a ForgeRockClientHandler to propagate the request.
- The route `oauth2-protected-resource.json` uses the AM introspection endpoint to resolve the access token and display its contents.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to to https://ig.example.com:8443/resource-owner.
   2. If you see warnings that the site isn't secure, respond to the warnings to access the site.

      A message shows that access is granted.

OpenID Connect

The following sections provide an overview of how IG supports OpenID Connect 1.0, and examples of to set up IG as an OpenID Connect relying party in different deployment scenarios:

About IG with OpenID Connect

IG supports OpenID Connect 1.0, an authentication layer built on OAuth 2.0. OpenID Connect 1.0 is a specific implementation of OAuth 2.0, where the identity provider holds the protected resource that the third-party application wants to access. For more information, refer to OpenID Connect.

OpenID Connect refers to the following entities:

- *End user* : An OAuth 2.0 resource owner whose user information the application needs to access.

  The end user wants to use an application through an existing identity provider account without signing up and creating credentials for another web service.

- *Relying Party* (RP): An OAuth 2.0 client that needs access to the end user's protected user information.

For example, an online mail application needs to know which end user is accessing the application in order to present the correct inbox.

As another example, an online shopping site needs to know which end user is accessing the site in order to present the right offerings, account, and shopping cart.

- *OpenID Provider* (OP): An OAuth 2.0 Authorization Server and also resource server that holds the user information and grants access.

  The OP requires the end user to give the RP permission to access to some of its user information. Because OpenID Connect 1.0 defines unique identification for an account (subject identifier + issuer identifier), the RP can use that identification to bind its own user profile to a remote identity.

  For the online mail application, this key could be used to access the mailboxes and related account information. For the online shopping site, this key could be used to access the offerings, account, shopping cart and others. The key makes it possible to serve users as if they had local accounts.

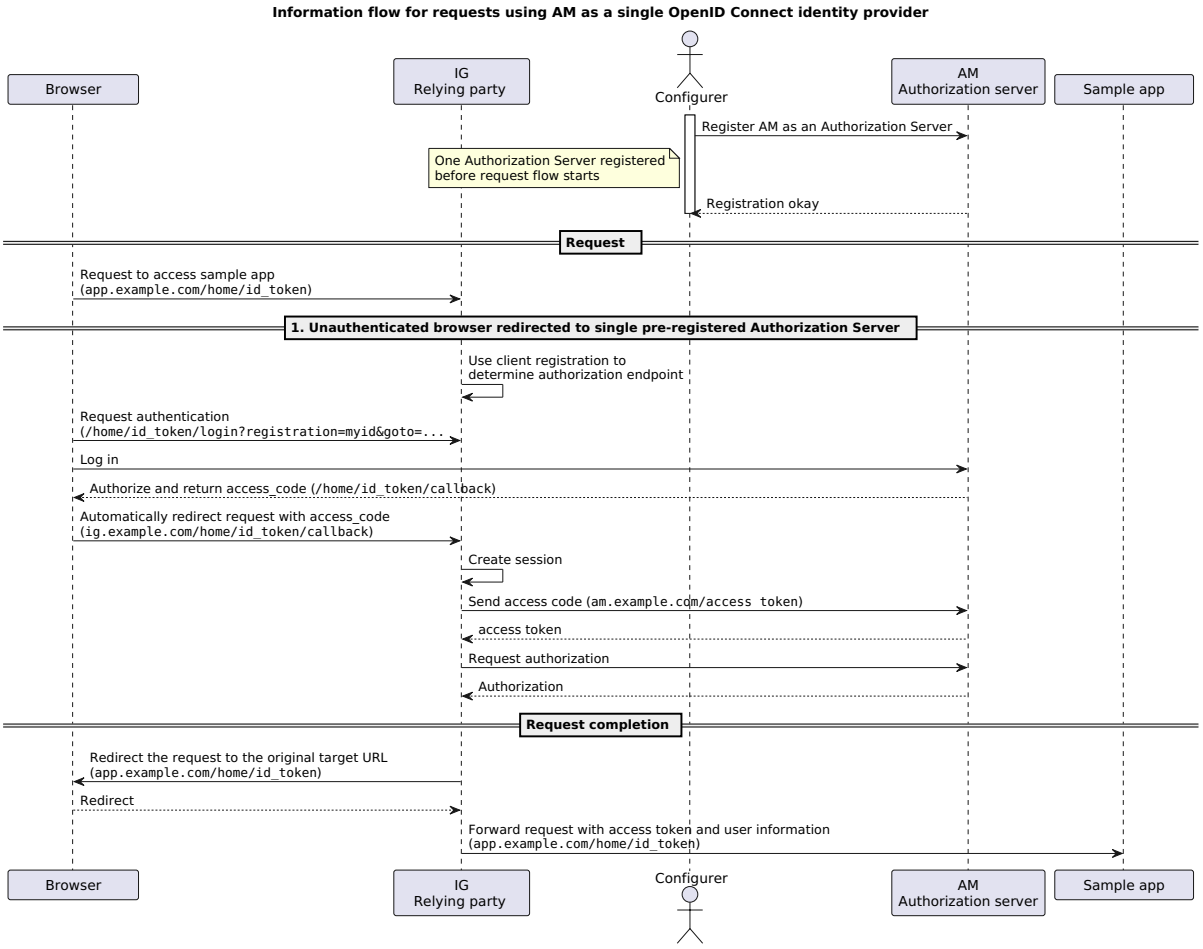
- *UserInfo* : The protected resource that the third-party application wants to access. The information about the authenticated end user is expressed in a standard format. The user-info endpoint is hosted on the Authorization Server and is protected with OAuth 2.0.

When IG acts as an OpenID Connect relying party, its role is to retrieve user information from the OpenID provider, and then to inject that information into the context for use by subsequent filters and handlers.

AM as an OpenID Connect provider

This section gives an example of how to set up AM as an OpenID Connect identity provider, and IG as a relying party for browser requests to the home page of the sample application.

The following sequence diagram shows the flow of information for a request to access the home page of the sample application, using AM as a single, preregistered OpenID Connect identity provider, and IG as a relying party:



Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

1. Set Up AM as an OpenID Connect provider:
    1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
        - `https://ig.example.com:8443/*`
        - `https://ig.example.com:8443/*?*`
    2. Create an OAuth 2.0 Authorization Server:
        1. Select **Services** > **Add a Service** > **OAuth2 Provider**.
        2. Add a service with the default values.
    3. Create an OAuth 2.0 Client to request OAuth 2.0 access tokens:
        1. Select **Applications** > **OAuth 2.0** > **Clients**.
        2. Add a client with the following values:
            - **Client ID**: `oidc_client`
            - **Client secret**: `password`
            - **Redirection URIs**: `https://ig.example.com:8443/home/id_token/callback`

- **Scope(s)**: `openid`, `profile`, and `email`

3. On the **Advanced** tab, select the following values:
   - **Grant Types**: `Authorization Code`
4. On the **Signing and Encryption** tab, change **ID Token Signing Algorithm** to `HS256`, `HS384`, or `HS512`. The algorithm must be HMAC.

2. Set up IG:
   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
   2. Set an environment variable for `oidc_client`, and then restart IG:

```
$ export OIDC_SECRET_ID='cGFzc3dvcmQ='
```

   3. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

   4. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/07-openid.json
```

Windows

```
%appdata%\OpenIG\config\routes\07-openid.json
```

```
{
  "name": "07-openid",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/id_token')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AuthenticatedRegistrationHandler-1",
      "type": "Chain",
      "config": {
        "filters": [
          {
            "name": "ClientSecretBasicAuthenticationFilter-1",
            "type": "ClientSecretBasicAuthenticationFilter",
            "config": {
              "clientId": "oidc_client",
              "clientSecretId": "oidc.secret.id",
              "secretsProvider": "SystemAndEnvSecretStore-1"
            }
          }
        ],
        "handler": "ForgeRockClientHandler"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "AuthorizationCodeOAuth2ClientFilter-1",
          "type": "AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/home/id_token",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "status": 500,
                "headers": {
                  "Content-Type": [
                    "text/plain"
                  ]
                },
                "entity": "Error in OAuth 2.0 setup."
              }
            },
            "registrations": [
              {
                "name": "oidc-user-info-client",
                "type": "ClientRegistration",
                "config": {
                  "clientId": "oidc_client",
                  "issuer": {
                    "name": "Issuer",
                    "type": "Issuer",
                    "config": {
                      "wellKnownEndpoint": "http://am.example.com:8088/openam/oauth2/.well-known/
```

```
openid-configuration"
                    }
                  },
                  "scopes": [
                    "openid",
                    "profile",
                    "email"
                  ],
                  "authenticatedRegistrationHandler": "AuthenticatedRegistrationHandler-1"
                }
              }
            ],
            "requireHttps": false,
            "cacheExpiration": "disabled"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the IG route in Studio, see OpenID Connect in Structured Editor.

Notice the following features about the route:

- The route matches requests to `/home/id_token`.
- The `AuthorizationCodeOAuth2ClientFilter` enables IG to act as a relying party. It uses a single client registration that is defined inline and refers to the AM server configured in AM as a single OpenID Connect provider.
- The filter has a base client endpoint of `/home/id_token`, which creates the following service URIs:
  - Requests to `/home/id_token/login` start the delegated authorization process.
  - Requests to `/home/id_token/callback` are expected as redirects from the OAuth 2.0 Authorization Server (OpenID Connect provider). This is why the redirect URI in the client profile in AM is set to `https://ig.example.com:8443/home/id_token/callback`.
  - Requests to `/home/id_token/logout` remove the authorization state for the end user, and redirect to the specified URL if a `goto` parameter is provided.

    These endpoints are implicitly reserved. Attempts to access them directly can cause undefined errors.
- For convenience in this test, `"requireHttps"` is false. In production environments, set it to true. So that you see the delegated authorization process when you make a request, `"requireLogin"` has the default value `true`.
- The target for storing authorization state information is `${attributes.openid}`. This is where subsequent filters and handlers can find access tokens and user information.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/id_token.

The AM login page is displayed.

2. Log in to AM as user `demo`, password `Ch4ng31t`, and then allow the application to access user information.

   The home page of the sample application is displayed.

Authenticate automatically to the sample application

To authenticate automatically to the sample application, change the last name of the user `demo` to match the password `Ch4ng31t`, and add a StaticRequestFilter like the following to the end of the chain in `07-openid.json`:

```
{
  "type": "StaticRequestFilter",
  "config": {
    "method": "POST",
    "uri": "http://app.example.com:8081/login",
    "form": {
      "username": [
        "${attributes.openid.user_info.sub}"
      ],
      "password": [
        "${attributes.openid.user_info.family_name}"
      ]
    }
  }
}
```

The StaticRequestFilter retrieves the username and password from the context, and replaces the original HTTP GET request with an HTTP POST login request containing credentials.

Identity Cloud as an OpenID Connect provider

This example sets up ForgeRock Identity Cloud as an OpenID Connect identity provider, and Identity Gateway as a relying party.

For more information about Identity Gateway and OpenID Connect, refer to OpenID Connect.

Before you start, prepare Identity Cloud, IG, and the sample application as described in Example installation for this guide.

1. Set up Identity Cloud:

   1. Log in to the Identity Cloud admin UI as an administrator.
   2. Make sure you are managing the `alpha` realm. If not, click the current realm at the top of the screen, and switch realm.
   3. Go to **Identities** > **Manage** > **Alpha realm - Users**, and add a user with the following values:

      - **Username**: `demo`
      - **First name**: `demo`
      - **Last name**: `user`
      - **Email Address**: `demo@example.com`

- **Password**: `Ch4ng3!t`

4. Go to **Applications** > **Custom Application** > **OIDC - OpenId Connect** > **Web** and add a web application with the following values:
   - **Name**: `oidc_client`
   - **Owners**: `demo user`
   - **Client Secret**: `password`
   - **Sign On** > **Sign-in URLs**: `https://ig.example.com:8443/home/id_token/callback`
   - **Sign On** > **Grant Types**: `Authorization Code`
   - **Sign On** > **Scopes**: `openid`, `profile`, `email`
   - **Show advanced settings** > **Authentication** > **Implied Consent**: `On`

For more information, refer to Identity Cloud's Application management.

2. Set up Identity Gateway:

1. Set an environment variable for the `oidc_client` password, and then restart IG:

```
$ export OIDC_SECRET_ID='cGFzc3dvcmQ='
```

1. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

2. Add the following route to Identity Gateway, replacing the value for the property `amInstanceUrl`:

Linux

```
$HOME/.openig/config/routes/oidc-idc.json
```

Windows

```
%appdata%\OpenIG\config\routes\oidc-idc.json
```

```
{
  "name": "oidc-idc",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/id_token')}",
  "properties": {
    "amInstanceUrl": "https://myTenant.forgeblocks.com/am"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AuthenticatedRegistrationHandler-1",
      "type": "Chain",
      "config": {
        "filters": [
          {
            "name": "ClientSecretBasicAuthenticationFilter-1",
            "type": "ClientSecretBasicAuthenticationFilter",
            "config": {
              "clientId": "oidc_client",
              "clientSecretId": "oidc.secret.id",
              "secretsProvider": "SystemAndEnvSecretStore-1"
            }
          }
        ],
        "handler": "ForgeRockClientHandler"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "AuthorizationCodeOAuth2ClientFilter-1",
          "type": "AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/home/id_token",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "status": 500,
                "headers": {
                  "Content-Type": [
                    "text/plain"
                  ]
                },
                "entity": "Error in OAuth 2.0 setup."
              }
            },
            "registrations": [
              {
                "name": "oauth2-client",
                "type": "ClientRegistration",
                "config": {
                  "clientId": "oidc_client",
                  "issuer": {
                    "name": "Issuer",
```

```
                    "type": "Issuer",
                    "config": {
                      "wellKnownEndpoint": "&{amInstanceUrl}/oauth2/realms/alpha/.well-known/openid-
configuration"
                    }
                  },
                  "scopes": [
                    "openid",
                    "profile",
                    "email"
                  ],
                  "authenticatedRegistrationHandler": "AuthenticatedRegistrationHandler-1"
                }
              }
            ],
            "requireHttps": false,
            "cacheExpiration": "disabled"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Compared to `07-openid.json` in AM as a single OpenID Connect provider, where Access Management is running locally, the ClientRegistration `wellKnownEndpoint` points to Identity Cloud.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/id_token.

      The Identity Cloud login page is displayed.

   2. Log in to Identity Cloud as user `demo`, password `Ch4ng3!t`. The home page of the sample application is displayed.

PingOne as an OpenID Connect provider

This example sets up PingOne as an OpenID Connect identity provider and Identity Gateway as a relying party.

Before you start, prepare IG and the sample application as described in the Quick install.

1. Set up the PingOne environment:

   1. Create a PingOne test environment.

   2. Add a PingOne test user.

2. Create a PingOne OIDC web application.

   Learn more from PingOne's Creating a web application.

   1. In the test environment, create a web application with the following values:

      - **Application Name**: `oidc_client`
      - **Description**: `OIDC client`

- **Application Type**: `OIDC Web App`

2. In the application, select the **Overview** panel and click **Protocol OpenID Connect**.
3. In the **Redirect URIs** field, add `https://ig.example.com:8443/home/id_token/callback` and then save the application.

   Learn more from PingOne's Editing an application - OIDC.
4. At the top-right of the page, click the slider to enable the application.
5. Go to the **Configuration** panel and make a note of the following values in the **URLs** drop-down list:

   - **OIDC Discovery Endpoint**
   - **Client ID**
   - **Client Secret**

     The values are used in the IG setup.

3. Set up IG:

   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
   2. Add the following route to IG to serve the sample application .css and other static resources:

      Linux

      ```
      $HOME/.openig/config/routes/00-static-resources.json
      ```

      Windows

      ```
      %appdata%\OpenIG\config\routes\00-static-resources.json
      ```

      ```
      {
        "name" : "00-static-resources",
        "baseURI" : "http://app.example.com:8081",
        "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
      or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
        "handler": "ReverseProxyHandler"
      }
      ```

   3. Base64-encode the **Client Secret** for the web application created in the previous step, and then set the value as an environment variable:

```
$ export OIDC_SECRET_ID='Yy5...A=='
```

4. Add the following route to IG, replacing the values of the following properties with values for the web application created in the previous step:

   - OIDC_Discovery_Endpoint: OIDC Discovery Endpoint
   - Client_ID: **Client ID**

Linux

```
$HOME/.openig/config/routes/oidc-ping.json
```

Windows

```
%appdata%\OpenIG\config\routes\oidc-ping.json
```

```
{
  "name": "oidc-ping",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/id_token')}",
  "properties": {
    "OIDC_Discovery_Endpoint": "OIDC Discovery endpoint of the web app",
    "Client_ID": "Client ID of the web app"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AuthenticatedRegistrationHandler-1",
      "type": "Chain",
      "config": {
        "filters": [
          {
            "name": "ClientSecretBasicAuthenticationFilter-1",
            "type": "ClientSecretBasicAuthenticationFilter",
            "config": {
              "clientId": "&{Client_ID}",
              "clientSecretId": "oidc.secret.id",
              "secretsProvider": "SystemAndEnvSecretStore-1"
            }
          }
        ],
        "handler": "ForgeRockClientHandler"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "AuthorizationCodeOAuth2ClientFilter-1",
          "type": "AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/home/id_token",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "status": 500,
                "headers": {
                  "Content-Type": [ "text/html; charset=UTF-8" ]
                },
                "entity": "<html><body>Error in OAuth 2.0 setup.<br> $
{contexts.oauth2Failure.exception.message}</body></html>"
              }
            },
            "registrations": [
              {
                "name": "oauth2-client",
                "type": "ClientRegistration",
                "config": {
                  "clientId": "${Client_ID}",
                  "issuer": {
                    "name": "PingOne",
```

```
                    "type": "Issuer",
                    "config": {
                      "wellKnownEndpoint": "&{OIDC_Discovery_Endpoint}"
                    }
                  },
                  "scopes": [
                    "openid"
                  ],
                  "authenticatedRegistrationHandler": "AuthenticatedRegistrationHandler-1"
                }
              }
            ],
            "requireHttps": false,
            "cacheExpiration": "disabled"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

5. Restart IG.

4. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/id_token.

      The PingOne login page is displayed.

   2. Log in to PingOne as user `demo`, password `Ch4ng3!t`.

   3. If prompted by PingOne, change the password of the demo user.

      The home page of the sample application is displayed.

Multiple OpenID Connect providers

This section gives an example of using OpenID Connect with two identity providers.

Client registrations for an AM provider and Identity Cloud provider are declared in the heap. The Nascar page helps the user to choose an identity provider.

1. Set up AM as the first identity provider, as described in AM as a single OpenID Connect provider.

2. Set up Identity Cloud as a second identity provider, as described in Identity Cloud as an OpenID Connect provider.

3. Add the following route to IG, replacing the value for the property `amInstanceUrl`:

Linux

```
$HOME/.openig/config/routes/07-openid-nascar.json
```

Windows

```
%appdata%\OpenIG\config\routes\07-openid-nascar.json
```

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AuthenticatedRegistrationHandler-1",
      "type": "Chain",
      "config": {
        "filters": [
          {
            "name": "ClientSecretBasicAuthenticationFilter-1",
            "type": "ClientSecretBasicAuthenticationFilter",
            "config": {
              "clientId": "oidc_client",
              "clientSecretId": "oidc.secret.id",
              "secretsProvider": "SystemAndEnvSecretStore-1"
            }
          }
        ],
        "handler": "ForgeRockClientHandler"
      }
    },
    {
      "name": "openam",
      "type": "ClientRegistration",
      "config": {
        "clientId": "oidc_client",
        "issuer": {
          "name": "am_issuer",
          "type": "Issuer",
          "config": {
            "wellKnownEndpoint": "http://am.example.com:8088/openam/oauth2/.well-known/openid-configuration"
          }
        },
        "scopes": [
          "openid",
          "profile",
          "email"
        ],
        "authenticatedRegistrationHandler": "AuthenticatedRegistrationHandler-1"
      }
    },
    {
      "name": "idcloud",
      "type": "ClientRegistration",
      "config": {
        "clientId": "oidc_client",
        "issuer": {
          "name": "idc_issuer",
          "type": "Issuer",
          "config": {
            "wellKnownEndpoint": "&{amInstanceUrl}/oauth2/realms/alpha/.well-known/openid-configuration"
          }
        },
        "scopes": [
          "openid",
          "profile",
          "email"
```

```
        ],
        "authenticatedRegistrationHandler": "AuthenticatedRegistrationHandler-1"
      }
    },
    {
      "name": "NascarPage",
      "type": "StaticResponseHandler",
      "config": {
        "status": 200,
        "headers": {
          "Content-Type": [ "text/html; charset=UTF-8" ]
        },
        "entity": [
          "<html>",
          "  <body>",
          "    <p><a href='/home/id_token/login?registration=oidc_client&issuer=am_issuer&goto=${urlEncodeQueryParameterNameOrValue('https://ig.example.com:8443/home/id_token')}'>Access Management login</a></p>",
          "    <p><a href='/home/id_token/login?registration=oidc_client&issuer=idc_issuer&goto=${urlEncodeQueryParameterNameOrValue('https://ig.example.com:8443/home/id_token')}'>Identity Cloud login</a></p>",
          "  </body>",
          "</html>"
        ]
      }
    }
  ],
  "name": "07-openid-nascar",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/id_token')}",
  "properties": {
    "amInstanceUrl": "https://myTenant.forgeblocks.com/am"
  },
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/home/id_token",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "comment": "Trivial failure handler for debugging only",
                "status": 500,
                "headers": {
                  "Content-Type": [ "text/plain; charset=UTF-8" ]
                },
                "entity": "${contexts.oauth2Failure.error}: ${contexts.oauth2Failure.description}"
              }
            },
            "loginHandler": "NascarPage",
            "registrations": [ "openam", "idcloud" ],
            "requireHttps": false,
            "cacheExpiration": "disabled"
          }
        }
```

```
        ],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

Consider the differences with `07-openid.json`:

- The heap objects `openam` and `idcloud` define client registrations.
- The StaticResponseHandler provides links to the client registrations.
- The AuthorizationCodeOAuth2ClientFilter uses a `loginHandler` to allow users to choose a client registration and therefore an identity provider.

4. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/id_token.

      The Nascar page offers the choice of identity provider.

   2. Using the following credentials, select a provider, log in, and allow the application to access user information:

      - AM: user `demo`, password `Ch4ng31t`.
      - Identity Cloud: user `demo`, password `Ch4ng3!t`

      The home page of the sample application is displayed.

Discovery and dynamic registration with OpenID Connect providers

OpenID Connect defines mechanisms for discovering and dynamically registering with an identity provider that isn't known in advance, as specified in the following publications: OpenID Connect Discovery, OpenID Connect Dynamic Client Registration, and OAuth 2.0 Dynamic Client Registration Protocol.

In dynamic registration, issuer and client registrations are generated dynamically. They are held in memory and can be reused, but don't persist when IG is restarted.

This section builds on the example in AM as a single OpenID Connect provider to give an example of discovering and dynamically registering with an identity provider that isn't known in advance. In this example, the client sends a signed JWT to the Authorization Server.

To facilitate the example, a WebFinger service is embedded in the sample application. In a normal deployment, the WebFinger server is likely to be a service on the issuer's domain.

1. Set up a key

   1. Locate a directory for secrets, and go to it:

      ```
      $ cd /path/to/secrets
      ```

   2. Create a key:

```
$ keytool -genkey \
  -alias myprivatekeyalias \
  -keyalg RSA \
  -keysize 2048 \
  -keystore keystore.p12 \
  -storepass keystore \
  -storetype PKCS12 \
  -keypass keystore \
  -validity 360 \
  -dname "CN=ig.example.com, OU=example, O=com, L=fr, ST=fr, C=fr"
```

2. Set up AM:

    1. Set up AM as described in AM as a single OpenID Connect provider.
    2. Select the user `demo`, and change the last name to `Ch4ng31t`. For this example, the last name must be the same as the password.
    3. Configure the OAuth 2.0 Authorization Server for dynamic registration:
        1. Select **Services** > **OAuth2 Provider**.
        2. On the **Advanced** tab, add the following scopes to **Client Registration Scope Allowlist**: `openid`, `profile`, `email`.
        3. On the **Client Dynamic Registration** tab, select these settings:
            - **Allow Open Dynamic Client Registration**: Enabled
            - **Generate Registration Access Tokens**: Disabled
    4. Configure the authentication method for the OAuth 2.0 Client:
        1. Select **Applications** > **OAuth 2.0** > **Clients**.
        2. Select `oidc_client`, and on the **Advanced** tab, select **Token Endpoint Authentication Method**: `private_key_jwt`.

3. Set up IG:

    1. In the IG configuration, set an environment variable for the keystore password, and then restart IG:

    ```
    $ export KEYSTORE_SECRET_ID='a2V5c3RvcmU='
    ```

    The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

    2. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

3. Add the following script to IG:

Linux

```
$HOME/.openig/scripts/groovy/discovery.groovy
```

Windows

```
%appdata%\OpenIG\scripts\groovy\discovery.groovy
```

```
/*
 * OIDC discovery with the sample application
 */
response = new Response(Status.OK)
response.getHeaders().put(ContentTypeHeader.NAME, "text/html");
response.entity = """
<!doctype html>
<html>
  <head>
    <title>OpenID Connect Discovery</title>
    <meta charset='UTF-8'>
  </head>
  <body>
    <form id='form' action='/discovery/login?'>
      Enter your user ID or email address:
        <input type='text' id='discovery' name='discovery'
          placeholder='demo or demo@example.com' />
        <input type='hidden' name='goto'
          value='${contexts.idpSelectionLogin.originalUri}' />
    </form>
    <script>
      // Make sure sampleAppUrl is correct for your sample app.
      window.onload = function() {
      document.getElementById('form').onsubmit = function() {
      // Fix the URL if not using the default settings.
      var sampleAppUrl = 'http://app.example.com:8081/';
      var discovery = document.getElementById('discovery');
      discovery.value = sampleAppUrl + discovery.value.split('@', 1)[0];
      };
    };
    </script>
  </body>
</html>""" as String
return response
```

The script transforms the input into a `discovery` value for IG. This is not a requirement for deployment, only a convenience for the purposes of this example. Alternatives are described in the discovery protocol specification.

4. Add the following route to IG, replacing `/path/to/secrets/keystore.p12` with your path:

Linux

```
$HOME/.openig/config/routes/07-discovery.json
```

Windows

```
%appdata%\OpenIG\config\routes\07-discovery.json
```

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "SecretsProvider-1",
      "type": "SecretsProvider",
      "config": {
        "stores": [
          {
            "type": "KeyStoreSecretStore",
            "config": {
              "file": "/path/to/secrets/keystore.p12",
              "mappings": [
                {
                  "aliases": [ "myprivatekeyalias" ],
                  "secretId": "private.key.jwt.signing.key"
                }
              ],
              "storePasswordSecretId": "keystore.secret.id",
              "storeType": "PKCS12",
              "secretsProvider": "SystemAndEnvSecretStore-1"
            }
          }
        ]
      }
    },
    {
      "name": "DiscoveryPage",
      "type": "ScriptableHandler",
      "config": {
        "type": "application/x-groovy",
        "file": "discovery.groovy"
      }
    }
  ],
  "name": "07-discovery",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/discovery')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "DynamicallyRegisteredClient",
          "type": "AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/discovery",
            "requireHttps": false,
            "requireLogin": true,
            "target": "${attributes.openid}",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "comment": "Trivial failure handler for debugging only",
                "status": 500,
                "headers": {
                  "Content-Type": [ "text/plain; charset=UTF-8" ]
```

```
                },
                "entity": "${contexts.oauth2Failure.error}: ${contexts.oauth2Failure.description}"
              }
            },
            "loginHandler": "DiscoveryPage",
            "discoverySecretId": "private.key.jwt.signing.key",
            "tokenEndpointAuthMethod": "private_key_jwt",
            "secretsProvider": "SecretsProvider-1",
            "metadata": {
              "client_name": "My Dynamically Registered Client",
              "redirect_uris": [ "http://ig.example.com:8080/discovery/callback" ],
              "scopes": [ "openid", "profile", "email" ]
            }
          }
        },
        {
          "type": "StaticRequestFilter",
          "config": {
            "method": "POST",
            "uri": "http://app.example.com:8081/login",
            "form": {
              "username": [
                "${attributes.openid.user_info.name}"
              ],
              "password": [
                "${attributes.openid.user_info.family_name}"
              ]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Consider the differences with `07-openid.json`:

- The route matches requests to `/discovery`.
- The AuthorizationCodeOAuth2ClientFilter uses `DiscoveryPage` as the login handler, and specifies metadata to prepare the dynamic registration request.
- `DiscoveryPage` uses a ScriptableHandler and script to provide the `discovery` parameter and `goto` parameter.

  If there is a match, then it can use the issuer's registration endpoint and avoid an additional request to look up the user's issuer using the WebFinger protocol.

  If there is no match, IG uses the `discovery` value as the `resource` for a WebFinger request using OpenID Connect Discovery protocol.
- IG uses the `discovery` parameter to find an identity provider. IG extracts the domain host and port from the value, and attempts to find a match in the `supportedDomains` lists for issuers configured for the route.
- When `discoverySecretId` is set, the `tokenEndpointAuthMethod` is always `private_key_jwt`. Clients send a signed JWT to the Authorization Server.

Redirects IG to the end user's browser, using the `goto` parameter, after the process is complete and IG has injected the OpenID Connect user information into the context.

4. Test the setup:

    1. Log out of AM, and clear any cookies.
    2. Go to http://ig.example.com:8080/discovery.
    3. Enter the following email address: `demo@example.com`. The AM login page is displayed.
    4. Log in as user `demo`, password `Ch4ng31t`, and then allow the application to access user information. The sample application returns the user's page.

Passing data along the chain

Pass profile data downstream

Retrieve user profile attributes of an AM user, and provide them in the UserProfileContext to downstream filters and handlers. Profile attributes that are enabled in AM can be retrieved, except the `roles` attribute.

The `userProfile` property of AmService is configured to retrieve `employeeNumber` and `mail`. When the property is not configured, all available attributes in `rawInfo` or `asJsonValue()` are displayed.

Retrieve profile attributes for a user authenticated with an SSO token

In this example, the user is authenticated with AM through the SingleSignOnFilter, which stores the SSO token and its validation information in the `SsoTokenContext`. The UserProfileFilter retrieves the user's mail and employee number, as well as the `username`, `_id`, and `_rev`, from that context.

1. Set up AM:

    1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:

        - `https://ig.example.com:8443/*`
        - `https://ig.example.com:8443/*?*`

    2. Register an IG agent with the following values, as described in Register an IG agent in AM:

        - **Agent ID**: `ig_agent`
        - **Password**: `password`

        > **Important**
        >
        > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

    3. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

> **Important**
>
> IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

4. Select **Services** > **Add a Service**, and add a **Validation Service** with the following **Valid goto URL Resources**:
   - `http://ig.example.com:8080/*`
   - `http://ig.example.com:8080/?`

2. Set up IG:
   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
   2. Set an environment variable for the IG agent password, and then restart IG:

      ```
      $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
      ```

      The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.
   3. Add the following route to IG:

      Linux

      ```
      $HOME/.openig/config/routes/user-profile-sso.json
      ```

      Windows

      ```
      %appdata%\OpenIG\config\routes\user-profile-sso.json
      ```

```
{
  "name": "user-profile-sso",
  "condition": "${find(request.uri.path, '^/user-profile-sso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "amHandler": "ForgeRockClientHandler"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "UserProfileFilter-1",
          "type": "UserProfileFilter",
          "config": {
            "username": "${contexts.ssoToken.info.uid}",
            "userProfileService": {
              "type": "UserProfileService",
              "config": {
                "amService": "AmService-1",
                "profileAttributes": [ "employeeNumber", "mail" ]
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body>username: ${contexts.userProfile.username}<br><br>rawInfo: <pre>$
{contexts.userProfile.rawInfo}</pre></body></html>"
```

```
        }
      }
    }
  }
}
```

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/user-profile-sso.
   2. Log in to AM with username `demo` and password `Ch4ng31t`.

      The UserProfileFilter retrieves the user's profile data and stores it in the UserProfileContext. The StaticResponseHandler displays the username and the profile data available in `rawInfo`:

      ```
      username: demo

      rawInfo:

      {_id=demo, _rev=-1, mail=[demo@example.com], username=demo}
      ```

Retrieve a username from the sessionInfo context

In this example, the UserProfileFilter retrieves AM profile information for the user identified by the SessionInfoContext, at `${contexts.amSession.username}`. The SessionInfoFilter validates an SSO token without redirecting the request to an authentication page.

1. Set up AM:

   1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:

      - `https://ig.example.com:8443/*`
      - `https://ig.example.com:8443/*?*`

   2. Register an IG agent with the following values, as described in Register an IG agent in AM:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

      > **Important**
      >
      > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   3. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

      > **Important**
      >
      > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

2. Set up IG:

    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
    2. Set an environment variable for the IG agent password, and then restart IG:

    ```
    $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
    ```

    The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

    3. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/user-profile-ses-info.json
```

Windows

```
%appdata%\OpenIG\config\routes\user-profile-ses-info.json
```

```
{
  "name": "user-profile-ses-info",
  "condition": "${find(request.uri.path, '^/user-profile-ses-info')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "amHandler": "ForgeRockClientHandler"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [
        {
          "name": "SessionInfoFilter-1",
          "type": "SessionInfoFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "UserProfileFilter-1",
          "type": "UserProfileFilter",
          "config": {
            "username": "${contexts.amSession.username}",
            "userProfileService": {
              "type": "UserProfileService",
              "config": {
                "amService": "AmService-1",
                "profileAttributes": [ "employeeNumber", "mail" ]
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "application/json" ]
          },
          "entity": "{ \"username\": \"${contexts.userProfile.username}\", \"user_profile\":  $
{contexts.userProfile.asJsonValue()} }"
```

```
        }
      }
    }
  }
}
```

3. Test the setup:

   1. In a terminal window, use a `curl` command similar to the following to retrieve an access token:

```
$ curl --request POST \
--url http://am.example.com:8088/openam/json/realms/root/authenticate \
--header 'accept-api-version: resource=2.0' \
--header 'content-type: application/json' \
--header 'x-openam-username: demo' \
--header 'x-openam-password: Ch4ng31t' \
--data '{}'

{"tokenId":"AQI...AA*","successUrl":"/openam/console"}
```

   2. Access the route, providing the path to the certificate and token ID retrieved in the previous step:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--cookie 'iPlanetDirectoryPro=tokenID'  \
https://ig.example.com:8443/user-profile-ses-info | jq .

{
  "username": "demo",
  "user_profile": {
    "_id": "demo",
    "_rev": "123...456",
    "employeeNumber": ["123"],
    "mail": ["demo@example.com"],
    "username": "demo"
  }
}
```

      iPlanetDirectoryPro is the name of the AM session cookie. For more information, refer to Find the AM session cookie name.

      The UserProfileFilter retrieves the user's profile data and stores it in the UserProfileContext. The StaticResponseHandler displays the username and the profile data available in `asJsonValue()`.

Retrieve a username from the OAuth2Context

In this example, the OAuth2ResourceServerFilter validates a request containing an OAuth 2.0 access token, using the introspection endpoint, and injects the token into the OAuth2Context context. The UserProfileFilter retrieves AM profile information for the user identified by this context.

Before you start, set up and test the example in Validate access tokens through the introspection endpoint.

1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/user-profile-oauth.json
```

Windows

```
%appdata%\OpenIG\config\routes\user-profile-oauth.json
```

```
{
  "name": "user-profile-oauth",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/user-profile-oauth')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "amHandler": "ForgeRockClientHandler"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "token-resolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
```

```
            }
          }
        },
        {
          "name": "UserProfileFilter-1",
          "type": "UserProfileFilter",
          "config": {
            "username": "${contexts.oauth2.accessToken.info.sub}",
            "userProfileService": {
              "type": "UserProfileService",
              "config": {
                "amService": "AmService-1",
                "profileAttributes": [ "employeeNumber", "mail" ]
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "application/json" ]
          },
          "entity": "{ \"username\": \"${contexts.userProfile.username}\", \"user_profile\":  $
{contexts.userProfile.asJsonValue()} }"
        }
      }
    }
  }
}
```

2. Test the setup:

    1. In a terminal window, use a `curl` command similar to the following to retrieve an access token:

    ```
    $ mytoken=$(curl -s \
    --user "client-application:password" \
    --data "grant_type=password&username=demo&password=Ch4ng31t&scope=mail%20employeenumber" \
    http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
    ```

    2. Validate the access token returned in the previous step:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/user-profile-oauth | jq .

{
  "username": "demo",
  "user_profile": {
    "_id": "demo",
    "_rev": "123...456",
    "employeeNumber": ["123"],
    "mail": ["demo@example.com"],
    "username": "demo"
  }
}
```

The UserProfileFilter retrieves the user's profile data and stores it in the UserProfileContext. The StaticResponseHandler displays the username and the profile data that is available in `asJsonValue()`.

Passing runtime data downstream

The following sections describe how to pass identity or other runtime information in a JWT, downstream to a protected application:

The examples in this section use the following objects:

- JwtBuilderFilter to collect runtime information and pack it into a JWT
- HeaderFilter to add the information to the forwarded request

To help with development, the sample application includes a `/jwt` endpoint to display the JWT, verify its signature, and decrypt the JWT.

Pass runtime data in a JWT signed with a PEM

1. Set up secrets
    1. Locate a directory for secrets, and go to it:

    ```
    $ cd /path/to/secrets
    ```

    2. Generate PEM files to sign and verify the JWT:

```
$ openssl req \
-newkey rsa:2048 \
-new \
-nodes \
-x509 \
-days 3650 \
-subj "/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
-keyout id.key.for.signing.jwt.pem \
-out id.key.for.verifying.jwt.pem
```

2. Set up AM:

   1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:

      - `https://ig.example.com:8443/*`
      - `https://ig.example.com:8443/*?*`

   2. Register an IG agent with the following values, as described in Register an IG agent in AM:

      - **Agent ID**: `ig_agent`
      - **Password**: `password`

      > **Important**
      >
      > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   3. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

      > **Important**
      >
      > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

3. Set up IG:

   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

   2. Set an environment variable for the IG agent password, and then restart IG:

      ```
      $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
      ```

      The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   3. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

4. Add the following route to IG, replacing value of the property `secretsDir` with the directory for the PEM file:

Linux

```
$HOME/.openig/config/routes/jwt-builder-sign-pem.json
```

Windows

```
%appdata%\OpenIG\config\routes\jwt-builder-sign-pem.json
```

```
{
  "name": "jwt-builder-sign-pem",
  "condition": "${find(request.uri.path, '/jwt-builder-sign-pem')}",
  "baseURI": "http://app.example.com:8081",
  "properties": {
    "secretsDir": "/path/to/secrets"
  },
  "capture": "all",
  "heap": [
    {
      "name": "pemPropertyFormat",
      "type": "PemPropertyFormat"
    },
    {
      "name": "FileSystemSecretStore-1",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "&{secretsDir}",
        "suffix": ".pem",
        "mappings": [{
          "secretId": "id.key.for.signing.jwt",
          "format": "pemPropertyFormat"
        }]
      }
    },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "name": "SingleSignOnFilter",
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      }, {
        "name": "UserProfileFilter",
        "type": "UserProfileFilter",
        "config": {
          "username": "${contexts.ssoToken.info.uid}",
          "userProfileService": {
            "type": "UserProfileService",
            "config": {
```

```
              "amService": "AmService-1"
            }
          }
        }
      }, {
        "name": "JwtBuilderFilter-1",
        "type": "JwtBuilderFilter",
        "config": {
          "template": {
            "name": "${contexts.userProfile.commonName}",
            "email": "${contexts.userProfile.rawInfo.mail[0]}"
          },
          "secretsProvider": "FileSystemSecretStore-1",
          "signature": {
            "secretId": "id.key.for.signing.jwt",
            "algorithm": "RS512"
          }
        }
      }, {
        "name": "HeaderFilter-1",
        "type": "HeaderFilter",
        "config": {
          "messageType": "REQUEST",
          "add": {
            "x-openig-user": ["${contexts.jwtBuilder.value}"]
          }
        }
      }],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/jwt-builder-sign-pem`.
- The agent password for AmService is provided by a SystemAndEnvSecretStore.
- If the request does not have a valid AM session cookie, the SingleSignOnFilter redirects the request to authenticate with AM. If the request already has a valid AM session cookie, the SingleSignOnFilter passes the request to the next filter, and stores the cookie value in an SsoTokenContext.
- The UserProfileFilter reads the username from the SsoTokenContext, uses it to retrieve the user's profile info from AM, and places the data into the UserProfileContext.
- The JwtBuilderFilter refers to the secret ID of the PEM, and uses the FileSystemSecretStore to manage the secret.
- The FileSystemSecretStore mapping refers to the secret ID of the PEM, and uses the PemPropertyFormat to define the format.
- The HeaderFilter retrieves the JWT from the JwtBuilderContext, and adds it to the header field `x-openig-user` in the request, so that the sample app can display the JWT.
- The ClientHandler passes the request to the sample app, which displays the JWT.

4. Test the setup:

    1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/jwt-builder-sign-pem.
    2. Log in to AM as user `demo`, password `Ch4ng31t`. The sample application displays the signed JWT along with its header and payload.
    3. In `USE PEM FILE` in the sample app, enter the path to `id.key.for.verifying.jwt.pem` to verify the JWT signature.

Pass runtime data in a JWT signed with PEM then encrypted with a symmetric key

This example passes runtime data in a JWT that is signed with a PEM, and then encrypted with a symmetric key.

1. Set up secrets

    1. Locate a directory for secrets, and go to it:

    ```
    $ cd /path/to/secrets
    ```

    2. From the secrets directory, generate PEM files to sign and verify the JWT:

    ```
    $ openssl req \
    -newkey rsa:2048 \
    -new \
    -nodes \
    -x509 \
    -days 3650 \
    -subj "/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
    -keyout id.key.for.signing.jwt.pem \
    -out id.key.for.verifying.jwt.pem
    ```

    3. Encrypt the PEM file used to sign the JWT:

    ```
    $ openssl pkcs8 \
    -topk8 \
    -inform PEM \
    -outform PEM \
    -in id.key.for.signing.jwt.pem \
    -out id.encrypted.key.for.signing.jwt.pem \
    -passout pass:encryptedpassword \
    -v1 PBE-SHA1-3DES
    ```

    The encrypted PEM file used for signatures is `id.encrypted.key.for.signing.jwt.pem`. The password to decode the file is `encryptedpassword`.

    > **Tip**
    >
    > If encryption fails, make sure your encryption methods and ciphers are supported by the Java Cryptography Extension.

4. Generate a symmetric key to encrypt the JWT:

```
$ openssl rand -base64 32 > symmetric.key.for.encrypting.jwt
```

5. Make sure you have the following keys in your secrets directory:
   - `id.encrypted.key.for.signing.jwt.pem`
   - `id.key.for.signing.jwt.pem`
   - `id.key.for.verifying.jwt.pem`
   - `symmetric.key.for.encrypting.jwt`

2. Set up AM:
   1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
      - `https://ig.example.com:8443/*`
      - `https://ig.example.com:8443/*?*`
   2. Register an IG agent with the following values, as described in Register an IG agent in AM:
      - **Agent ID**: `ig_agent`
      - **Password**: `password`

      > **Important**
      >
      > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

   3. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

      > **Important**
      >
      > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

3. Set up IG:
   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
   2. Set an environment variable for the IG agent password, and then restart IG:

      ```
      $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
      ```

      The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   3. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

4. In IG, create an environment variable for the base64-encoded password to decrypt the PEM file used to sign the JWT:

```
$ export ID_DECRYPTED_KEY_FOR_SIGNING_JWT='ZW5jcnlwdGVkcGFzc3dvcmQ='
```

5. Add the following route to IG, replacing the value of `secretsDir` with your secrets directory:

Linux

```
$HOME/.openig/config/routes/jwtbuilder-sign-then-encrypt.json
```

Windows

```
%appdata%\OpenIG\config\routes\jwtbuilder-sign-then-encrypt.json
```

```
{
  "name": "jwtbuilder-sign-then-encrypt",
  "condition": "${find(request.uri.path, '/jwtbuilder-sign-then-encrypt')}",
  "baseURI": "http://app.example.com:8081",
  "properties": {
    "secretsDir": "/path/to/secrets"
  },
  "capture": "all",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore",
      "type": "SystemAndEnvSecretStore",
      "config": {
        "mappings": [{
          "secretId": "id.decrypted.key.for.signing.jwt",
          "format": "BASE64"
        }]
      }
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore",
        "url": "http://am.example.com:8088/openam"
      }
    },
    {
      "name": "pemPropertyFormat",
      "type": "PemPropertyFormat",
      "config": {
        "decryptionSecretId": "id.decrypted.key.for.signing.jwt",
        "secretsProvider": "SystemAndEnvSecretStore"
      }
    },
    {
      "name": "FileSystemSecretStore-1",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "&{secretsDir}",
        "mappings": [{
          "secretId": "id.encrypted.key.for.signing.jwt.pem",
          "format": "pemPropertyFormat"
        }, {
          "secretId": "symmetric.key.for.encrypting.jwt",
          "format": {
            "type": "SecretKeyPropertyFormat",
            "config": {
              "format": "BASE64",
              "algorithm": "AES"
            }
          }
        }]
      }
    }
```

```
    ],
    "handler": {
      "type": "Chain",
      "config": {
        "filters": [{
          "name": "SingleSignOnFilter",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }, {
          "name": "UserProfileFilter",
          "type": "UserProfileFilter",
          "config": {
            "username": "${contexts.ssoToken.info.uid}",
            "userProfileService": {
              "type": "UserProfileService",
              "config": {
                "amService": "AmService-1"
              }
            }
          }
        }, {
          "name": "JwtBuilderFilter-1",
          "type": "JwtBuilderFilter",
          "config": {
            "template": {
              "name": "${contexts.userProfile.commonName}",
              "email": "${contexts.userProfile.rawInfo.mail[0]}"
            },
            "secretsProvider": "FileSystemSecretStore-1",
            "signature": {
              "secretId": "id.encrypted.key.for.signing.jwt.pem",
              "algorithm": "RS512",
              "encryption": {
                "secretId": "symmetric.key.for.encrypting.jwt",
                "algorithm": "dir",
                "method": "A128CBC-HS256"
              }
            }
          }
        }, {
          "name": "AddBuiltJwtToHeader",
          "type": "HeaderFilter",
          "config": {
            "messageType": "REQUEST",
            "add": {
              "x-openig-user": ["${contexts.jwtBuilder.value}"]
            }
          }
        },
          {
            "name": "AddBuiltJwtAsCookie",
            "type": "HeaderFilter",
            "config": {
              "messageType": "RESPONSE",
              "add": {
                "set-cookie": ["my-jwt=${contexts.jwtBuilder.value};PATH=/"]
              }
            }
```

```
          }],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

Notice the following features of the route:

- The route matches requests to `/jwtbuilder-sign-then-encrypt`.
- The SystemAndEnvSecretStore provides the IG agent password and the password to decode the PEM file for the signing keys.
- The FileSystemSecretStore maps the secret IDs of the encrypted PEM file used to sign the JWT, and the symmetric key used to encrypt the JWT.
- After authentication, the UserProfileFilter reads the username from the SsoTokenContext, uses it to retrieve the user's profile info from AM, and places the data into the UserProfileContext.
- The JwtBuilderFilter takes the username and email from the UserProfileContext, and stores them in a JWT in the JwtBuilderContext. It uses the secrets mapped in the FileSystemSecretStore to sign then encrypt the JWT.
- The `AddBuiltJwtToHeader` HeaderFilter retrieves the JWT from the JwtBuilderContext, and adds it to the header field `x-openig-user` in the request so that the sample app can display the JWT.
- The `AddBuiltJwtAsCookie` HeaderFilter adds the JWT to a cookie called `my-jwt` so that it can be retrieved by the JwtValidationFilter in JWT validation. The cookie is ignored in this example.
- The ClientHandler passes the request to the sample app.

4. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/jwtbuilder-sign-then-encrypt.
   2. Log in to AM as user `demo`, password `Ch4ng31t`. The sample app displays the encrypted JWT. The payload is concealed because the JWT is encrypted.
   3. In the `ENTER SECRET` box, enter the value of `symmetric.key.for.encrypting.jwt` to decrypt the JWT. The signed JWT and its payload are now displayed.
   4. In the `USE PEM FILE` box, enter the path to `id.key.for.verifying.jwt.pem` to verify the JWT signature.

Pass runtime data in JWT encrypted with a symmetric key

1. Set up secrets:

   1. Locate a directory for secrets, and go to it:

      ```
      $ cd /path/to/secrets
      ```

   2. In the secrets folder, generate an AES 256-bit key:

```
$ openssl rand -base64 32

loH...UFQ=
```

3. In the secrets folder, create a file called `symmetric.key.for.encrypting.jwt` containing the AES key:

```
$ echo -n 'loH...UFQ=' > symmetric.key.for.encrypting.jwt
```

Make sure the password file contains only the password, with no trailing spaces or carriage returns.

2. Set up AM:

1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:

- `https://ig.example.com:8443/*`
- `https://ig.example.com:8443/*?*`

2. Register an IG agent with the following values, as described in Register an IG agent in AM:

- **Agent ID**: `ig_agent`
- **Password**: `password`

> **Important**
>
> Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

3. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

> **Important**
>
> IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

3. Set up IG:

1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

2. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

3. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

4. Add the following route to IG, replacing the value of the property `secretsDir` with your value:

Linux

```
$HOME/.openig/config/routes/jwtbuilder-encrypt-symmetric.json
```

Windows

```
%appdata%\OpenIG\config\routes\jwtbuilder-encrypt-symmetric.json
```

```
{
  "name": "jwtbuilder-encrypt-symmetric",
  "condition": "${find(request.uri.path, '/jwtbuilder-encrypt-symmetric')}",
  "baseURI": "http://app.example.com:8081",
  "properties": {
    "secretsDir": "/path/to/secrets"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam"
      }
    },
    {
      "name": "FileSystemSecretStore-1",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "&{secretsDir}",
        "mappings": [{
          "secretId": "symmetric.key.for.encrypting.jwt",
          "format": {
            "type": "SecretKeyPropertyFormat",
            "config": {
              "format": "BASE64",
              "algorithm": "AES"
            }
          }
        }]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "name": "SingleSignOnFilter-1",
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      }, {
        "name": "UserProfileFilter-1",
        "type": "UserProfileFilter",
        "config": {
          "username": "${contexts.ssoToken.info.uid}",
          "userProfileService": {
            "type": "UserProfileService",
            "config": {
```

```
                "amService": "AmService-1"
              }
            }
          }
        }, {
          "name": "JwtBuilderFilter-1",
          "type": "JwtBuilderFilter",
          "config": {
            "template": {
              "name": "${contexts.userProfile.commonName}",
              "email": "${contexts.userProfile.rawInfo.mail[0]}"
            },
            "secretsProvider": "FileSystemSecretStore-1",
            "encryption": {
              "secretId": "symmetric.key.for.encrypting.jwt",
              "algorithm": "dir",
              "method": "A128CBC-HS256"
            }
          }
        }, {
          "name": "HeaderFilter-1",
          "type": "HeaderFilter",
          "config": {
            "messageType": "REQUEST",
            "add": {
              "x-openig-user": ["${contexts.jwtBuilder.value}"]
            }
          }
        }],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

Notice the following features of the route:

- The route matches requests to `/jwtbuilder-encrypt-symmetric`.
- The JWT encryption key is managed by the FileSystemSecretStore in the heap, which defines the SecretKeyPropertyFormat.
- The JwtBuilderFilter `encryption` property refers to key in the FileSystemSecretStore.
- The HeaderFilter retrieves the JWT from the JwtBuilderContext, and adds it to the header field `x-openig-user` in the request, so that the sample app can display the JWT.

4. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/jwtbuilder-encrypt-symmetric.
   2. Log in to AM as user `demo`, password `Ch4ng31t`, or as another user. The JWT is displayed in the sample app.
   3. In the `ENTER SECRET` field, enter the value of the AES 256-bit key to decrypt the JWT and display its payload.

Pass runtime data in JWT encrypted with an asymmetric key

The asymmetric key in this example is a PEM, but you can equally use a keystore.

1. Set up secrets:
    1. Locate a directory for secrets, and go to it:

    ```
    $ cd /path/to/secrets
    ```

    2. Generate an encrypted PEM file:

    ```
    $ openssl req \
    -newkey rsa:2048 \
    -new \
    -nodes \
    -x509 \
    -days 3650 \
    -subj "/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
    -keyout id.key.for.encrypting.jwt.pem \
    -out id.key.for.decrypting.jwt.pem
    ```

2. Set up AM:
    1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
        - `https://ig.example.com:8443/*`
        - `https://ig.example.com:8443/*?*`
    2. Register an IG agent with the following values, as described in Register an IG agent in AM:
        - **Agent ID**: `ig_agent`
        - **Password**: `password`

        > **Important**
        >
        > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

    3. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

    > **Important**
    >
    > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

3. Set up IG:
    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

2. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

3. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

4. Add the following route to IG, replacing value of the property `secretsDir` with the directory for the PEM file:

Linux

```
$HOME/.openig/config/routes/jwtbuilder-encrypt-asymmetric.json
```

Windows

```
%appdata%\OpenIG\config\routes\jwtbuilder-encrypt-asymmetric.json
```

```
{
  "name": "jwtbuilder-encrypt-asymmetric",
  "condition": "${find(request.uri.path, '/jwtbuilder-encrypt-asymmetric')}",
  "baseURI": "http://app.example.com:8081",
  "properties": {
    "secretsDir": "/path/to/secrets"
  },
  "capture": "all",
  "heap": [
    {
      "name": "pemPropertyFormat",
      "type": "PemPropertyFormat"
    },
    {
      "name": "FileSystemSecretStore-1",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "&{secretsDir}",
        "suffix": ".pem",
        "mappings": [{
          "secretId": "id.key.for.decrypting.jwt",
          "format": "pemPropertyFormat"
        }]
      }
    },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "name": "SingleSignOnFilter",
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      }, {
        "name": "UserProfileFilter",
        "type": "UserProfileFilter",
        "config": {
          "username": "${contexts.ssoToken.info.uid}",
          "userProfileService": {
            "type": "UserProfileService",
            "config": {
```

```
                "amService": "AmService-1"
              }
            }
          }
        }, {
          "name": "JwtBuilderFilter-1",
          "type": "JwtBuilderFilter",
          "config": {
            "template": {
              "name": "${contexts.userProfile.commonName}",
              "email": "${contexts.userProfile.rawInfo.mail[0]}"
            },
            "secretsProvider": "FileSystemSecretStore-1",
            "encryption": {
              "secretId": "id.key.for.decrypting.jwt",
              "algorithm": "RSA-OAEP-256",
              "method": "A128CBC-HS256"
            }
          }
        }, {
          "name": "HeaderFilter-1",
          "type": "HeaderFilter",
          "config": {
            "messageType": "REQUEST",
            "add": {
              "x-openig-user": ["${contexts.jwtBuilder.value}"]
            }
          }
        }],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

Notice the following features of the route:

- The route matches requests to `/jwtbuilder-encrypt-asymmetric`.
- The JwtBuilderFilter refers to the secret ID of the PEM, and uses the FileSystemSecretStore to manage the secret.
- The FileSystemSecretStore mapping refers to the secret ID of the PEM, and uses the default PemPropertyFormat.
- The HeaderFilter retrieves the JWT from the JwtBuilderContext, and adds it to the header field `x-openig-user` in the request, so that the sample app can display the JWT.

4. Test the setup:

    1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/jwtbuilder-encrypt-asymmetric.
    2. Log in to AM as user `demo`, password `Ch4ng31t`, or as another user. The JWT is displayed in the sample app.
    3. In the `USE PEM FILE` field, enter the path to `id.key.for.encrypting.jwt.pem` to decrypt the JWT and display its payload.

SAML

IG implements SAML 2.0 to validate users and log them in to protected applications.

For more information about the SAML 2.0 standard, refer to RFC 7522. The following terms are used:

- *Identity Provider* (IDP): The service that manages the user identity, for example Identity Cloud or AM.
- *Service Provider* (SP): The service that a user wants to access. IG acts as a SAML 2.0 SP for SSO, providing an interface to applications that don't support SAML 2.0.
- *Circle of trust* (CoT): An IDP and SP that participate in federation.
- *Fedlet*: SAML configuration files.

SAML assertions

SAML assertions can be signed and encrypted. ForgeRock recommends using *SHA-256 variants (rsa-sha256 or ecdsa-sha256).

SAML assertions can contain configurable attribute values, such as user meta-information or anything else provided by the IDP. The attributes of a SAML assertion can contain one or more values, made available as a list of strings. Even if an attribute contains a single value, it is made available as a list of strings.

SAML configuration

IG scans SAML configuration files once, the first time that a request accesses the SamlFederationFilter or SamlFederationHandler (deprecated) after startup. Restart IG after any change to the SAML configuration files.

SAML in deployments with multiple instances of IG

When IG acts as a SAML service provider, session information is stored in the fedlet not the session cookie. In deployments with multiple instances of IG as a SAML service provider, it is necessary to set up sticky sessions so that requests always hit the instance where the SAML interaction was started.

For information, refer to Session state considerations in AM's *SAML v2.0 guide*.

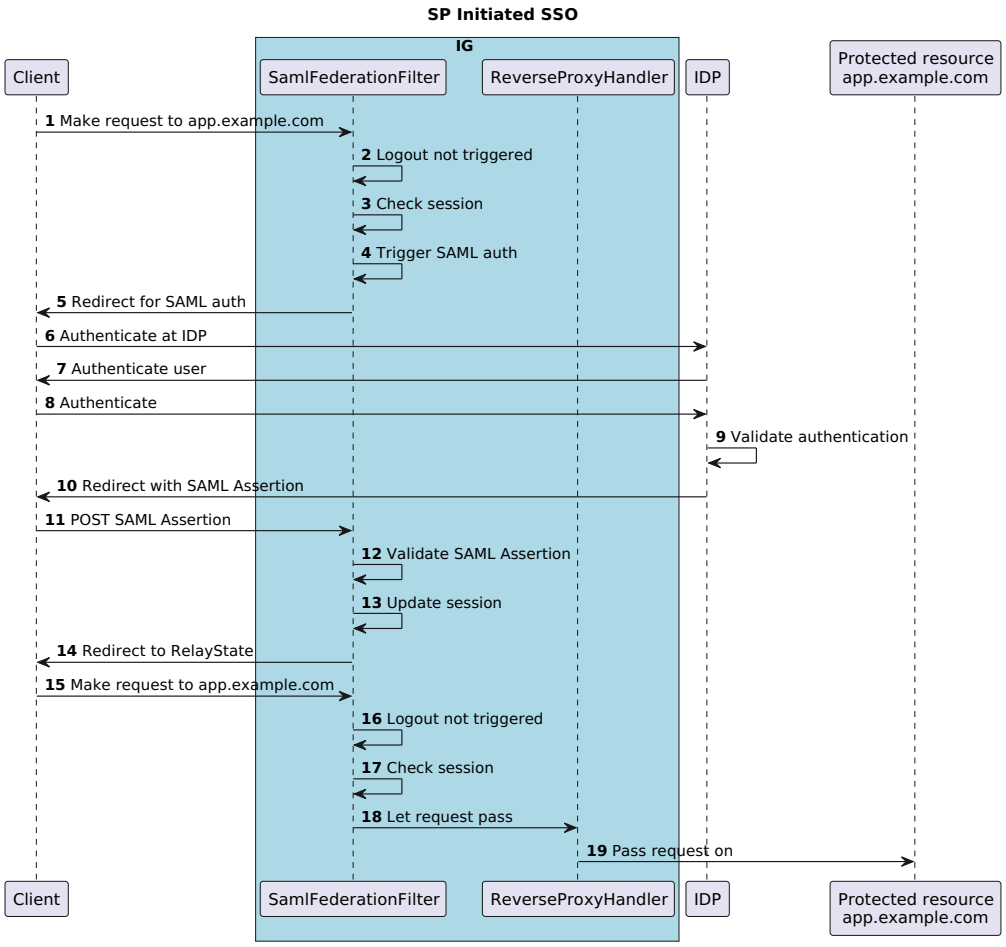
About SP-initiated SSO

SP-initiated SSO occurs when a user attempts to access a protected application directly through the SP. Because the user's federated identity is managed by the IDP, the SP sends a SAML authentication request to the IDP. After the IDP authenticates the user, it provides the SP with a SAML assertion for the user.

For the SamlFederationFilter, SP-initiated SSO is the preferred to IDP-initiated SSO:

- A dedicated SAML URI is not required to start SP-initiated authentication.
- The HTTP session tracks the state of the user session.

The following sequence diagram shows the flow of information in SP-initiated SSO, when IG acts as a SAML 2.0 SP:



SAML with AM as the identity provider

SAML with AM as the identity provider using unsigned/unencrypted assertions

This example set up federation using AM as the identity provider with unsigned/unencrypted assertions.

1. Set up the network:

   Add `sp.example.com` to your `/etc/hosts` file:

   ```
   127.0.0.1 localhost am.example.com ig.example.com app.example.com sp.example.com
   ```

   Traffic to the application is proxied through IG, using the host name `sp.example.com`.

2. Configure a Java Fedlet:

Note

The SAML library component validates the SP's AssertionConsumerService Location against the incoming IDP SAML Assertion, based on the request information, including the port. In `sp.xml`, always specify the port in the Location value of `AssertionConsumerService`, even when using defaults of 443 or 80, as follows:

```
<AssertionConsumerService isDefault="true"
                          index="0"
                          Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
                          Location="https://sp.example.com:443/fedletapplication" />
```

For more information about Java Fedlets, refer to Creating and configuring the Fedlet in AM's *SAML v2.0 guide*.

1. Copy and unzip the fedlet zip file, `Fedlet-7.4.0.zip`, delivered with the AM installation, into a local directory.

```
$ unzip $HOME/openam/Fedlet-7.4.0.zip

Archive:  Fedlet-7.4.0.zip
creating: conf/
inflating: README
inflating: conf/FederationConfig.properties
inflating: conf/fedlet.cot-template
inflating: conf/idp-extended.xml-template
inflating: conf/sp-extended.xml-template
inflating: conf/sp.xml-template
inflating: fedlet.war
```

2. In each file, search and replace the following properties:

Replace this	With this
`IDP_ENTITY_ID`	`openam`
`FEDLET_ENTITY_ID`	`sp`
`FEDLET_PROTOCOL://FEDLET_HOST:FEDLET_PORT/FEDLET_DEPLOY_URI`	`https://sp.example.com:8443/home/saml`
`fedletcot` and `FEDLET_COT`	`Circle of Trust`
`sp.example.com:8443/home/saml/fedletapplication`	`sp.example.com:8443/home/saml/fedletapplication/metaAlias/sp`

3. Save the files as .xml, without the `-template` extension, so that the directory looks like this:

```
conf
├── FederationConfig.properties
├── fedlet.cot
├── idp-extended.xml
├── sp-extended.xml
└── sp.xml
```

By default, AM as an IDP uses the NameID format `urn:oasis:names:tc:SAML:2.0:nameid-format:transient` to communicate about a user. For information about using a different NameID format, refer to Use a non-transient NameID format.

3. Set up AM:

    1. In the AM admin UI, select **Identities**, select the user `demo`, and change the last name to `Ch4ng31t`. Note that, for this example, the last name must be the same as the password.
    2. Select **Applications** > **Federation** > **Circles of Trust**, and add a circle of trust called `Circle of Trust`, with the default settings.
    3. Set up a remote service provider:
        1. Select **Applications** > **Federation** > **Entity Providers**, and add a remote entity provider.
        2. Drag in or import `sp.xml` created in the previous step.
        3. Select **Circles of Trust**: `Circle of Trust`.
    4. Set up a hosted identity provider:
        1. Select **Applications** > **Federation** > **Entity Providers**, and add a hosted entity provider with the following values:
            - **Entity ID**: `openam`
            - **Entity Provider Base URL**: `http://am.example.com:8088/openam`
            - **Identity Provider Meta Alias**: `idp`
            - **Circles of Trust**: `Circle of Trust`
        2. Select **Assertion Processing** > **Attribute Mapper**, map the following SAML attribute keys and values, and then save your changes:
            - **SAML Attribute**: `cn`, **Local Attribute**: `cn`
            - **SAML Attribute**: `sn`, **Local Attribute**: `sn`
        3. In a terminal, export the XML-based metadata for the IDP:

            ```
            $ curl -v \
            --output idp.xml \
            "http://am.example.com:8088/openam/saml2/jsp/exportmetadata.jsp?entityid=openam"
            ```

            The `idp.xml` file is created locally.

4. Set up IG:

    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

    2. Copy the edited fedlet files, and the exported `idp.xml` file into the IG configuration, at `$HOME/.openig/SAML`.

    ```
    $ ls -l $HOME/.openig/SAML

    FederationConfig.properties
    fedlet.cot
    idp-extended.xml
    idp.xml
    sp-extended.xml
    sp.xml
    ```

    3. Add the following route to IG to serve the sample application .css and other static resources:

    Linux

    ```
    $HOME/.openig/config/routes/00-static-resources.json
    ```

    Windows

    ```
    %appdata%\OpenIG\config\routes\00-static-resources.json
    ```

    ```
    {
      "name" : "00-static-resources",
      "baseURI" : "http://app.example.com:8081",
      "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
    or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
      "handler": "ReverseProxyHandler"
    }
    ```

    4. Add the following route to IG:

    Linux

    ```
    $HOME/.openig/config/routes/saml-filter.json
    ```

Windows

```
%appdata%\OpenIG\config\routes\saml-filter.json
```

```
{
  "name": "saml-filter",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SamlFilter",
          "type": "SamlFederationFilter",
          "config": {
            "assertionMapping": {
              "name": "cn",
              "surname": "sn"
            },
            "subjectMapping": "sp-subject-name",
            "redirectURI": "/home/saml-filter"
          }
        },
        {
          "name": "SetSamlHeaders",
          "type": "HeaderFilter",
          "config": {
            "messageType": "REQUEST",
            "add": {
              "x-saml-cn": [ "${toString(session.name)}" ],
              "x-saml-sn": [ "${toString(session.surname)}" ]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/home`.
- The SamlFederationFilter extracts `cn` and `sn` from the SAML assertion, and maps them to the SessionContext, at `session.name[0]` and `session.surname[0]`.
- The HeaderFilter adds the session name and surname as headers to the request so that they are displayed by the sample application.

5. Restart IG.

5. Test the setup:

    1. In your browser's privacy or incognito mode, go to https://sp.example.com:8443/home.
    2. Log in to AM as user `demo`, password `Ch4ng31t`. The request is redirected to the sample application.

> **Tip**
>
> If a request returns an HTTP 414 URI Too Long error, consider the information in URI Too Long error.

SAML with AM as the identity provider using signed/encrypted assertions

This example set up federation using AM as the identity provider with signed/encrypted assertions.

Before you start, set up and test the example in SAML with AM as the identity provider using unsigned/unencrypted assertions.

1. Set up the SAML keystore:

    1. Find the values of AM's default SAML keypass and storepass:

    ```
    $ more /path/to/am/secrets/default/.keypass
    $ more /path/to/am/secrets/default/.storepass
    ```

    2. Copy the SAML keystore from the AM configuration to IG:

    ```
    $ cp /path/to/am/secrets/keystores/keystore.jceks /path/to/ig/secrets/keystore.jceks
    ```

    > **Warning**
    >
    > Legacy keystore types such as JKS and JCEKS are supported but are not secure. Consider using the PKCS#12 keystore type.

2. Configure the Fedlet in IG:

    1. In `FederationConfig.properties`, make the following changes:

        1. Delete the following lines:

            - `com.sun.identity.saml.xmlsig.keystore=%BASE_DIR%/security/keystores/keystore.jks`
            - `com.sun.identity.saml.xmlsig.storepass=%BASE_DIR%/.storepass`
            - `com.sun.identity.saml.xmlsig.keypass=%BASE_DIR%/.keypass`
            - `com.sun.identity.saml.xmlsig.certalias=test`
            - `com.sun.identity.saml.xmlsig.storetype=JKS`
            - `am.encryption.pwd=@AM_ENC_PWD@`

        2. Add the following line:

`org.forgerock.openam.saml2.credential.resolver.class=org.forgerock.openig.handler.saml.SecretsSaml2CredentialResolver`

This class is responsible for resolving secrets and supplying credentials.

> **Tip**
>
> Be sure to leave no space at the end of the line.

2. In `sp.xml`, make the following changes:

    1. Change `AuthnRequestsSigned="false"` to `AuthnRequestsSigned="true"`.

    2. Add the following KeyDescriptor just before `</SPSSODescriptor>`

```
        <KeyDescriptor use="signing">
            <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#" >
                <ds:X509Data>
                    <ds:X509Certificate>

                    </ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </KeyDescriptor>
    </SPSSODescriptor>
```

    3. Copy the value of the signing certificate from `idp.xml` to this file:

```
<KeyDescriptor use="signing">
  <ds:KeyInfo>
    <ds:X509Data>
      <ds:X509Certificate>

        MII...zA6

      </ds:X509Certificate>
```

    This is the public key used for signing so that the IDP can verify request signatures.

3. Replace the remote service provider in AM:

    1. Select **Applications** > **Federation** > **Entity Providers**, and remove the `sp` entity provider.

    2. Drag in or import the new `sp.xml` updated in the previous step.

    3. Select **Circles of Trust**: `Circle of Trust`.

4. Set up IG

    1. In the IG configuration, set environment variables for the following secrets, and then restart IG:

```
$ export KEYSTORE_SECRET_ID='a2V5c3RvcmU='
$ export SAML_KEYSTORE_STOREPASS_SECRET_ID='base64-encoded value of the SAML storepass'
$ export SAML_KEYSTORE_KEYPASS_SECRET_ID='base64-encoded value of the SAML keypass'
```

The passwords are retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

2. Remove `saml-filter.json` from the configuration, and add the following route, replacing the path to `keystore.jceks` with your path:

Linux

```
$HOME/.openig/config/routes/saml-filter-secure.json
```

Windows

```
%appdata%\OpenIG\config\routes\saml-filter-secure.json
```

```
{
  "name": "saml-filter-secure",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "KeyStoreSecretStore-1",
      "type" : "KeyStoreSecretStore",
      "config" : {
        "file" : "/path/to/ig/keystore.jceks",
        "storeType" : "jceks",
        "storePasswordSecretId" : "saml.keystore.storepass.secret.id",
        "entryPasswordSecretId" : "saml.keystore.keypass.secret.id",
        "secretsProvider" : "SystemAndEnvSecretStore-1",
        "mappings" : [ {
          "secretId" : "sp.signing.sp",
          "aliases" : [ "rsajwtsigningkey" ]
        }, {
          "secretId" : "sp.decryption.sp",
          "aliases" : [ "test" ]
        } ]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SamlFilter",
          "type": "SamlFederationFilter",
          "config": {
            "assertionMapping": {
              "name": "cn",
              "surname": "sn"
            },
            "subjectMapping": "sp-subject-name",
            "redirectURI": "/home/saml-filter",
            "secretsProvider" : "KeyStoreSecretStore-1"
          }
        },
        {
          "name": "SetSamlHeaders",
          "type": "HeaderFilter",
          "config": {
            "messageType": "REQUEST",
            "add": {
              "x-saml-cn": [  "${toString(session.name)}" ],
              "x-saml-sn": [  "${toString(session.surname)}" ]
            }
          }
        }
```

```
        ],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

Notice the following features of the route compared to `saml-filter.json`:

- The SamlFederationHandler refers to the KeyStoreSecretStore to provide the keys for the signed and encrypted SAML assertions.
- The secret IDs, `sp.signing.sp` and `sp.decryption.sp`, follow a naming convention based on the name of the service provider, `sp`.
- The alias for the signing key corresponds to the PEM in `keystore.jceks`.

3. Restart IG.

5. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://sp.example.com:8443/home.
   2. Log in to AM as user `demo`, password `Ch4ng31t`. The request is redirected to the sample application.

> **Tip**
>
> If a request returns an HTTP 414 URI Too Long error, consider the information in URI Too Long error.

SAML with PingOne as the identity provider

This example set up federation using PingOne as the identity provider with unsigned/unencrypted assertions.

Before you start, prepare IG and the sample application as described in the Quick install.

1. Set up the PingOne environment:

   1. Create a PingOne test environment.
   2. Add a PingOne test user.

2. Set up the network:

   Add `sp.example.com` to your `/etc/hosts` file:

   ```
   127.0.0.1 localhost am.example.com ig.example.com app.example.com sp.example.com
   ```

   Traffic to the application is proxied through IG, using the host name `sp.example.com`.

3. Save the file sp.xml as the SAML service provider configuration file `$HOME/.openig/SAML/sp.xml`.

4. Create a PingOne SAML application.

Learn more from PingOne's Add a SAML application.

1. In the PingOne test environment, create a web application with the following values:
   - **Application Name**: `saml_app`
   - **Description**: `SAML application`
   - **Application Type**: `SAML Application`
2. In the application, select the **Import Metadata** panel, drag-in or select the SAML configuration file `sp.xml`, and then save the application.
3. On the **Attribute Mappings** panel, click (edit) and add the following mappings:

saml_app	PingOne
cn	Given Name
sn	Family Name

4. On the **Configuration** panel, click (edit) and set the SLO BINDING's **SUBJECT NAMEID FORMAT** to `urn:oasis:names:tc:SAML:2.0:nameid-format:transient`.
5. On the **Configuration** panel, click **Download Metadata** and save the downloaded file as the identity provider configuration file `$HOME/.openig/SAML/idp.xml`.
6. On the **Configuration** panel, note the **Initiate Single Sign-on URL**. The value is used in the IG setup.
7. At the top-right of the page, click the slider to enable the application.

5. Complete the SAML configuration:
   1. Copy the following example SAML configuration files to `$HOME/.openig/SAML` and edit them to match your configuration:

File	Required changes
FederationConfig.properties	None
fedlet.cot	Replace idp-entityID with the value of `EntityDescriptor entityID` in `idp.xml`.
idp-extended.xml	Replace idp-entityID with the value of `EntityDescriptor entityID` in `idp.xml`.
sp-extended.xml	None

   2. Make sure that the IG configuration at `$HOME/.openig/SAML` contains the following files.

```
$ ls -l $HOME/.openig/SAML

FederationConfig.properties
fedlet.cot
idp-extended.xml
idp.xml
sp-extended.xml
sp.xml
```

6. Set up IG:

    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

    2. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

    3. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/saml-filter.json
```

Windows

```
%appdata%\OpenIG\config\routes\saml-filter.json
```

```
{
  "name": "saml-filter",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SamlFilter",
          "type": "SamlFederationFilter",
          "config": {
            "assertionMapping": {
              "name": "cn",
              "surname": "sn"
            },
            "subjectMapping": "sp-subject-name",
            "redirectURI": "/home/saml-filter"
          }
        },
        {
          "name": "SetSamlHeaders",
          "type": "HeaderFilter",
          "config": {
            "messageType": "REQUEST",
            "add": {
              "x-saml-cn": [ "${toString(session.name)}" ],
              "x-saml-sn": [ "${toString(session.surname)}" ]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

4. Restart IG.

7. Test IDP-initiated login:

   1. In your browser's privacy or incognito mode, go to the URL given by the web application property **Initiate Single Sign-on URL**.

      The PingOne login page is displayed.

   2. Log in to PingOne as user `demo`, password `Ch4ng3!t`.

   3. If prompted, change the password of the demo user.

The home page of the sample application is displayed.

8. Test SP-initiated login:

   1. In your browser's privacy or incognito mode, go to https://sp.example.com:8443/home.
   2. Log in as user `demo`, password `Ch4ng31t`. The request is redirected to the sample application.

      The home page of the sample application is displayed.

> **Tip**
>
> If a request returns an HTTP 414 URI Too Long error, consider the information in URI Too Long error.

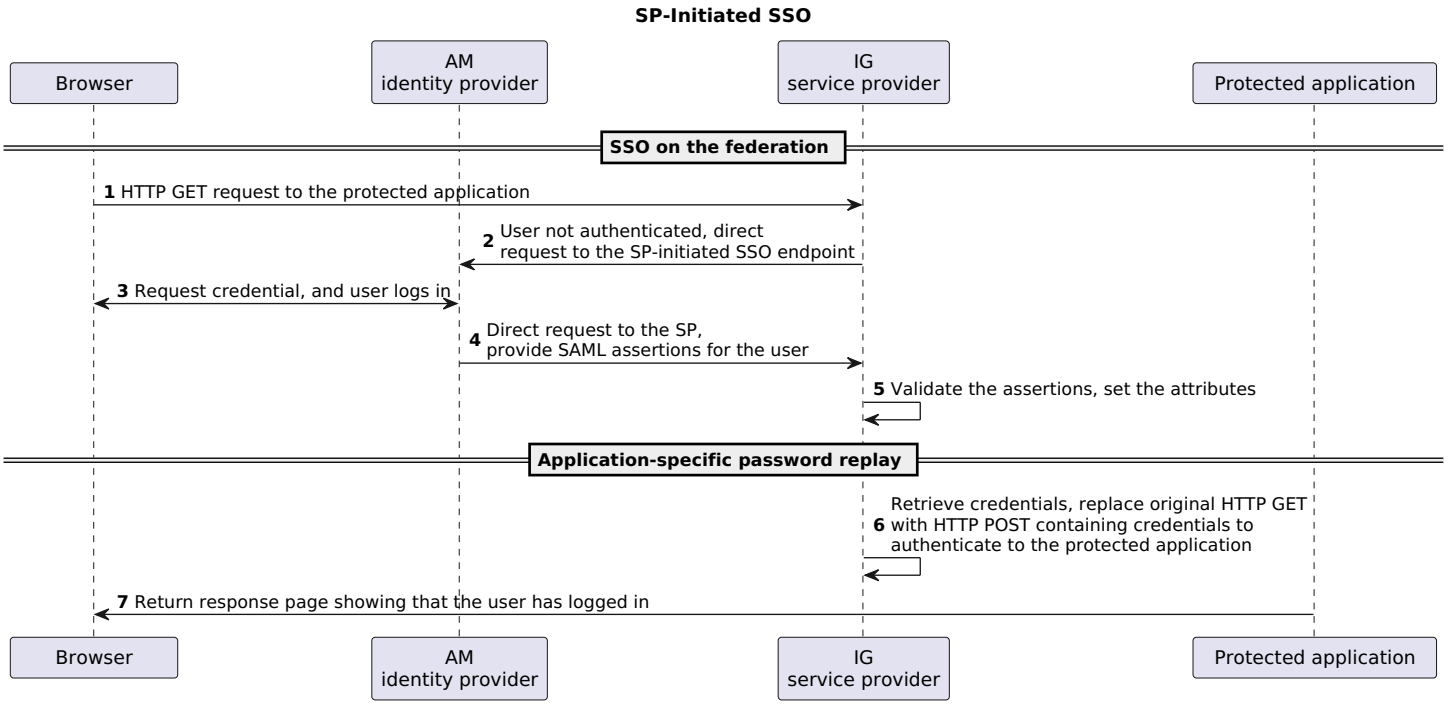
Federation using the SamlFederationHandler (deprecated)

> **Important**
>
> The SamlFederationHandler is deprecated; use the SamlFederationFilter instead. For more information, refer to the Deprecated section of the *Release Notes*.

About SP-initiated SSO with the SamlFederationHandler

SP-initiated SSO occurs when a user attempts to access a protected application directly through the SP. Because the user's federated identity is managed by the IDP, the SP sends a SAML authentication request to the IDP. After the IDP authenticates the user, it provides the SP with a SAML assertion for the user.

The following sequence diagram shows the flow of information in SP-initiated SSO, when IG acts as a SAML 2.0 SP:



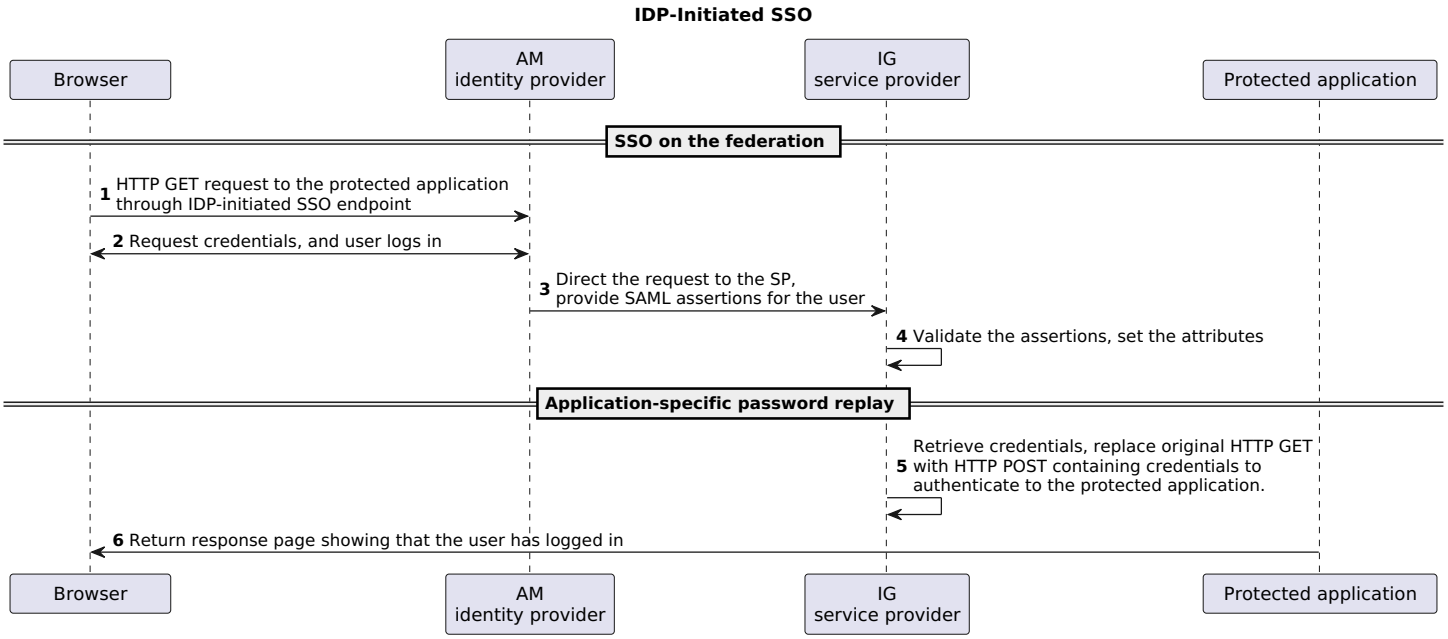
About IDP-initiated SSO the the SamlFederationHandler

IDP-initiated SSO occurs when a user attempts to access a protected application, using the IDP for authentication. The IDP sends an unsolicited authentication statement to the SP.

Before IDP-initiated SSO can occur:

- The user must access a link on the IDP that refers to the remote SP.
- The user must authenticate to the IDP.
- The IDP must be configured with links that refer to the SP.

The following sequence diagram shows the flow of information in IDP-initiated SSO when IG acts as a SAML 2.0 SP:



Set up federation with unsigned/unencrypted assertions with the SamlFederationHandler

For examples of the federation configuration files, refer to Example fedlet files. To set up multiple SPs, work through this section, and then SAML 2.0 and multiple applications.

1. Set up the network:

   Add `sp.example.com` to your `/etc/hosts` file:

   ```
   127.0.0.1 localhost am.example.com ig.example.com app.example.com sp.example.com
   ```

   Traffic to the application is proxied through IG, using the host name `sp.example.com`.

2. Configure a Java Fedlet:

Note

The SAML library component validates the SP's AssertionConsumerService Location against the incoming IDP SAML Assertion, based on the request information, including the port. In `sp.xml`, always specify the port in the Location value of `AssertionConsumerService`, even when using defaults of 443 or 80, as follows:

```
<AssertionConsumerService isDefault="true"
                          index="0"
                          Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
                          Location="https://sp.example.com:443/fedletapplication" />
```

For more information about Java Fedlets, refer to Creating and configuring the Fedlet in AM's *SAML v2.0 guide*.

1. Copy and unzip the fedlet zip file, `Fedlet-7.4.0.zip`, delivered with the AM installation, into a local directory.

```
$ unzip $HOME/openam/Fedlet-7.4.0.zip

Archive:  Fedlet-7.4.0.zip
creating: conf/
inflating: README
inflating: conf/FederationConfig.properties
inflating: conf/fedlet.cot-template
inflating: conf/idp-extended.xml-template
inflating: conf/sp-extended.xml-template
inflating: conf/sp.xml-template
inflating: fedlet.war
```

2. In each file, search and replace the following properties:

Replace this	With this
`IDP_ENTITY_ID`	`openam`
`FEDLET_ENTITY_ID`	`sp`
`FEDLET_PROTOCOL://FEDLET_HOST:FEDLET_PORT/FEDLET_DEPLOY_URI`	`http://sp.example.com:8080/saml`
`fedletcot` and `FEDLET_COT`	`Circle of Trust`
`sp.example.com:8080/saml/fedletapplication`	`sp.example.com:8080/saml/fedletapplication/metaAlias/sp`

3. Save the files as .xml, without the `-template` extension, so that the directory looks like this:

```
conf
├── FederationConfig.properties
├── fedlet.cot
├── idp-extended.xml
├── sp-extended.xml
└── sp.xml
```

By default, AM as an IDP uses the NameID format `urn:oasis:names:tc:SAML:2.0:nameid-format:transient` to communicate about a user. For information about using a different NameID format, refer to Use a non-transient NameID format.

3. Set up AM:

    1. In the AM admin UI, select **Identities**, select the user `demo`, and change the last name to `Ch4ng31t`. Note that, for this example, the last name must be the same as the password.
    2. Select **Applications** > **Federation** > **Circles of Trust**, and add a circle of trust called `Circle of Trust`, with the default settings.
    3. Set up a remote service provider:
        1. Select **Applications** > **Federation** > **Entity Providers**, and add a remote entity provider.
        2. Drag in or import `sp.xml` created in the previous step.
        3. Select **Circles of Trust**: `Circle of Trust`.
    4. Set up a hosted identity provider:
        1. Select **Applications** > **Federation** > **Entity Providers**, and add a hosted entity provider with the following values:
            - **Entity ID**: `openam`
            - **Entity Provider Base URL**: `http://am.example.com:8088/openam`
            - **Identity Provider Meta Alias**: `idp`
            - **Circles of Trust**: `Circle of Trust`
        2. Select **Assertion Processing** > **Attribute Mapper**, map the following SAML attribute keys and values, and then save your changes:
            - **SAML Attribute**: `cn`, **Local Attribute**: `cn`
            - **SAML Attribute**: `sn`, **Local Attribute**: `sn`
        3. In a terminal, export the XML-based metadata for the IDP:

            ```
            $ curl -v \
            --output idp.xml \
            "http://am.example.com:8088/openam/saml2/jsp/exportmetadata.jsp?entityid=openam"
            ```

            The `idp.xml` file is created locally.

4. Set up IG:

   1. Copy the edited fedlet files, and the exported `idp.xml` file into the IG configuration, at `$HOME/.openig/SAML`.

      ```
      $ ls -l $HOME/.openig/SAML

      FederationConfig.properties
      fedlet.cot
      idp-extended.xml
      idp.xml
      sp-extended.xml
      sp.xml
      ```

   2. In `config.json`, comment out or remove the `baseURI`:

      ```
      {
        "handler": {
          "_baseURI": "http://app.example.com:8081",
          ...
        }
      }
      ```

      Requests to the SamlFederationHandler must not be rebased, because the request URI must match the endpoint in the SAML metadata.

   3. Add the following route to IG to serve the sample application .css and other static resources:

      Linux

      ```
      $HOME/.openig/config/routes/00-static-resources.json
      ```

      Windows

      ```
      %appdata%\OpenIG\config\routes\00-static-resources.json
      ```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

4. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/saml-handler.json
```

Windows

```
%appdata%\OpenIG\config\routes\saml-handler.json
```

```
{
  "name": "saml-handler",
  "condition": "${find(request.uri.path, '^/saml')}",
  "session": "JwtSession",
  "handler": {
    "type": "SamlFederationHandler",
    "config": {
      "useOriginalUri": true,
      "assertionMapping": {
        "username": "cn",
        "password": "sn"
      },
      "subjectMapping": "sp-subject-name",
      "redirectURI": "/home/federate"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/saml`.
- After authentication, the SamlFederationHandler extracts `cn` and `sn` from the SAML assertion, and maps them to the SessionContext, at `session.username` and `session.password`.

- The handler stores the subject name as a string in the session field `session.sp-subject-name`, which is named by the `subjectMapping` property. By default, the subject name is stored in the session field `session.subjectName`.
- The handler redirects the request to the `/federate` route.
- The route uses the `JwtSession` implementation, meaning it stores encrypted session information in a browser cookie. The name is a reference to the `JwtSession` object defined in `config.json`. For information, see JwtSession.

5. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/federate-handler.json
```

Windows

```
%appdata%\OpenIG\config\routes\federate-handler.json
```

```
{
  "name": "federate-handler",
  "condition": "${find(request.uri.path, '^/home/federate')}",
  "session": "JwtSession",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${empty session.username}",
          "handler": {
            "type": "StaticResponseHandler",
            "config": {
              "status": 302,
              "headers": {
                "Location": [
                  "http://sp.example.com:8080/saml/SPInitiatedSSO?metaAlias=/sp"
                ]
              }
            }
          }
        },
        {
          "handler": {
            "type": "Chain",
            "config": {
              "filters": [
                {
                  "type": "HeaderFilter",
                  "config": {
                    "messageType": "REQUEST",
                    "add": {
                      "x-username": ["${session.username[0]}"],
                      "x-password": ["${session.password[0]}"]
                    }
                  }
                }
              ],
              "handler": "ReverseProxyHandler"
            }
          }
        }
      ]
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/home/federate`.
- If the user is not authenticated with AM, the username is not populated in the context. The DispatchHandler then dispatches the request to the StaticResponseHandler, which redirects it to the SP-initiated SSO endpoint.

  If the credentials are in the context, or after successful authentication, the DispatchHandler dispatches the request to the Chain.

- The HeaderFilter adds headers for the first value for the `username` and `password` attributes of the SAML assertion.
- The route uses the `JwtSession` implementation, meaning it stores encrypted session information in a browser cookie. The name is a reference to the `JwtSession` object defined in `config.json`. For information, see JwtSession.

6. Restart IG.

1. Test the setup:

   1. Log out of AM, and test the setup with the following links:

      - IDP-initiated SSO
      - SP-initiated SSO

   2. Log in to AM with username `demo` and password `Ch4ng31t`.

      IG returns the response page showing that the the demo user has logged in.

Tip

For more control over the URL where the user agent is redirected, use the `RelayState` query string parameter in the URL of the redirect `Location` header. `RelayState` specifies where to redirect the user when the SAML 2.0 web browser SSO process is complete. It overrides the `redirectURI` set in the SamlFederationHandler.
The `RelayState` value must be URL-encoded. When using an expression, use a function to encode the value. For example, use `${urlEncodeQueryParameterNameOrValue(contexts.router.originalUri)}`.
In the following example, the user is finally redirected to the original URI from the request:

```
"headers": {
  "Location": [
    "http://ig.example.com:8080/saml/SPInitiatedSSO?RelayState=$
{urlEncodeQueryParameterNameOrValue(contexts.router.originalUri)}"
  ]
}
```

Set up federation with signed/encrypted assertions with the SamlFederationHandler

1. Set up the example in Set up federation with unsigned/unencrypted assertions with the SamlFederationHandler.
2. Set up the SAML keystore:

   1. Find the values of AM's default SAML keypass and storepass:

      ```
      $ more /path/to/am/secrets/default/.keypass
      $ more /path/to/am/secrets/default/.storepass
      ```

   2. Copy the SAML keystore from the AM configuration to IG:

      ```
      $ cp /path/to/am/secrets/keystores/keystore.jceks /path/to/ig/secrets/keystore.jceks
      ```

> **Warning**
>
> Legacy keystore types such as JKS and JCEKS are supported but are not secure. Consider using the PKCS#12 keystore type.

3. Configure the Fedlet in IG:

    1. In `FederationConfig.properties`, make the following changes:

        1. Delete the following lines:

            - `com.sun.identity.saml.xmlsig.keystore=%BASE_DIR%/security/keystores/keystore.jks`
            - `com.sun.identity.saml.xmlsig.storepass=%BASE_DIR%/.storepass`
            - `com.sun.identity.saml.xmlsig.keypass=%BASE_DIR%/.keypass`
            - `com.sun.identity.saml.xmlsig.certalias=test`
            - `com.sun.identity.saml.xmlsig.storetype=JKS`
            - `am.encryption.pwd=@AM_ENC_PWD@`

        2. Add the following line:

            `org.forgerock.openam.saml2.credential.resolver.class=org.forgerock.openig.handler.saml.SecretsSaml2CredentialResolver`

            This class is responsible for resolving secrets and supplying credentials.

            > **Tip**
            >
            > Be sure to leave no space at the end of the line.

    2. In `sp.xml`, make the following changes:

        1. Change `AuthnRequestsSigned="false"` to `AuthnRequestsSigned="true"`.

        2. Add the following KeyDescriptor just before `</SPSSODescriptor>`

```
        <KeyDescriptor use="signing">
            <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#" >
                <ds:X509Data>
                    <ds:X509Certificate>

                    </ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </KeyDescriptor>
    </SPSSODescriptor>
```

        3. Copy the value of the signing certificate from `idp.xml` to this file:

```
<KeyDescriptor use="signing">
  <ds:KeyInfo>
    <ds:X509Data>
      <ds:X509Certificate>

        MII...zA6

      </ds:X509Certificate>
```

This is the public key used for signing so that the IDP can verify request signatures.

4. Replace the remote service provider in AM:

    1. Select **Applications** > **Federation** > **Entity Providers**, and remove the `sp` entity provider.
    2. Drag in or import the new `sp.xml` updated in the previous step.
    3. Select **Circles of Trust**: `Circle of Trust`.

5. Set up IG:

    1. In the IG configuration, set environment variables for the following secrets, and then restart IG:

    ```
    $ export KEYSTORE_SECRET_ID='a2V5c3RvcmU='
    $ export SAML_KEYSTORE_STOREPASS_SECRET_ID='base64-encoded value of the SAML storepass'
    $ export SAML_KEYSTORE_KEYPASS_SECRET_ID='base64-encoded value of the SAML keypass'
    ```

    The passwords are retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

    2. Remove `saml-handler.json` from the configuration, and add the following route, replacing the path to `keystore.jceks` with your path:

    Linux

    ```
    $HOME/.openig/config/routes/saml-handler-secure.json
    ```

    Windows

    ```
    %appdata%\OpenIG\config\routes\saml-handler-secure.json
    ```

```
{
  "name": "saml-handler-secure",
  "condition": "${find(request.uri.path, '^/saml')}",
  "session": "JwtSession",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "KeyStoreSecretStore-1",
      "type" : "KeyStoreSecretStore",
      "config" : {
        "file" : "/path/to/ig/keystore.jceks",
        "storeType" : "jceks",
        "storePasswordSecretId" : "saml.keystore.storepass.secret.id",
        "entryPasswordSecretId" : "saml.keystore.keypass.secret.id",
        "secretsProvider" : "SystemAndEnvSecretStore-1",
        "mappings" : [ {
          "secretId" : "sp.signing.sp",
          "aliases" : [ "rsajwtsigningkey" ]
        }, {
          "secretId" : "sp.decryption.sp",
          "aliases" : [ "test" ]
        } ]
      }
    }
  ],
  "handler": {
    "type": "SamlFederationHandler",
    "config": {
      "useOriginalUri": true,
      "assertionMapping": {
        "username": "cn",
        "password": "sn"
      },
      "subjectMapping": "sp-subject-name",
      "redirectURI": "/home/federate",
      "secretsProvider" : "KeyStoreSecretStore-1"
    }
  }
}
```

Notice the following features of the route compared to `saml-handler.json`:

- The SamlFederationHandler refers to the KeyStoreSecretStore to provide the keys for the signed and encrypted SAML assertions.
- The secret IDs, `sp.signing.sp` and `sp.decryption.sp`, follow a naming convention based on the name of the service provider, `sp`.
- The alias for the signing key corresponds to the PEM in `keystore.jceks`.

3. Restart IG.

6. Test the setup:

    1. Log out of AM, and test the setup with the following links:

        - IDP-initiated SSO
        - SP-initiated SSO

    2. Log in to AM with username `demo` and password `Ch4ng31t`.

        IG returns the response page showing that the the demo user has logged in.

SAML 2.0 and multiple applications with the SamlFederationHandler

The chapter extends the example in SAML to add a second service provider.

The new service provider has entity ID `sp2` and runs on the host `sp2.example.com`. To prevent unwanted behavior, the service providers must have different values.

1. Add `sp2.example.com` to your `/etc/hosts` file:

```
127.0.0.1 localhost am.example.com ig.example.com app.example.com sp.example.com sp2.example.com
```

2. In IG, configure the service provider files for `sp2`, using the files you created in Configure a Java Fedlet::

    1. In `fedlet.cot`, add `sp2` to the list of sun-fm-trusted-providers:

```
cot-name=Circle of Trust
sun-fm-cot-status=Active
sun-fm-trusted-providers=openam, sp, sp2
sun-fm-saml2-readerservice-url=
sun-fm-saml2-writerservice-url=
```

    2. Copy `sp.xml` to `sp2.xml`, and copy `sp-extended.xml` to `sp2-extended.xml`.

    3. In both files, search and replace the following strings:

        - `entityID=sp`: replace with `entityID=sp2`
        - `sp.example.com`: replace with `sp2.example.com`
        - `metaAlias=/sp`: replace with `metaAlias=/sp2`
        - `/metaAlias/sp`: replace with `/metaAlias/sp2`

    4. Restart IG.

3. In AM, set up a remote service provider for `sp2`:

    1. Select **Applications** > **Federation** > **Entity Providers**.

    2. Drag in or import `sp2.xml` created in the previous step.

    3. Select **Circles of Trust**: `Circle of Trust`.

4. Add the following routes to IG:

Linux

```
$HOME/.openig/config/routes/saml-handler-sp2.json
```

Windows

```
%appdata%\OpenIG\config\routes\saml-handler-sp2.json
```

```
{
  "name": "saml-handler-sp2",
  "condition": "${find(request.uri.host, 'sp2.example.com') and find(request.uri.path, '^/saml')}",
  "handler": {
    "type": "SamlFederationHandler",
    "config": {
      "comment": "Use unique session properties for this SP.",
      "useOriginalUri": true,
      "assertionMapping": {
        "sp2Username": "cn",
        "sp2Password": "sn"
      },
      "authnContext": "sp2AuthnContext",
      "sessionIndexMapping": "sp2SessionIndex",
      "subjectMapping": "sp2SubjectName",
      "redirectURI": "/sp2"
    }
  }
}
```

Linux

```
$HOME/.openig/config/routes/federate-handler-sp2.json
```

Windows

```
%appdata%\OpenIG\config\routes\federate-handler-sp2.json
```

```
{
  "name": "federate-handler-sp2",
  "condition": "${find(request.uri.host, 'sp2.example.com') and not find(request.uri.path, '^/saml')}",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${empty session.sp2Username}",
          "handler": {
            "type": "StaticResponseHandler",
            "config": {
              "status": 302,
              "headers": {
                "Location": [
                  "http://sp2.example.com:8080/saml/SPInitiatedSSO?metaAlias=/sp2"
                ]
              }
            }
          }
        },
        {
          "handler": {
            "type": "Chain",
            "config": {
              "filters": [
                {
                  "type": "HeaderFilter",
                  "config": {
                    "messageType": "REQUEST",
                    "add": {
                      "x-username": ["${session.sp2Username[0]}"],
                      "x-password": ["${session.sp2Password[0]}"]
                    }
                  }
                }
              ],
              "handler": "ReverseProxyHandler"
            }
          }
        }
      ]
    }
  }
}
```

5. Test the setup:

   1. Log out of AM, and test the setup with the following links:

      - IDP-initiated SSO
      - SP-initiated SSO

   2. Log in to AM with username `demo` and password `Ch4ng31t`.

      IG returns the response page showing that the user has logged in.

Use a non-transient NameID format

By default, AM as an IDP uses the NameID format `urn:oasis:names:tc:SAML:2.0:nameid-format:transient`. For more information, refer to Hosted identity provider configuration properties in AM's *SAML v2.0 guide*.

When the IDP uses another NameID format, configure IG to use that NameID format by editing the Fedlet configuration file `sp-extended.xml`:

- To use the NameID value provided by the IDP, add the following attribute:

```
<Attribute name="useNameIDAsSPUserID">
  <Value>true</Value>
</Attribute>
```

- To use an attribute from the assertion, add the following attribute:

```
<Attribute name="autofedEnabled">
  <Value>true</Value>
</Attribute>
<Attribute name="autofedAttribute">
  <Value>sn</Value>
</Attribute>
```

  This example uses the value in `SN` to identify the subject.

Although IG supports the `persistent` NameID format, IG does not store the mapping. To configure this behavior, edit the file `sp-extended.xml`:

- To disable attempts to persist the user mapping, add the following attribute:

```
<Attribute name="spDoNotWriteFederationInfo">
  <Value>true</Value>
</Attribute>
```

- To enable attempts to persist the user mapping, add the following attribute:

```
<Attribute name="spDoNotWriteFederationInfo">
  <Value>false</Value>
</Attribute>
```

If a login request doesn't contain a NameID format query parameter, the value is defined by the presence and content of the NameID format list for the SP and IDP. For example, an SP-initiated login can be constructed with the binding and `NameIDFormat` as a parameter, as follows:

```
http://fedlet.example.org:7070/fedlet/SPInitiatedSSO?binding=urn:oasis:names:tc:SAML:2.0:bindings:HTTP-
POST&NameIDFormat=urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified
```

When the NameID format is provided in a list, it is resolved as follows:

- If both the IDP and SP have a list, the first matching NameID format in the lists.
- If either the IDP or SP list is empty, the first NameID format in the other list.
- If neither the IDP nor SP has a list, then AM defaults to `transient`, and IG defaults to `persistent`.

Example fedlet files

File	Description
`FederationConfig.properties`	Fedlet properties
`fedlet.cot`	Circle of trust for IG and the IDP
`idp.xml`	Standard metadata for the IDP
`idp-extended.xml`	Metadata extensions for the IDP
`sp.xml`	Standard metadata for the IG SP
`sp-extended.xml`	Metadata extensions for the IG SP

AM as the SAML IDP

FederationConfig.properties

The following example of `$HOME/.openig/SAML/FederationConfig.properties` defines the fedlet properties:

```
#
# DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.
#
# Copyright (c) 2006 Sun Microsystems Inc. All Rights Reserved
#
# The contents of this file are subject to the terms
# of the Common Development and Distribution License
# (the License). You may not use this file except in
# compliance with the License.
#
# You can obtain a copy of the License at
# https://opensso.dev.java.net/public/CDDLv1.0.html or
# opensso/legal/CDDLv1.0.txt
# See the License for the specific language governing
# permission and limitations under the License.
#
# When distributing Covered Code, include this CDDL
# Header Notice in each file and include the License file
# at opensso/legal/CDDLv1.0.txt.
# If applicable, add the following below the CDDL Header,
# with the fields enclosed by brackets [] replaced by
# your own identifying information:
# "Portions Copyrighted [year] [name of copyright owner]"
#
# $Id: FederationConfig.properties,v 1.21 2010/01/08 22:41:28 exu Exp $
#
# Portions Copyright 2016-2023 ForgeRock AS.

# If a component wants to use a different datastore provider than the
# default one defined above, it can define a property like follows:
# com.sun.identity.plugin.datastore.class.<componentName>=<provider class>

# com.sun.identity.plugin.configuration.class specifies implementation for
# com.sun.identity.plugin.configuration.ConfigurationInstance interface.
com.sun.identity.plugin.configuration.class=com.sun.identity.plugin.configuration.impl.FedletConfigurationImpl

# Specifies implementation for
# com.sun.identity.plugin.datastore.DataStoreProvider interface.
# This property defines the default datastore provider.
com.sun.identity.plugin.datastore.class.default=com.sun.identity.plugin.datastore.impl.FedletDataStoreProvider

# Specifies implementation for
# org.forgerock.openam.federation.plugin.rooturl.RootUrlProvider interface.
# This property defines the default base url provider.
com.sun.identity.plugin.root.url.class.default=org.forgerock.openam.federation.plugin.rooturl.impl.FedletRootUrlProvi
der

# com.sun.identity.plugin.log.class specifies implementation for
# com.sun.identity.plugin.log.Logger interface.
com.sun.identity.plugin.log.class=com.sun.identity.plugin.log.impl.FedletLogger

# com.sun.identity.plugin.session.class specifies implementation for
# com.sun.identity.plugin.session.SessionProvider interface.
com.sun.identity.plugin.session.class=com.sun.identity.plugin.session.impl.FedletSessionProvider

# com.sun.identity.plugin.monitoring.agent.class specifies implementation for
# com.sun.identity.plugin.monitoring.FedMonAgent interface.
com.sun.identity.plugin.monitoring.agent.class=com.sun.identity.plugin.monitoring.impl.FedletAgentProvider

# com.sun.identity.plugin.monitoring.saml2.class specifies implementation for
```

```
# com.sun.identity.plugin.monitoring.FedMonSAML2Svc interface.
com.sun.identity.plugin.monitoring.saml2.class=com.sun.identity.plugin.monitoring.impl.FedletMonSAML2SvcProvider

# com.sun.identity.saml.xmlsig.keyprovider.class specified the implementation
# class for com.sun.identity.saml.xmlsig.KeyProvider interface
com.sun.identity.saml.xmlsig.keyprovider.class=com.sun.identity.saml.xmlsig.JKSKeyProvider

# com.sun.identity.saml.xmlsig.signatureprovider.class specified the
# implementation class for com.sun.identity.saml.xmlsig.SignatureProvider
# interface
com.sun.identity.saml.xmlsig.signatureprovider.class=com.sun.identity.saml.xmlsig.AMSignatureProvider

com.iplanet.am.server.protocol=http
com.iplanet.am.server.host=am.example.com
com.iplanet.am.server.port=8080
com.iplanet.am.services.deploymentDescriptor=/openam
com.iplanet.am.logstatus=ACTIVE

# Name of the webcontainer.
# Even though the servlet/JSP are web container independent,
# Access/Federation Manager uses servlet 2.3 API request.setCharacterEncoding()
# to decode incoming non English characters. These APIs will not work if
# Access/Federation Manager is deployed on Sun Java System Web Server 6.1.
# We use gx_charset mechanism to correctly decode incoming data in
# Sun Java System Web Server 6.1 and S1AS7.0. Possible values
# are BEA6.1, BEA 8.1, IBM5.1 or IAS7.0.
# If the web container is Sun Java System Webserver, the tag is not replaced.
com.sun.identity.webcontainer=WEB_CONTAINER

# Identify saml xml signature keystore file, keystore password file
# key password file
com.sun.identity.saml.xmlsig.keystore=%BASE_DIR%/security/keystores/keystore.jks
com.sun.identity.saml.xmlsig.storepass=%BASE_DIR%/.storepass
com.sun.identity.saml.xmlsig.keypass=%BASE_DIR%/.keypass
com.sun.identity.saml.xmlsig.certalias=test

# Type of keystore used for saml xml signature. Default is JKS.
#
# com.sun.identity.saml.xmlsig.storetype=JKS

# Specifies the implementation class for
# com.sun.identity.saml.xmlsig.PasswordDecoder interface.
com.sun.identity.saml.xmlsig.passwordDecoder=com.sun.identity.fedlet.FedletEncodeDecode

# The following key is used to specify the maximum content-length
# for an HttpRequest that will be accepted by the OpenSSO
# The default value is 16384 which is 16k
com.iplanet.services.comm.server.pllrequest.maxContentLength=16384

# The following keys are used to configure the Debug service.
# Possible values for the key 'level' are: off | error | warning | message.
# The key 'directory' specifies the output directory where the debug files
# will be created.
# Trailing spaces are significant.
# Windows: Use forward slashes "/" separate directories, not backslash "\".
# Windows: Spaces in the file name are allowed for Windows.
#
com.iplanet.services.debug.level=message
com.iplanet.services.debug.directory=%BASE_DIR%%SERVER_URI%/debug

# The following keys are used to configure the Stats service.
# Possible values for the key 'level' are: off | file | console
```

```
# Stats state 'file' will write to a file under the specified directory,
# and 'console' will write into  webserver log files
# The key 'directory' specifies the output directory where the debug files
# will be created.
# Trailing spaces are significant.
# Windows: Use forward slashes "/" separate directories, not backslash "\".
# Windows: Spaces in the file name are allowed for Windows.
# Stats interval should be atleast 5 secs to avoid CPU saturation,
# the product would assume any thing less than 5 secs is 5 secs.
com.iplanet.am.stats.interval=60
com.iplanet.services.stats.state=file
com.iplanet.services.stats.directory=%BASE_DIR%/var/stats

# The key that will be used to encrypt and decrypt passwords.
am.encryption.pwd=@AM_ENC_PWD@

# SecureRandom Properties: The key
# "com.iplanet.security.SecureRandomFactoryImpl"
# specifies the factory class name for SecureRandomFactory
# Available impl classes are:
#   com.iplanet.am.util.JSSSecureRandomFactoryImpl (uses JSS)
#   com.iplanet.am.util.SecureRandomFactoryImpl (pure Java)
com.iplanet.security.SecureRandomFactoryImpl=com.iplanet.am.util.SecureRandomFactoryImpl

# SocketFactory properties: The key "com.iplanet.security.SSLSocketFactoryImpl"
# specifies the factory class name for LDAPSocketFactory
# Available classes are:
#    com.iplanet.services.ldap.JSSSocketFactory (uses JSS)
#    com.sun.identity.shared.ldap.factory.JSSESocketFactory    (pure Java)
com.iplanet.security.SSLSocketFactoryImpl=com.sun.identity.shared.ldap.factory.JSSESocketFactory

# Encryption: The key "com.iplanet.security.encryptor" specifies
# the encrypting class implementation.
# Available classes are:
#    com.iplanet.services.util.JCEEncryption
#    com.iplanet.services.util.JSSEncryption
com.iplanet.security.encryptor=com.iplanet.services.util.JCEEncryption

# Determines if JSS will be added with highest priority to JCE
# Set this to "true" if other JCE providers should be used for
# digial signatures and encryptions.
com.sun.identity.jss.donotInstallAtHighestPriority=true

# Configuration File (serverconfig.xml) Location
com.iplanet.services.configpath=@BASE_DIR@
```

fedlet.cot

The following example of `$HOME/.openig/SAML/fedlet.cot` defines a circle of trust between AM as the IDP, and IG as the SP:

```
cot-name=Circle of Trust
sun-fm-cot-status=Active
sun-fm-trusted-providers=openam, sp
sun-fm-saml2-readerservice-url=
sun-fm-saml2-writerservice-url=
```

idp.xml

The following example of `$HOME/.openig/SAML/idp.xml` defines a SAML configuration file for the AM IDP, `idp`:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<EntityDescriptor entityID="openam" xmlns="urn:oasis:names:tc:SAML:2.0:metadata"
xmlns:query="urn:oasis:names:tc:SAML:metadata:ext:query" xmlns:mdattr="urn:oasis:names:tc:SAML:metadata:attribute"
xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
xmlns:xenc11="http://www.w3.org/2009/xmlenc11#" xmlns:alg="urn:oasis:names:tc:SAML:metadata:algsupport"
xmlns:x509qry="urn:oasis:names:tc:SAML:metadata:X509:query" xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <IDPSSODescriptor protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
        <KeyDescriptor use="signing">
            <ds:KeyInfo>
                <ds:X509Data>
                    <ds:X509Certificate>
...
                    </ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
        </KeyDescriptor>
        <KeyDescriptor use="encryption">
            <ds:KeyInfo>
                <ds:X509Data>
                    <ds:X509Certificate>
...
                    </ds:X509Certificate>
                </ds:X509Data>
            </ds:KeyInfo>
            <EncryptionMethod Algorithm="http://www.w3.org/2009/xmlenc11#rsa-oaep">
                <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
                <xenc11:MGF Algorithm="http://www.w3.org/2009/xmlenc11#mgf1sha256"/>
            </EncryptionMethod>
            <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc">
                <xenc:KeySize>128</xenc:KeySize>
            </EncryptionMethod>
        </KeyDescriptor>
        <ArtifactResolutionService index="0" Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://
am.example.com:8088/openam/ArtifactResolver/metaAlias/idp"/>
        <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect" Location="http://
am.example.com:8088/openam/IDPSloRedirect/metaAlias/idp" ResponseLocation="http://am.example.com:8088/openam/
IDPSloRedirect/metaAlias/idp"/>
        <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="http://
am.example.com:8088/openam/IDPSloPOST/metaAlias/idp" ResponseLocation="http://am.example.com:8088/openam/IDPSloPOST/
metaAlias/idp"/>
        <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://am.example.com:
8088/openam/IDPSloSoap/metaAlias/idp"/>
        <ManageNameIDService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect" Location="http://
am.example.com:8088/openam/IDPMniRedirect/metaAlias/idp" ResponseLocation="http://am.example.com:8088/openam/
IDPMniRedirect/metaAlias/idp"/>
        <ManageNameIDService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="http://
am.example.com:8088/openam/IDPMniPOST/metaAlias/idp" ResponseLocation="http://am.example.com:8088/openam/IDPMniPOST/
metaAlias/idp"/>
        <ManageNameIDService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://am.example.com:
8088/openam/IDPMniSoap/metaAlias/idp"/>
        <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:persistent</NameIDFormat>
        <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:transient</NameIDFormat>
        <NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress</NameIDFormat>
        <NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified</NameIDFormat>
        <NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-format:WindowsDomainQualifiedName</NameIDFormat>
        <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:kerberos</NameIDFormat>
        <NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-format:X509SubjectName</NameIDFormat>
        <SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect" Location="http://
am.example.com:8088/openam/SSORedirect/metaAlias/idp"/>
        <SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="http://
```

```
am.example.com:8088/openam/SSOPOST/metaAlias/idp"/>
        <SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://am.example.com:
8088/openam/SSOSoap/metaAlias/idp"/>
        <NameIDMappingService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://am.example.com:
8088/openam/NIMSoap/metaAlias/idp"/>
        <AssertionIDRequestService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://
am.example.com:8088/openam/AIDReqSoap/IDPRole/metaAlias/idp"/>
        <AssertionIDRequestService Binding="urn:oasis:names:tc:SAML:2.0:bindings:URI" Location="http://
am.example.com:8088/openam/AIDReqUri/IDPRole/metaAlias/idp"/>
    </IDPSSODescriptor>
</EntityDescriptor>
```

idp-extended.xml

The following example of `$HOME/.openig/SAML/idp-extended.xml` defines an AM-specific SAML descriptor file for the IDP:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!--
   DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.

   Copyright (c) 2002-2010 Sun Microsystems Inc. All Rights Reserved

   The contents of this file are subject to the terms
   of the Common Development and Distribution License
   (the License). You may not use this file except in
   compliance with the License.

   You can obtain a copy of the License at
   https://opensso.dev.java.net/public/CDDLv1.0.html or
   opensso/legal/CDDLv1.0.txt
   See the License for the specific language governing
   permission and limitations under the License.

   When distributing Covered Code, include this CDDL
   Header Notice in each file and include the License file
   at opensso/legal/CDDLv1.0.txt.
   If applicable, add the following below the CDDL Header,
   with the fields enclosed by brackets [] replaced by
   your own identifying information:
   "Portions Copyrighted [year] [name of copyright owner]"

   Portions Copyrighted 2010-2017 ForgeRock AS.
-->
<EntityConfig entityID="openam" hosted="0" xmlns="urn:sun:fm:SAML:2.0:entityconfig">
    <IDPSSOConfig>
        <Attribute name="description">
            <Value/>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </IDPSSOConfig>
    <AttributeAuthorityConfig>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </AttributeAuthorityConfig>
    <XACMLPDPConfig>
        <Attribute name="wantXACMLAuthzDecisionQuerySigned">
            <Value></Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </XACMLPDPConfig>
</EntityConfig>
```

sp.xml

Note

The SAML library component validates the SP's AssertionConsumerService Location against the incoming IDP SAML Assertion, based on the request information, including the port. Always specify the port in the Location value of `AssertionConsumerService`, even when using defaults of 443 or 80, as follows:

```
<AssertionConsumerService isDefault="true"
                          index="0"
                          Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
                          Location="https://sp.example.com:443/fedletapplication" />
```

The following example of `$HOME/.openig/SAML/sp.xml` defines a SAML configuration file for the IG SP, `sp`.

```
<!--
   DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.

   Copyright (c) 2002-2010 Sun Microsystems Inc. All Rights Reserved

   The contents of this file are subject to the terms
   of the Common Development and Distribution License
   (the License). You may not use this file except in
   compliance with the License.

   You can obtain a copy of the License at
   https://opensso.dev.java.net/public/CDDLv1.0.html or
   opensso/legal/CDDLv1.0.txt
   See the License for the specific language governing
   permission and limitations under the License.

   When distributing Covered Code, include this CDDL
   Header Notice in each file and include the License file
   at opensso/legal/CDDLv1.0.txt.
   If applicable, add the following below the CDDL Header,
   with the fields enclosed by brackets [] replaced by
   your own identifying information:
   "Portions Copyrighted [year] [name of copyright owner]"

   Portions Copyrighted 2010-2017 ForgeRock AS.
-->
<EntityDescriptor entityID="sp" xmlns="urn:oasis:names:tc:SAML:2.0:metadata">
    <SPSSODescriptor AuthnRequestsSigned="false" WantAssertionsSigned="false"
protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
        <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect" Location="http://
sp.example.com:8080/saml/fedletSloRedirect" ResponseLocation="http://sp.example.com:8080/saml/fedletSloRedirect"/>
        <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="http://
sp.example.com:8080/saml/fedletSloPOST" ResponseLocation="http://sp.example.com:8080/saml/fedletSloPOST"/>
        <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://sp.example.com:
8080/saml/fedletSloSoap"/>
        <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:transient</NameIDFormat>
        <AssertionConsumerService isDefault="true" index="0" Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
Location="http://sp.example.com:8080/saml/fedletapplication/metaAlias/sp"/>
        <AssertionConsumerService index="1" Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Artifact"
Location="http://sp.example.com:8080/saml/fedletapplication/metaAlias/sp"/>
    </SPSSODescriptor>
    <RoleDescriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:query="urn:oasis:names:tc:SAML:metadata:ext:query" xsi:type="query:AttributeQueryDescriptorType"
protocolSupportEnumeration= "urn:oasis:names:tc:SAML:2.0:protocol">
    </RoleDescriptor>
    <XACMLAuthzDecisionQueryDescriptor WantAssertionsSigned="false"
protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
    </XACMLAuthzDecisionQueryDescriptor>
</EntityDescriptor>
```

sp-extended.xml

The following example of `$HOME/.openig/SAML/sp-extended.xml` defines an AM-specific SAML descriptor file for the SP:

```
<!--
   DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.

   Copyright (c) 2002-2010 Sun Microsystems Inc. All Rights Reserved

   The contents of this file are subject to the terms
   of the Common Development and Distribution License
   (the License). You may not use this file except in
   compliance with the License.

   You can obtain a copy of the License at
   https://opensso.dev.java.net/public/CDDLv1.0.html or
   opensso/legal/CDDLv1.0.txt
   See the License for the specific language governing
   permission and limitations under the License.

   When distributing Covered Code, include this CDDL
   Header Notice in each file and include the License file
   at opensso/legal/CDDLv1.0.txt.
   If applicable, add the following below the CDDL Header,
   with the fields enclosed by brackets [] replaced by
   your own identifying information:
   "Portions Copyrighted [year] [name of copyright owner]"

   Portions Copyrighted 2010-2017 ForgeRock AS.
-->
<EntityConfig xmlns="urn:sun:fm:SAML:2.0:entityconfig" xmlns:fm="urn:sun:fm:SAML:2.0:entityconfig" hosted="1"
entityID="sp">
    <SPSSOConfig metaAlias="/sp">
        <Attribute name="description">
            <Value></Value>
        </Attribute>
        <Attribute name="signingCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthOn">
            <Value>false</Value>
        </Attribute>
        <Attribute name="basicAuthUser">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthPassword">
            <Value></Value>
        </Attribute>
        <Attribute name="autofedEnabled">
            <Value>false</Value>
        </Attribute>
        <Attribute name="autofedAttribute">
            <Value></Value>
        </Attribute>
        <Attribute name="transientUser">
            <Value>anonymous</Value>
        </Attribute>
        <Attribute name="spAdapter">
            <Value></Value>
        </Attribute>
        <Attribute name="spAdapterEnv">
```

```
            <Value></Value>
        </Attribute>
        <Attribute name="fedletAdapter">
            <Value></Value>
        </Attribute>
        <Attribute name="fedletAdapterEnv">
            <Value></Value>
        </Attribute>
        <Attribute name="spAccountMapper">
            <Value>com.sun.identity.saml2.plugins.DefaultLibrarySPAccountMapper</Value>
        </Attribute>
        <Attribute name="spAttributeMapper">
            <Value>com.sun.identity.saml2.plugins.DefaultSPAttributeMapper</Value>
        </Attribute>
        <Attribute name="spAuthncontextMapper">
            <Value>com.sun.identity.saml2.plugins.DefaultSPAuthnContextMapper</Value>
        </Attribute>
        <Attribute name="spAuthncontextClassrefMapping">
            <Value>urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport|0|default</Value>
        </Attribute>
        <Attribute name="spAuthncontextComparisonType">
           <Value>exact</Value>
        </Attribute>
        <Attribute name="attributeMap">
           <Value>*=*</Value>
        </Attribute>
        <Attribute name="saml2AuthModuleName">
           <Value></Value>
        </Attribute>
        <Attribute name="localAuthURL">
            <Value></Value>
        </Attribute>
        <Attribute name="intermediateUrl">
            <Value></Value>
        </Attribute>
        <Attribute name="defaultRelayState">
            <Value></Value>
        </Attribute>
        <Attribute name="appLogoutUrl">
            <Value>http://sp.example.com:8080/saml/logout</Value>
        </Attribute>
        <Attribute name="assertionTimeSkew">
            <Value>300</Value>
        </Attribute>
        <Attribute name="wantAttributeEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="wantAssertionEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="wantNameIDEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="wantPOSTResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantArtifactResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantLogoutRequestSigned">
            <Value></Value>
        </Attribute>
```

```
        <Attribute name="wantLogoutResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantMNIRequestSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantMNIResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value></Attribute>
        <Attribute name="saeAppSecretList">
        </Attribute>
        <Attribute name="saeSPUrl">
            <Value></Value>
        </Attribute>
        <Attribute name="saeSPLogoutUrl">
        </Attribute>
        <Attribute name="ECPRequestIDPListFinderImpl">
            <Value>com.sun.identity.saml2.plugins.ECPIDPFinder</Value>
        </Attribute>
        <Attribute name="ECPRequestIDPList">
            <Value></Value>
        </Attribute>
        <Attribute name="enableIDPProxy">
            <Value>false</Value>
        </Attribute>
        <Attribute name="idpProxyList">
            <Value></Value>
        </Attribute>
        <Attribute name="idpProxyCount">
            <Value>0</Value>
        </Attribute>
        <Attribute name="useIntroductionForIDPProxy">
            <Value>false</Value>
        </Attribute>
    </SPSSOConfig>
    <AttributeQueryConfig metaAlias="/attrQuery">
        <Attribute name="signingCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="wantNameIDEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </AttributeQueryConfig>
    <XACMLAuthzDecisionQueryConfig metaAlias="/pep">
        <Attribute name="signingCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthOn">
            <Value>false</Value>
        </Attribute>
        <Attribute name="basicAuthUser">
```

```
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthPassword">
            <Value></Value>
        </Attribute>
        <Attribute name="wantXACMLAuthzDecisionResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantAssertionEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
   </XACMLAuthzDecisionQueryConfig>
</EntityConfig>
```

PingOne as the SAML IDP

FederationConfig.properties

```
# If a component wants to use a different datastore provider than the
# default one defined above, it can define a property like follows:
# com.sun.identity.plugin.datastore.class.<componentName>=<provider class>

# com.sun.identity.plugin.configuration.class specifies implementation for
# com.sun.identity.plugin.configuration.ConfigurationInstance interface.
com.sun.identity.plugin.configuration.class=com.sun.identity.plugin.configuration.impl.FedletConfigurationImpl

# Specifies implementation for
# com.sun.identity.plugin.datastore.DataStoreProvider interface.
# This property defines the default datastore provider.
com.sun.identity.plugin.datastore.class.default=com.sun.identity.plugin.datastore.impl.FedletDataStoreProvider

# Specifies implementation for
# org.forgerock.openam.federation.plugin.rooturl.RootUrlProvider interface.
# This property defines the default base url provider.
com.sun.identity.plugin.root.url.class.default=org.forgerock.openam.federation.plugin.rooturl.impl.FedletRootUrlProvi
der

# com.sun.identity.plugin.log.class specifies implementation for
# com.sun.identity.plugin.log.Logger interface.
com.sun.identity.plugin.log.class=com.sun.identity.plugin.log.impl.FedletLogger

# com.sun.identity.plugin.session.class specifies implementation for
# com.sun.identity.plugin.session.SessionProvider interface.
com.sun.identity.plugin.session.class=com.sun.identity.plugin.session.impl.FedletSessionProvider

# com.sun.identity.plugin.monitoring.agent.class specifies implementation for
# com.sun.identity.plugin.monitoring.FedMonAgent interface.
com.sun.identity.plugin.monitoring.agent.class=com.sun.identity.plugin.monitoring.impl.FedletAgentProvider

# com.sun.identity.plugin.monitoring.saml2.class specifies implementation for
# com.sun.identity.plugin.monitoring.FedMonSAML2Svc interface.
com.sun.identity.plugin.monitoring.saml2.class=com.sun.identity.plugin.monitoring.impl.FedletMonSAML2SvcProvider

# com.sun.identity.saml.xmlsig.keyprovider.class specified the implementation
# class for com.sun.identity.saml.xmlsig.KeyProvider interface
com.sun.identity.saml.xmlsig.keyprovider.class=com.sun.identity.saml.xmlsig.JKSKeyProvider

# com.sun.identity.saml.xmlsig.signatureprovider.class specified the
# implementation class for com.sun.identity.saml.xmlsig.SignatureProvider
# interface
com.sun.identity.saml.xmlsig.signatureprovider.class=com.sun.identity.saml.xmlsig.AMSignatureProvider

com.iplanet.am.server.protocol=http
com.iplanet.am.server.host=am.example.com
com.iplanet.am.server.port=8080
com.iplanet.am.services.deploymentDescriptor=/openam
com.iplanet.am.logstatus=ACTIVE

# Name of the webcontainer.
# Even though the servlet/JSP are web container independent,
# Access/Federation Manager uses servlet 2.3 API request.setCharacterEncoding()
# to decode incoming non English characters. These APIs will not work if
# Access/Federation Manager is deployed on Sun Java System Web Server 6.1.
# We use gx_charset mechanism to correctly decode incoming data in
# Sun Java System Web Server 6.1 and S1AS7.0. Possible values
# are BEA6.1, BEA 8.1, IBM5.1 or IAS7.0.
# If the web container is Sun Java System Webserver, the tag is not replaced.
com.sun.identity.webcontainer=WEB_CONTAINER
```

```
# Identify saml xml signature keystore file, keystore password file
# key password file
com.sun.identity.saml.xmlsig.keystore=%BASE_DIR%/security/keystores/keystore.jks
com.sun.identity.saml.xmlsig.storepass=%BASE_DIR%/.storepass
com.sun.identity.saml.xmlsig.keypass=%BASE_DIR%/.keypass
com.sun.identity.saml.xmlsig.certalias=test

# Type of keystore used for saml xml signature. Default is JKS.
#
# com.sun.identity.saml.xmlsig.storetype=JKS

# Specifies the implementation class for
# com.sun.identity.saml.xmlsig.PasswordDecoder interface.
com.sun.identity.saml.xmlsig.passwordDecoder=com.sun.identity.fedlet.FedletEncodeDecode

# The following key is used to specify the maximum content-length
# for an HttpRequest that will be accepted by the OpenSSO
# The default value is 16384 which is 16k
com.iplanet.services.comm.server.pllrequest.maxContentLength=16384

# The following keys are used to configure the Debug service.
# Possible values for the key 'level' are: off | error | warning | message.
# The key 'directory' specifies the output directory where the debug files
# will be created.
# Trailing spaces are significant.
# Windows: Use forward slashes "/" separate directories, not backslash "\".
# Windows: Spaces in the file name are allowed for Windows.
#
com.iplanet.services.debug.level=message
com.iplanet.services.debug.directory=%BASE_DIR%%SERVER_URI%/debug

# The following keys are used to configure the Stats service.
# Possible values for the key 'level' are: off | file | console
# Stats state 'file' will write to a file under the specified directory,
# and 'console' will write into  webserver log files
# The key 'directory' specifies the output directory where the debug files
# will be created.
# Trailing spaces are significant.
# Windows: Use forward slashes "/" separate directories, not backslash "\".
# Windows: Spaces in the file name are allowed for Windows.
# Stats interval should be atleast 5 secs to avoid CPU saturation,
# the product would assume any thing less than 5 secs is 5 secs.
com.iplanet.am.stats.interval=60
com.iplanet.services.stats.state=file
com.iplanet.services.stats.directory=%BASE_DIR%/var/stats

# The key that will be used to encrypt and decrypt passwords.
am.encryption.pwd=@AM_ENC_PWD@

# SecureRandom Properties: The key
# "com.iplanet.security.SecureRandomFactoryImpl"
# specifies the factory class name for SecureRandomFactory
# Available impl classes are:
#   com.iplanet.am.util.JSSSecureRandomFactoryImpl (uses JSS)
#   com.iplanet.am.util.SecureRandomFactoryImpl (pure Java)
com.iplanet.security.SecureRandomFactoryImpl=com.iplanet.am.util.SecureRandomFactoryImpl

# SocketFactory properties: The key "com.iplanet.security.SSLSocketFactoryImpl"
# specifies the factory class name for LDAPSocketFactory
# Available classes are:
#    com.iplanet.services.ldap.JSSSocketFactory (uses JSS)
```

```
#    com.sun.identity.shared.ldap.factory.JSSESocketFactory    (pure Java)
com.iplanet.security.SSLSocketFactoryImpl=com.sun.identity.shared.ldap.factory.JSSESocketFactory

# Encryption: The key "com.iplanet.security.encryptor" specifies
# the encrypting class implementation.
# Available classes are:
#    com.iplanet.services.util.JCEEncryption
#    com.iplanet.services.util.JSSEncryption
com.iplanet.security.encryptor=com.iplanet.services.util.JCEEncryption

# Determines if JSS will be added with highest priority to JCE
# Set this to "true" if other JCE providers should be used for
# digial signatures and encryptions.
com.sun.identity.jss.donotInstallAtHighestPriority=true

# Configuration File (serverconfig.xml) Location
com.iplanet.services.configpath=@BASE_DIR@
```

fedlet.cot

```
cot-name=Circle of Trust
sun-fm-cot-status=Active
sun-fm-trusted-providers=idp-entityID, sp
sun-fm-saml2-readerservice-url=
sun-fm-saml2-writerservice-url=
```

idp-extended.xml

```
<EntityConfig entityID="idp-entityID" hosted="0" xmlns="urn:sun:fm:SAML:2.0:entityconfig">
    <IDPSSOConfig>
        <Attribute name="description">
            <Value/>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </IDPSSOConfig>
    <AttributeAuthorityConfig>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </AttributeAuthorityConfig>
    <XACMLPDPConfig>
        <Attribute name="wantXACMLAuthzDecisionQuerySigned">
            <Value></Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </XACMLPDPConfig>
</EntityConfig>
```

sp.xml

```
<EntityDescriptor entityID="sp" xmlns="urn:oasis:names:tc:SAML:2.0:metadata">
    <SPSSODescriptor AuthnRequestsSigned="false" WantAssertionsSigned="false"
protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
        <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect" Location="https://
sp.example.com:8443/home/saml/fedletSloRedirect" ResponseLocation="https://sp.example.com:8443/home/saml/
fedletSloRedirect"/>
        <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="https://
sp.example.com:8443/home/saml/fedletSloPOST" ResponseLocation="https://sp.example.com:8443/home/saml/fedletSloPOST"/>
        <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="https://sp.example.com:
8443/home/saml/fedletSloSoap"/>
        <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:transient</NameIDFormat>
        <AssertionConsumerService isDefault="true" index="0" Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
Location="https://sp.example.com:8443/home/saml/fedletapplication"/>
        <AssertionConsumerService index="1" Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Artifact"
Location="https://sp.example.com:8443/home/saml/fedletapplication"/>
    </SPSSODescriptor>
    <RoleDescriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:query="urn:oasis:names:tc:SAML:metadata:ext:query" xsi:type="query:AttributeQueryDescriptorType"
protocolSupportEnumeration= "urn:oasis:names:tc:SAML:2.0:protocol">
    </RoleDescriptor>
    <XACMLAuthzDecisionQueryDescriptor WantAssertionsSigned="false"
protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
    </XACMLAuthzDecisionQueryDescriptor>
</EntityDescriptor>
```

sp-extended.xml

```
<EntityConfig xmlns="urn:sun:fm:SAML:2.0:entityconfig" xmlns:fm="urn:sun:fm:SAML:2.0:entityconfig" hosted="1"
entityID="sp">
    <SPSSOConfig metaAlias="/sp">
        <Attribute name="description">
            <Value></Value>
        </Attribute>
        <Attribute name="signingCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthOn">
            <Value>false</Value>
        </Attribute>
        <Attribute name="basicAuthUser">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthPassword">
            <Value></Value>
        </Attribute>
        <Attribute name="autofedEnabled">
            <Value>false</Value>
        </Attribute>
        <Attribute name="autofedAttribute">
            <Value></Value>
        </Attribute>
        <Attribute name="transientUser">
            <Value>anonymous</Value>
        </Attribute>
        <Attribute name="spAdapter">
            <Value></Value>
        </Attribute>
        <Attribute name="spAdapterEnv">
            <Value></Value>
        </Attribute>
        <Attribute name="fedletAdapter">
            <Value></Value>
        </Attribute>
        <Attribute name="fedletAdapterEnv">
            <Value></Value>
        </Attribute>
        <Attribute name="spAccountMapper">
            <Value>com.sun.identity.saml2.plugins.DefaultLibrarySPAccountMapper</Value>
        </Attribute>
        <Attribute name="spAttributeMapper">
            <Value>com.sun.identity.saml2.plugins.DefaultSPAttributeMapper</Value>
        </Attribute>
        <Attribute name="spAuthncontextMapper">
            <Value>com.sun.identity.saml2.plugins.DefaultSPAuthnContextMapper</Value>
        </Attribute>
        <Attribute name="spAuthncontextClassrefMapping">
          <Value>urn:oasis:names:tc:SAML:2.0:ac:classes:unspecified|0|default</Value>
        </Attribute>
        <Attribute name="spAuthncontextComparisonType">
           <Value>exact</Value>
        </Attribute>
        <Attribute name="attributeMap">
           <Value>*=*</Value>
        </Attribute>
```

```
          <Attribute name="saml2AuthModuleName">
              <Value></Value>
          </Attribute>
          <Attribute name="localAuthURL">
              <Value></Value>
          </Attribute>
          <Attribute name="intermediateUrl">
              <Value></Value>
          </Attribute>
          <Attribute name="defaultRelayState">
              <Value></Value>
          </Attribute>
          <Attribute name="appLogoutUrl">
              <Value>https://sp.example.com:8443/home/saml/logout</Value>
          </Attribute>
          <Attribute name="assertionTimeSkew">
              <Value>300</Value>
          </Attribute>
          <Attribute name="wantAttributeEncrypted">
              <Value></Value>
          </Attribute>
          <Attribute name="wantAssertionEncrypted">
              <Value></Value>
          </Attribute>
          <Attribute name="wantNameIDEncrypted">
              <Value></Value>
          </Attribute>
          <Attribute name="wantPOSTResponseSigned">
              <Value></Value>
          </Attribute>
          <Attribute name="wantArtifactResponseSigned">
              <Value></Value>
          </Attribute>
          <Attribute name="wantLogoutRequestSigned">
              <Value></Value>
          </Attribute>
          <Attribute name="wantLogoutResponseSigned">
              <Value></Value>
          </Attribute>
          <Attribute name="wantMNIRequestSigned">
              <Value></Value>
          </Attribute>
          <Attribute name="wantMNIResponseSigned">
              <Value></Value>
          </Attribute>
          <Attribute name="cotlist">
              <Value>Circle of Trust</Value></Attribute>
          <Attribute name="saeAppSecretList">
          </Attribute>
          <Attribute name="saeSPUrl">
              <Value></Value>
          </Attribute>
          <Attribute name="saeSPLogoutUrl">
          </Attribute>
          <Attribute name="ECPRequestIDPListFinderImpl">
              <Value>com.sun.identity.saml2.plugins.ECPIDPFinder</Value>
          </Attribute>
          <Attribute name="ECPRequestIDPList">
              <Value></Value>
          </Attribute>
          <Attribute name="enableIDPProxy">
              <Value>false</Value>
```

```
        </Attribute>
        <Attribute name="idpProxyList">
            <Value></Value>
        </Attribute>
        <Attribute name="idpProxyCount">
            <Value>0</Value>
        </Attribute>
        <Attribute name="useIntroductionForIDPProxy">
            <Value>false</Value>
        </Attribute>
    </SPSSOConfig>
    <AttributeQueryConfig metaAlias="/attrQuery">
        <Attribute name="signingCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="wantNameIDEncrypted">
           <Value></Value>
        </Attribute>
        <Attribute name="cotlist">
           <Value>Circle of Trust</Value>
        </Attribute>
    </AttributeQueryConfig>
    <XACMLAuthzDecisionQueryConfig metaAlias="/pep">
        <Attribute name="signingCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthOn">
            <Value>false</Value>
        </Attribute>
        <Attribute name="basicAuthUser">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthPassword">
            <Value></Value>
        </Attribute>
        <Attribute name="wantXACMLAuthzDecisionResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantAssertionEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
   </XACMLAuthzDecisionQueryConfig>
</EntityConfig>
```

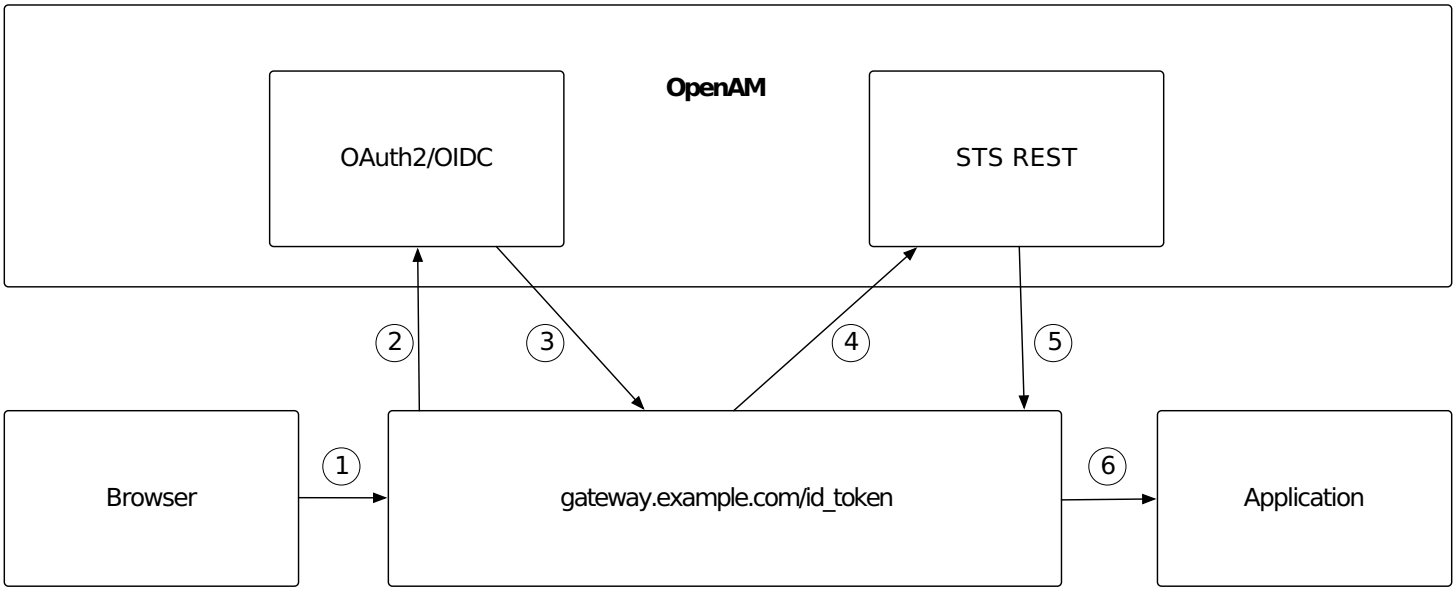
Token transformation

Transform OpenID Connect ID tokens into SAML assertions

This chapter builds on the example in OpenID Connect to transform OpenID Connect ID tokens into SAML 2.0 assertions.

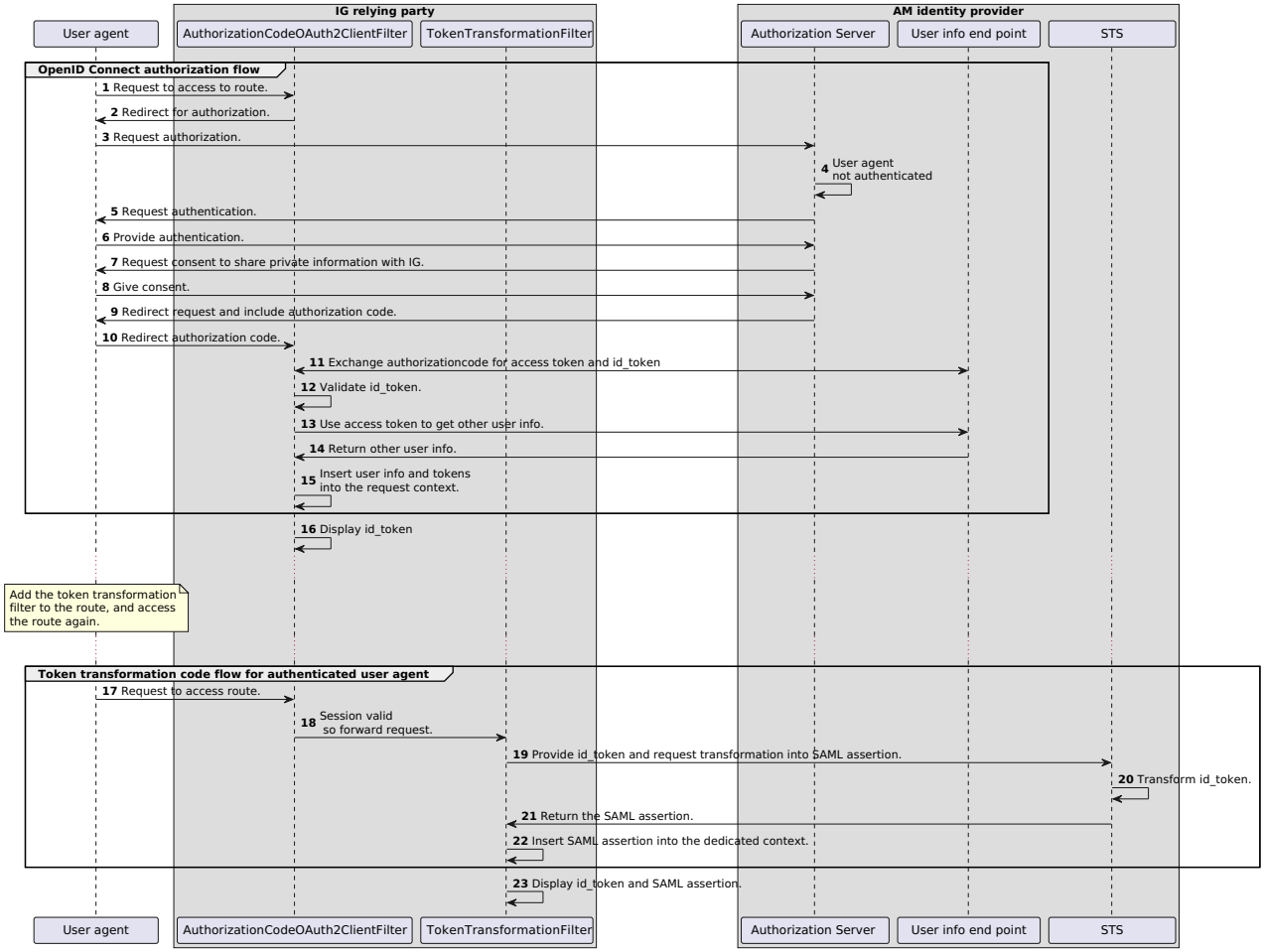
Many enterprises use existing or legacy, SAML 2.0-based SSO, but many mobile and social applications are managed by OpenID Connect. Use the IG TokenTransformationFilter to bridge the gap between OpenID Connect and SAML 2.0 frameworks.

The following figure illustrates the data flow:



1. A user tries to access to a protected resource.
2. If the user isn't authenticated, the AuthorizationCodeOAuth2ClientFilter redirects the request to AM. After authentication, AM asks for the user's consent to give IG access to private information.
3. If the user consents, AM returns an id_token to the AuthorizationCodeOAuth2ClientFilter. The filter opens the id_token JWT and makes it available in `attributes.openid .id_token` and `attributes.openid.id_token_claims` for downstream filters.
4. The TokenTransformationFilter calls the AM STS to transform the id_token into a SAML 2.0 assertion.
5. The STS validates the signature, decodes the payload, and verifies that the user issued the transaction. The STS then issues a SAML assertion to IG on behalf of the user.
6. The TokenTransformationFilter makes the result of the token transformation available to downstream handlers in the `issuedToken` property of the `${contexts.sts}` context.

The following sequence diagram shows a more detailed view of the flow:



Before you start, set up and test the example in AM as a single OpenID Connect provider.

1. Set up an AM Security Token Service (STS), where the subject confirmation method is Bearer. For more information about setting up a REST STS instance, see AM's Security Token Service (STS) guide.

    1. Create a Bearer Module:

        1. In the top level realm, select **Authentication** > **Modules**, and add a module with the following values:

            - **Module name** : `oidc`
            - **Type** : `OpenID Connect id_token bearer`

        2. In the configuration page, enter the following values:

            - **OpenID Connect validation configuration type** : `Client Secret`
            - **OpenID Connect validation configuration value** : `password`

              This is the password of the OAuth 2.0/OpenID Connect client.

            - **Client secret** : `password`
            - **Name of OpenID Connect ID Token Issuer** : `http://am.example.com:8088/openam/oauth2`
            - **Audience name** : `oidc_client`

This is the name of the OAuth 2.0/OpenID Connect client.

- **List of accepted authorized parties** : `oidc_client`

Leave all other values as default, and save your settings.

2. Create an instance of STS REST.

1. In the top level realm, select **STS**, and add a Rest STS instance with the following values:

- **Deployment URL Element** : `openig`

This value identifies the STS instance and is used by the `instance` parameter in the TokenTransformationFilter.

- **SAML2 Token**

> **Note**
>
> For STS, it isn't necessary to create a SAML SP configuration in AM.

- **SAML2 issuer Id** : `OpenAM`
- **Service Provider Entity Id** : `openig_sp`
- **NameIdFormat** : Select `urn:oasis:names:tc:SAML:2.0:nameid-format:transient`

- **OpenID Connect Token**

- **OpenID Connect Token Provider Issuer Id** : `oidc`
- **Token signature algorithm** : Enter a value that is consistent with AM as a single OpenID Connect provider, for example, `HMAC SHA 256`
- **Client Secret** : `password`
- **Issued Tokens Audience** : `oidc_client`

2. On the **SAML 2 Token** tab, add the following **Attribute Mappings**:

- **Key** : `userName` , **Value** : `uid`
- **Key** : `password` , **Value** : `mail`

2. Set up IG:

1. Set an environment variable for `oidc_client` and `ig_agent` , and then restart IG:

```
$ export OIDC_SECRET_ID='cGFzc3dvcmQ='
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/50-idtoken.json
```

Windows

```
%appdata%\OpenIG\config\routes\50-idtoken.json
```

```
{
  "name": "50-idtoken",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/id_token')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AuthenticatedRegistrationHandler-1",
      "type": "Chain",
      "config": {
        "filters": [
          {
            "name": "ClientSecretBasicAuthenticationFilter-1",
            "type": "ClientSecretBasicAuthenticationFilter",
            "config": {
              "clientId": "oidc_client",
              "clientSecretId": "oidc.secret.id",
              "secretsProvider": "SystemAndEnvSecretStore-1"
            }
          }
        ],
        "handler": "ForgeRockClientHandler"
      }
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "AuthorizationCodeOAuth2ClientFilter-1",
          "type": "AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/home/id_token",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "status": 500,
                "headers": {
                  "Content-Type": [
                    "text/plain"
                  ]
                },
                "entity": "An error occurred during the OAuth2 setup."
              }
```

```
            },
            "registrations": [
              {
                "name": "oidc-user-info-client",
                "type": "ClientRegistration",
                "config": {
                  "clientId": "oidc_client",
                  "issuer": {
                    "name": "Issuer",
                    "type": "Issuer",
                    "config": {
                      "wellKnownEndpoint": "http://am.example.com:8088/openam/oauth2/.well-known/
openid-configuration"
                    }
                  },
                  "scopes": [
                    "openid",
                    "profile",
                    "email"
                  ],
                  "authenticatedRegistrationHandler": "AuthenticatedRegistrationHandler-1"
                }
              }
            ],
            "requireHttps": false,
            "cacheExpiration": "disabled"
          }
        },
        {
          "name": "TokenTransformationFilter-1",
          "type": "TokenTransformationFilter",
          "config": {
            "idToken": "${attributes.openid.id_token}",
            "instance": "openig",
            "amService": "AmService-1"
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/plain; charset=UTF-8" ]
          },
          "entity": "{\"id_token\":\n\"${attributes.openid.id_token}\"} \n\n\n{\"saml_assertions\":
\n\"${contexts.sts.issuedToken}\"}"
        }
      }
    }
  }
}
```

For information about how to set up the IG route in Studio, refer to Token transformation in Structured Editor.

Notice the following features of the route:

- The route matches requests to `/home/id_token`.
- The AmService in the heap is used for authentication and REST STS requests.

- The AuthorizationCodeOAuth2ClientFilter enables IG to act as an OpenID Connect relying party:
  - The client endpoint is set to `/home/id_token`, so the service URIs for this filter on the IG server are `/home/id_token/login`, `/home/id_token/logout`, and `/home/id_token/callback`.
  - For convenience in this test, `requireHttps` is false. In production environments, set it to true. So that you see the delegated authorization process when you make a request, `requireLogin` is true.
  - The target for storing authorization state information is `${attributes.openid}`. Subsequent filters and handlers can find access tokens and user information at this target.
- The ClientRegistration holds configuration provided in AM as a single OpenID Connect provider, and used by IG to connect with AM.
- The TokenTransformationFilter transforms an id_token into a SAML assertion:
  - The `id_token` parameter defines where this filter gets the id_token created by the `AuthorizationCodeOAuth2ClientFilter`.

    The TokenTransformationFilter makes the result of the token transformation available to downstream handlers in the `issuedToken` property of the `${contexts.sts}` context.
  - The `instance` parameter must match the `Deployment URL Element` for the REST STS instance.

    Errors that occur during token transformation cause an error response to be returned to the client and an error message to be logged for the IG administrator.
- When the request succeeds, a StaticResponseHandler retrieves and displays the id_token from the target `{attributes.openid.id_token}`.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/id_token.

      The AM login screen is displayed.

   2. Log in to AM as username `demo`, password `Ch4ng31t`.

      An OpenID Connect request to access private information is displayed.

   3. Select **Allow**.

      The id_token and SAML assertions are displayed:

      ```
      {"id_token": "eyA . . ."}

      {"saml_assertions": "<\"saml:Assertion xmlns:saml=\"urn:oasis:names:tc:SAML:2.0:assertion\"
      Version= . . ."}
      ```

> **Tip**
>
> If a request returns an HTTP 414 URI Too Long error, consider the information in URI Too Long error.

OAuth 2.0 token exchange

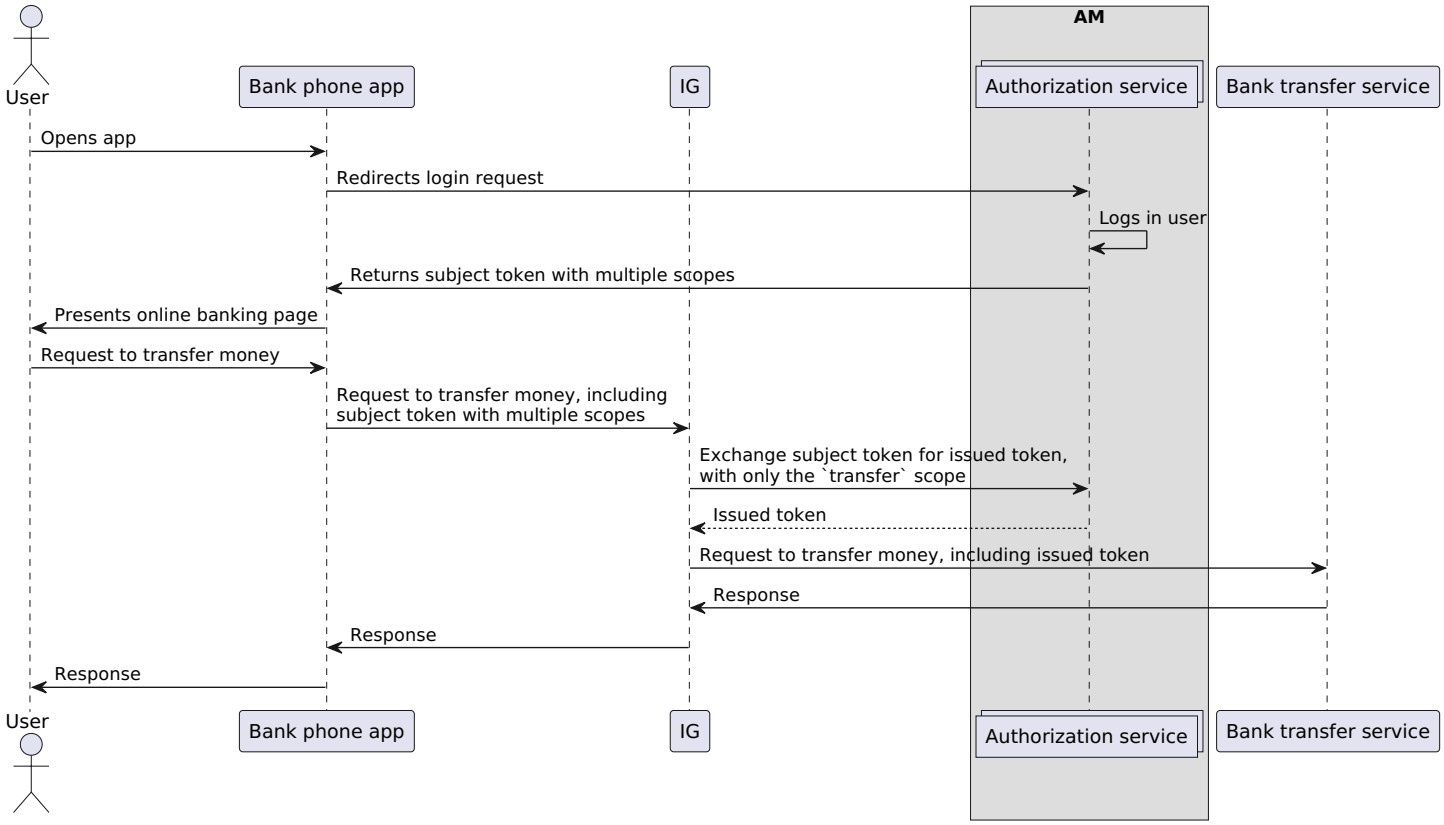
The following sections describe how to exchange an OAuth 2.0 access token for another access token, with AM as an Authorization Server. Other authorization providers can be used instead of AM.

Token exchange requires a *subject token* and provides an *issued token*. The subject token is the original access token, obtained using the OAuth 2.0/OpenID Connect flow. The issued token is provided in exchange for the subject token.

The token exchange can be conducted only by an OAuth 2.0 client that "may act" on the subject token, as configured in the authorization service.

This example is a typical scenario for *token impersonation*. For more information, refer to Token exchange in AM's *OAuth 2.0 guide*.

The following sequence diagram shows the flow of information during token exchange between IG and AM:



Important

This procedure uses the *Resource Owner Password Credentials* grant type. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize other grant types whenever possible.

Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

1. Set up AM:
    1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
        - `https://ig.example.com:8443/*`

- `https://ig.example.com:8443/*?*`

2. Register an IG agent with the following values, as described in Register an IG agent in AM:
   - **Agent ID**: `ig_agent`
   - **Password**: `password`
   - **Token Introspection**: `Realm Only`

   > **Important**
   >
   > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

3. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

   > **Important**
   >
   > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

4. Select **Services** > **Add a Service**, and add an **OAuth2 Provider** service with the following values:
   - **OAuth2 Access Token May Act Script** : `OAuth2 May Act Script`
   - **OAuth2 ID Token May Act Script** : `OAuth2 May Act Script`

5. Select **</> Scripts**# > **OAuth2 May Act Script**, and replace the example script with the following script:

```
import org.forgerock.json.JsonValue
token.setMayAct(
    JsonValue.json(JsonValue.object(
        JsonValue.field("client_id", "serviceConfidentialClient"))))
```

   This script adds a `may_act` claim to the token, indicating that the OAuth 2.0 client, `serviceConfidentialClient`, may act to exchange the subject token in the impersonation use case.

6. Add an OAuth 2.0 Client to request OAuth 2.0 access tokens:
   1. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:
      - **Client ID** : `client-application`
      - **Client secret** : `password`
      - **Scope(s)** : `mail`, `employeenumber`
   2. On the **Advanced** tab, select **Grant Types** : `Resource Owner Password Credentials`.

7. Add an OAuth 2.0 client to perform the token exchange:
    1. Select **Applications** > **OAuth 2.0** > **Clients**, and add a client with the following values:
        - **Client ID** : `serviceConfidentialClient`
        - **Client secret** : `password`
        - **Scope(s)** : `mail`, `employeenumber`
    2. On the **Advanced** tab, select:
        - **Grant Types** : `Token Exchange`
        - **Token Endpoint Authentication Methods** : `client_secret_post`

2. Set up IG:
    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
    2. Set the following environment variables for the serviceConfidentialClient password:

```
$ export CLIENT_SECRET_ID='cGFzc3dvcmQ='
```

    3. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

    The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

    4. Add the following route to IG to exchange the access token:

Linux

```
$HOME/.openig/config/routes/token-exchange.json
```

Windows

```
%appdata%\OpenIG\config\routes\token-exchange.json
```

```
{
  "name": "token-exchange",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/token-exchange')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    },
    {
      "name": "ExchangeHandler",
      "type": "Chain",
      "capture": "all",
      "config": {
        "handler": "ForgeRockClientHandler",
        "filters": [
          {
            "type": "ClientSecretBasicAuthenticationFilter",
            "config": {
              "clientId": "serviceConfidentialClient",
              "clientSecretId": "client.secret.id",
              "secretsProvider" : "SystemAndEnvSecretStore-1"
            }
          }
        ]
      }
    },
    {
      "name": "ExchangeFailureHandler",
      "type": "StaticResponseHandler",
      "capture": "all",
      "config": {
        "status": 400,
        "entity": "${contexts.oauth2Failure.error}: ${contexts.oauth2Failure.description}",
        "headers": {
          "Content-Type": [
            "application/json"
          ]
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "oauth2TokenExchangeFilter",
```

```
            "type": "OAuth2TokenExchangeFilter",
            "config": {
              "amService": "AmService-1",
              "endpointHandler": "ExchangeHandler",
              "subjectToken": "#{request.entity.form['subject_token'][0]}",
              "scopes": ["mail"],
              "failureHandler": "ExchangeFailureHandler"
            }
          }
        ],
        "handler": {
          "type": "StaticResponseHandler",
          "config": {
            "status": 200,
            "headers": {
              "content-type": [
                "application/json"
              ]
            },
            "entity": "{\"access_token\": \"${contexts.oauth2TokenExchange.issuedToken}\",
\"issued_token_type\": \"${contexts.oauth2TokenExchange.issuedTokenType}\"}"
          }
        }
      }
    }
  }
```

Notice the following features of the route:

- The route matches requests to `/token-exchange`
- IG reads the `subjectToken` from the request entity.
- The StaticResponseHandler returns an issued token.

3. Test the setup:

   1. In a terminal window, use a `curl` command similar to the following to retrieve an access token, which is the *subject token*:

      ```
      $ subjecttoken=$(curl -s \
      --user "client-application:password" \
      --data "grant_type=password&username=demo&password=Ch4ng31t&scope=mail%20employeenumber" \
      http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token") \
      && echo $subjecttoken

      hc-...c6A
      ```

   2. Introspect the subject token at the AM introspection endpoint:

```
$ curl --location \
--request POST 'http://am.example.com:8088/openam/oauth2/realms/root/introspect' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode "token=${subjecttoken}" \
--data-urlencode 'client_id=client-application' \
--data-urlencode 'client_secret=password'

Decoded access_token: {
  "active": true,
  "scope": "employeenumber mail",
  "realm": "/",
  "client_id": "client-application",
  "user_id": "demo",
  "username": "demo",
  "token_type": "Bearer",
  "exp": 1626796888,
  "sub": "(usr!demo)",
  "subname": "demo",
  "iss": "http://am.example.com:8088/openam/oauth2",
  "auth_level": 0,
  "authGrantId": "W-j...E1E",
  "may_act": {
    "client_id": "serviceConfidentialClient"
  },
  "auditTrackingId": "4be...169"
}
```

Note that in the subject token, the `client_id` is `client-application`, and the scopes are `employeenumber` and `mail`. The `may_act` claim indicates that `serviceConfidentialClient` is authorized to exchange this token.

3. Exchange the subject token for an issued token:

```
$ issuedtoken=$(curl \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--location \
--request POST 'https://ig.example.com:8443/token-exchange' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data "subject_token=${subjecttoken}" | jq -r ".access_token") \
&& echo $issuedtoken

F8e...Q3E
```

4. Introspect the issued token at the AM introspection endpoint:

```
$ curl --location \
--request POST 'http://am.example.com:8088/openam/oauth2/realms/root/introspect' \
--header 'Content-Type: application/x-www-form-urlencoded' \
--data-urlencode "token=${issuedtoken}" \
--data-urlencode 'client_id=serviceConfidentialClient' \
--data-urlencode 'client_secret=password'

{
  "active": true,
  "scope": "mail",
  "realm": "/",
  "client_id": "serviceConfidentialClient",
  "user_id": "demo",
  "username": "demo",
  "token_type": "Bearer",
  "exp": 1629200490,
  "sub": "(usr!demo)",
  "subname": "demo",
  "iss": "http://am.example.com:8088/openam/oauth2",
  "auth_level": 0,
  "authGrantId": "aYK...EPA",
  "may_act": {
    "client_id": "serviceConfidentialClient"
  },
  "auditTrackingId": "814...367"
}
```

Note that in the issued token, the `client_id` is `serviceConfidentialClient`, and the only the scope is `mail`.

Not-enforced URIs

By default, IG routes protect resources (such as a websites or applications) from all requests on the route's condition path. Some parts of the resource, however, do not need to be protected. For example, it can be okay for unauthenticated requests to access the welcome page of a web site, or an image or favicon.

The following sections give examples of routes that do not enforce authentication for a specific request URL or URL pattern, but enforce authentication for other request URLs:

Implement not-enforced URIs with a SwitchFilter

Before you start:

- Prepare IG and the sample app as described in the Quick install
- Install and configure AM on http://am.example.com:8088/openam, using the default configuration.

1. On your system, add the following data in a comma-separated value file:

Linux

```
/tmp/userfile.txt
```

Windows

```
C:\Temp\userfile.txt
```

```
username,password,fullname,email
george,C0stanza,George Costanza,george@example.com
kramer,N3wman12,Kramer,kramer@example.com
bjensen,H1falutin,Babs Jensen,bjensen@example.com
demo,Ch4ng31t,Demo User,demo@example.com
kvaughan,B5ibery12,Kirsten Vaughan,kvaughan@example.com
scarter,S9rain12,Sam Carter,scarter@example.com
```

2. Set up AM:

    1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:

        - `https://ig.example.com:8443/*`
        - `https://ig.example.com:8443/*?*`

    2. Register an IG agent with the following values, as described in Register an IG agent in AM:

        - **Agent ID**: `ig_agent`
        - **Password**: `password`

        Important

        Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

    3. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

        Important

        IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

3. Set up IG:

    1. Set an environment variable for the IG agent password, and then restart IG:

    ```
    $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
    ```

    The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

    2. Add the following route to IG to serve the sample application .css and other static resources:

    Linux

    ```
    $HOME/.openig/config/routes/00-static-resources.json
    ```

    Windows

    ```
    %appdata%\OpenIG\config\routes\00-static-resources.json
    ```

    ```
    {
      "name" : "00-static-resources",
      "baseURI" : "http://app.example.com:8081",
      "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path,
    '^/.*\\\\.ico$') or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
      "handler": "ReverseProxyHandler"
    }
    ```

    3. Add the following route to IG:

    Linux

    ```
    $HOME/.openig/config/routes/not-enforced-switch.json
    ```

Windows

```
%appdata%\OpenIG\config\routes\not-enforced-switch.json
```

```
{
  "properties": {
    "notEnforcedPathPatterns": "^/home|^/favicon.ico|^/css"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "name": "not-enforced-switch",
  "condition": "${find(request.uri.path, '^/')}",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SwitchFilter-1",
          "type": "SwitchFilter",
          "config": {
            "onRequest": [{
              "condition": "${find(request.uri.path, '&{notEnforcedPathPatterns}')}",
              "handler": "ReverseProxyHandler"
            }]
          }
        },
        {
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${true}",
            "credentials": {
              "type": "FileAttributesFilter",
              "config": {
                "file": "/tmp/userfile.txt",
                "key": "email",
                "value": "${contexts.ssoToken.info.uid}@example.com",
                "target": "${attributes.credentials}"
              }
            },
            "request": {
              "method": "POST",
```

```
                "uri": "http://app.example.com:8081/login",
                "form": {
                  "username": [
                    "${attributes.credentials.username}"
                  ],
                  "password": [
                    "${attributes.credentials.password}"
                  ]
                }
              }
            }
          }
        ],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

Notice the following features of the route:

- The route condition is `/`, so the route matches all requests.
- The SwitchFilter passes requests on the path `^/home`, `^/favicon.ico`, and `^/css` directly to the ReverseProxyHandler. All other requests continue the along the chain to the SingleSignOnFilter.
- If the request does not have a valid AM session cookie, the SingleSignOnFilter redirects the request to AM for authentication. The SingleSignOnFilter stores the cookie value in an `SsoTokenContext`.
- Because the PasswordReplayFilter detects that the response is a login page, it uses the FileAttributesFilter to replay the password, and logs the request into the sample application.

4. Test the setup:

    1. If you are logged in to AM, log out and clear any cookies.
    2. Access the route on the not-enforced URL http://ig.example.com:8080/home. The home page of the sample app is displayed without authentication.
    3. Access the route on the enforced URL http://ig.example.com:8080/profile. The SingleSignOnFilter redirects the request to AM for authentication.
    4. Log in to AM as user `demo`, password `Ch4ng31t`. The PasswordReplayFilter replays the credentials for the demo user. The request is passed to the sample app's profile page for the demo user.

Implement not-enforced URIs with a DispatchHandler

To use a DispatchHandler for not-enforced URIs, replace the route in Implement not-enforced URIs with a SwitchFilter with the following route. If the request is on the path `^/home`, `^/favicon.ico`, or `^/css`, the DispatchHandler sends it directly to the ReverseProxyHandler, without authentication. It passes all other requests into the Chain for authentication.

```
{
  "properties": {
    "notEnforcedPathPatterns": "^/home|^/favicon.ico|^/css"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "name": "not-enforced-dispatch",
  "condition": "${find(request.uri.path, '^/')}",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${find(request.uri.path, '&{notEnforcedPathPatterns}')}",
          "handler": "ReverseProxyHandler"
        },
        {
          "handler": {
            "type": "Chain",
            "config": {
              "filters": [
                {
                  "type": "SingleSignOnFilter",
                  "config": {
                    "amService": "AmService-1"
                  }
                },
                {
                  "type": "PasswordReplayFilter",
                  "config": {
                    "loginPage": "${true}",
                    "credentials": {
                      "type": "FileAttributesFilter",
                      "config": {
                        "file": "/tmp/userfile.txt",
                        "key": "email",
                        "value": "${contexts.ssoToken.info.uid}@example.com",
                        "target": "${attributes.credentials}"
                      }
                    },
                    "request": {
                      "method": "POST",
                      "uri": "http://app.example.com:8081/login",
```

```
                        "form": {
                          "username": [
                            "${attributes.credentials.username}"
                          ],
                          "password": [
                            "${attributes.credentials.password}"
                          ]
                        }
                      }
                    }
                  }
                ],
                "handler": "ReverseProxyHandler"
              }
            }
          }
        ]
      }
    }
  }
```

POST data preservation

The DataPreservationFilter triggers POST data preservation when an unauthenticated client posts HTML form data to a protected resource.

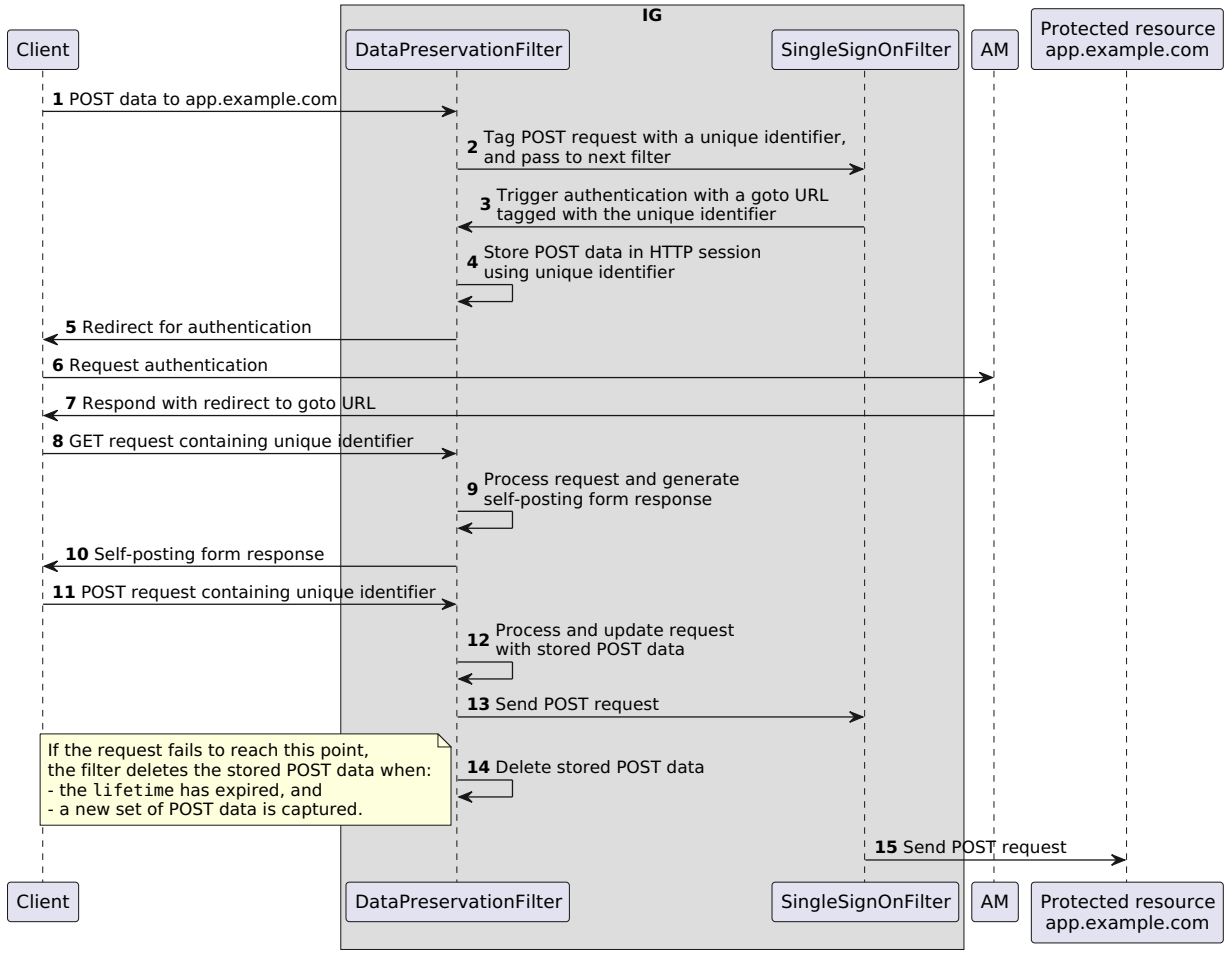
When an authentication redirect is triggered, the filter stores the data in the HTTP session, and redirects the client for authentication. After authentication, the filter generates an empty self-submitting form POST to emulate the original POST. It then replays the stored data into the request before passing it along the chain.

The data can be any POST content, such as HTML form data or a file upload.

Consider the following points for POST data preservation:

- The size of the POST data is important because the data is stored in the HTTP session.
- Stateless sessions store form content in encrypted JWT session cookies. To prevent requests from being rejected because the HTTP headers are too long, configure `connectors:maxTotalHeadersSize` in admin.json.
- Sticky sessions may be required for deployments with stateful sessions, and multiple IG instances.

The following image shows a simplified data flow for POST data preservation:



1. An unauthenticated client requests a POST to a protected resource.

2. The DataPreservationFilter tags the the request with a unique identifier, and passes it along the chain. The next filter should be an authentication filter such as a SingleSignOnFilter.

3. The next filter triggers the authentication, and includes a goto URL tagged with the unique identifier from the previous step.

4-5. The DataPreservationFilter stores the POST data in the HTTP session, and redirects the request for authentication. The POST data is identified by the unique identifier.

6-7. The client authenticates with AM, and AM provides an authentication response to the goto URL.

8. The authenticated client sends a GET request containing the unique identifier.

9-10. The DataPreservationFilter validates the unique identifier, and generates a self-posting form response for the client.

The presence of the unique identifier in the goto URL ensures that requests at the URL can be individually identified. Additionally, it is more difficult to hijack user data, because there is little chance of guessing the code within the login window.

If the identifier is not validated, IG denies the request.

11. The client resends the POST request, including the identifier.

12-13. The DataPreservationFilter updates the request with the POST data, and sends it along the chain.

Preserve POST data during CDSSO

Before you start, set up and test the example in Cross-domain single sign-on. This example extends that example.

1. In IG, replace `cdsso.json` with the following route:

Linux

```
$HOME/.openig/config/routes/pdp.json
```

Windows

```
%appdata%\OpenIG\config\routes\pdp.json
```

```
{
  "name": "pdp",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/cdsso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "agent": {
          "username": "ig_agent_cdsso",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "sessionCache": {
          "enabled": false
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "DataPreservationFilter",
          "type": "DataPreservationFilter"
        },
        {
          "name": "CrossDomainSingleSignOnFilter-1",
          "type": "CrossDomainSingleSignOnFilter",
          "config": {
            "redirectEndpoint": "/home/cdsso/redirect",
            "authCookie": {
              "path": "/home",
              "name": "ig-token-cookie"
            },
            "amService": "AmService-1",
            "logoutExpression": "${find(request.uri.query, 'logOff=true')}",
            "defaultLogoutLandingPage": "/form"
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [
              "text/html; charset=UTF-8"
            ]
          },
          "entity": [
            "<html>",
```

```
            "  <body>",
            "    <h1>Request Information</h1>",
            "      <p>Request method: #{request.method}",
            "      <p>Request URI: #{request.uri}",
            "      <p>Query string: #{request.queryParams}",
            "      <p>Form: #{request.entity.form}",
            "      <p>Content length: #{request.headers['Content-Length'][0]}",
            "      <p>Content type: #{request.headers['Content-Type'][0]}",
            "  </body>",
            "</html>"
          ]
        }
      }
    }
  }
}
```

Notice the following differences compared to `cdsso.json`:

- A DataPreservationFilter is positioned in front of the CrossDomainSingleSignOnFilter to manage POST data preservation before authentication.
- The ReverseProxyHandler is replaced by a StaticResponseHandler, which displays the POST data provided in the request.

2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/form.json
```

Windows

```
%appdata%\OpenIG\config\routes\pdp.json
```

```
{
  "condition": "${request.uri.path == '/form'}",
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/html" ]
      },
      "entity" : [
        "<html>",
        "  <body>",
        "    <h1>Test page : POST Data Preservation containing visible and hidden form elements</h1>",
        "    <form id='testingPDP' enctype='application/x-www-form-urlencoded' name='test_form' action='/home/
cdsso/pdp.info?foo=bar&baz=pdp' method='post'>",
        "      <input name='email' value='user@example.com' size='60'>",
        "      <input type='hidden' name='phone' value='555-123-456'/>",
        "      <input type='hidden' name='manager' value='Bob'/>",
        "      <input type='hidden' name='dept' value='Engineering'/>",
        "      <input type='submit' value='Press to demo form posting' id='form_post_button'/>",
        "    </form>",
        "  </body>",
        "</html>"
      ]
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/home/form`.
- The StaticResponseHandler includes the following entity to present visible and hidden form elements from the original request:

```
<!DOCTYPE html>
<html>
   <body>
      <h1>Test page : POST Data Preservation containing visible and hidden form elements</h1>
      <form
         id='testingPDP'
         enctype='application/x-www-form-urlencoded'
         name='test_form'
         action='/home/cdsso/pdp.info?foo=bar&baz=pdp'
         method='post'>
         <input
            name='email'
            value='user@example.com'
            size='60'>
         <input
            type='hidden'
            name='phone'
            value='555-123-456'/>
         <input
            type='hidden'
            name='manager'
            value='Bob'/>
         <input
            type='hidden'
            name='dept'
            value='Engineering'/>
         <input
            type='submit'
            value='Press to demo form posting'
            id='form_post_button'/>
      </form>
   </body>
</html>
```

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.ext.com:8443/form.

      The script in the StaticResponseHandler entity of `form.json` creates a button to demonstrate form posting.

   2. Press the button, and log in to AM as user `demo`, password `Ch4ng31t`.

      When you have authenticated, the script presents the POST data from the original request.

CSRF protection

In a Cross Site Request Forgery (CSRF) attack, a user unknowingly executes a malicious request on a website where they're authenticated. A CSRF attack usually includes a link or script in a web page. When a user accesses the link or script, the page executes an HTTP request on the site where the user is authenticated.

CSRF attacks interact with HTTP requests as follows:

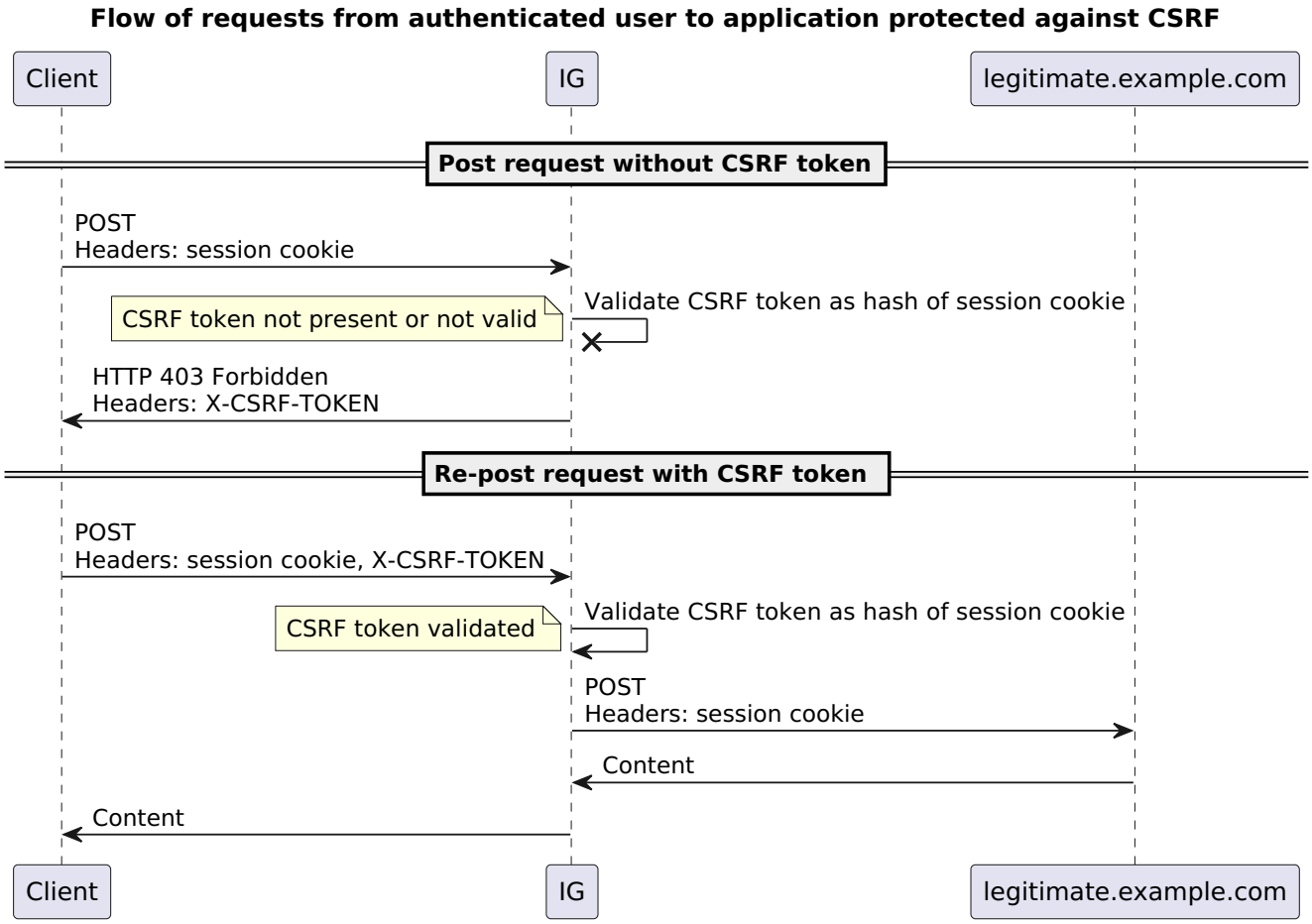
- CSRF attacks can execute POST, PUT, and DELETE requests on the targeted server. For example, a CSRF attack can transfer funds out of a bank account or change a user's password.

- Because of same-origin policy, CSRF attacks **cannot** access any response from the targeted server.

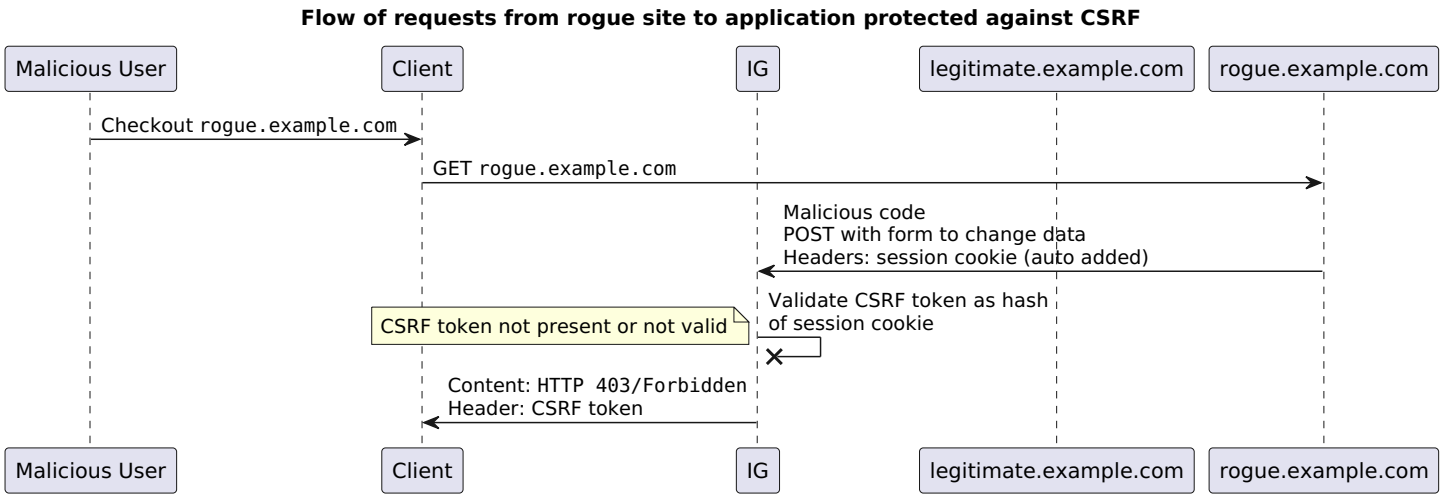
When IG processes POST, PUT, and DELETE requests, it checks a custom HTTP header in the request. If a CSRF token isn't present in the header or not valid, IG rejects the request and returns a valid CSRF token in the response.

Rogue websites that attempt CSRF attacks operate in a different website domain to the targeted website. Because of same-origin policy, rogue websites can't access a response from the targeted website, and can't, therefore, access the response or CSRF token.

The following example shows the data flow when an authenticated user sends a POST request to an application protected against CSRF:



The following example shows the data flow when an authenticated user sends a POST request from a rogue site to an application protected against CSRF:



1. Set up SSO, so that AM authenticates users to the sample app through IG:

    1. Set up AM and IG as described in Authenticate with SSO through the default authentication service.

    2. Remove the condition in `sso.json`, so that the route matches all requests:

    ```
    "condition": "${find(request.uri.path, '^/home/sso')}"
    ```

2. Test the setup without CSRF protection:

    1. Go to https://ig.example.com:8443/bank/index, and log in to the Sample App Bank through AM, as user `demo`, password `Ch4ng31t`.

    2. Send a bank transfer of $10 to Bob, and note that the transfer is successful.

    3. Go to http://localhost:8081/bank/attack-autosubmit to simulate a CSRF attack.

    Tip

    In deployments that use a different protocol, hostname, or port for IG, append the `igUrl` parameter, as follows:

    ```
    http://localhost:8081/bank/attack-autosubmit?igUrl=protocol://hostname:port
    ```

    When you access this page, a hidden HTML form is automatically submitted to transfer $1000 to the rogue user, using the IG session cookie to authenticate to the bank.

    In the bank transaction history, note that $1000 is debited.

3. Test the setup with CSRF protection:

    1. In IG, replace `sso.json` with the following route:

```
{
  "name": "Csrf",
  "baseURI": "http://app.example.com:8081",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    },
    {
      "name": "FailureHandler-1",
      "type": "StaticResponseHandler",
      "config": {
        "status": 403,
        "headers": {
          "Content-Type": [ "text/plain; charset=UTF-8" ]
        },
        "entity": "Request forbidden"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "CsrfFilter-1",
          "type": "CsrfFilter",
          "config": {
            "cookieName": "iPlanetDirectoryPro",
            "failureHandler": "FailureHandler-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route compared to `sso.json`:

- The CsrfFilter checks the AM session cookie for the `X-CSRF-Token` header. If a CSRF token isn't present in the header or not valid, the filter rejects the request and provides a valid CSRF token in the header.

2. Go to https://ig.example.com:8443/bank/index, and send a bank transfer of $10 to Alice.

   Because there is no CSRF token, IG responds with an HTTP 403, and provides the token.

3. Send the transfer again, and note that because the CSRF token is provided the transfer is successful.

4. Go to http://localhost:8081/bank/attack-autosubmit to automatically send a rogue transfer.

> **Tip**
>
> In deployments that use a different protocol, hostname, or port for IG, append the `igUrl` parameter, as follows:
>
> ```
> http://localhost:8081/bank/attack-autosubmit?igUrl=protocol://hostname:port
> ```

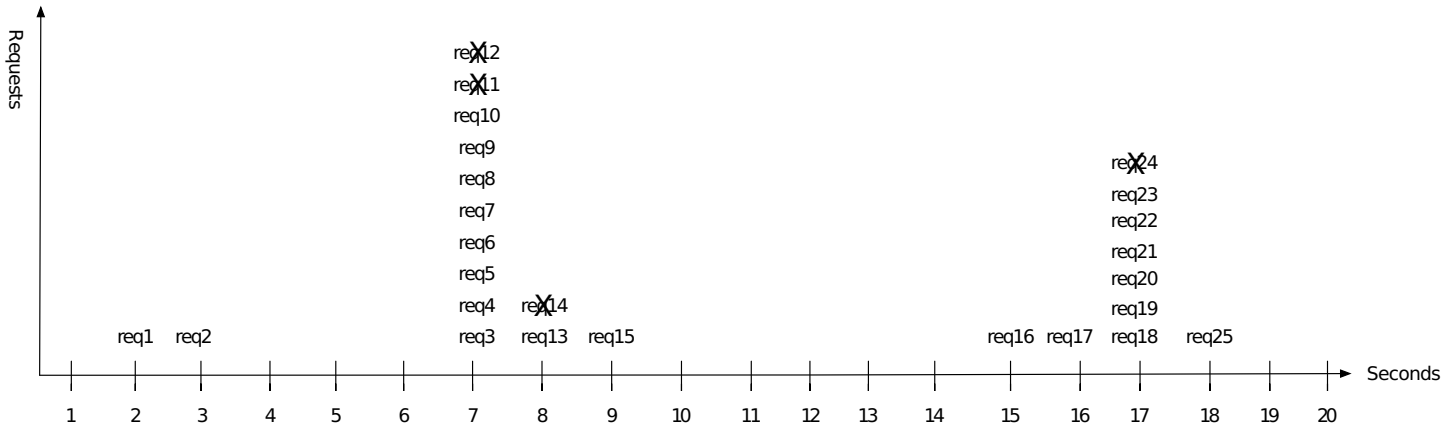
Because there is no CSRF token, IG rejects the request and provides the CSRF token. However, because the rogue site is in a different domain to `ig.example.com` it can't access the CSRF token.

Throttling

To protect applications from being overused by clients, use a throttling filter to limit how many requests can be made in a defined time. The maximum number of requests that are allowed in a defined time is called the *throttling rate*. The following sections describe how to set up simple, mapped, and scriptable throttling filters:

About throttling

The throttling filter uses the token bucket algorithm, allowing some unevenness or bursts in the request flow. The following image shows how IG manages requests for a throttling rate of 10 requests/10 seconds:



- At 7 seconds, 2 requests have previously passed when there is a burst of 9 requests. IG allows 8 requests, but disregards the 9th because the throttling rate for the 10-second throttling period has been reached.

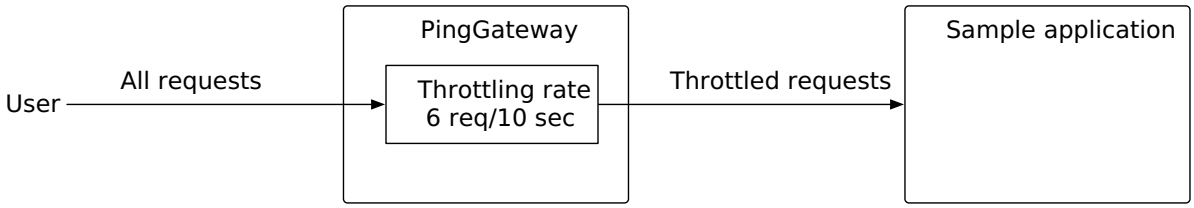
- At 8 and 9 seconds, although 10 requests have already passed in the 10-second throttling period, IG allows 1 request each second.
- At 17 seconds, 4 requests have passed in the previous 10-second throttling period, and IG allows another burst of 6 requests.

When the throttling rate is reached, IG issues an HTTP status code 429 `Too Many Requests` and a `Retry-After` header like the following, where the value is the number of seconds to wait before trying the request again:

```
GET https://ig.example.com:8443/home/throttle-scriptable HTTP/2
. . .

HTTP/2 429 Too Many Requests
Retry-After: 10
```

Configure simple throttling

This section describes how to configure a simple throttling filter that applies a throttling rate of 6 requests/10 seconds. When an application is protected by this throttling filter, no more than 6 requests, irrespective of their origin, can access the sample application in a 10 second period.



1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/00-throttle-simple.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-throttle-simple.json
```

```
{
  "name": "00-throttle-simple",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/throttle-simple')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ThrottlingFilter",
          "name": "ThrottlingFilter-1",
          "config": {
            "requestGroupingPolicy": "",
            "rate": {
              "numberOfRequests": 6,
              "duration": "10 s"
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the IG route in Studio, refer to Simple throttling filter in Structured Editor.

Notice the following features of the route:

- The route matches requests to `/home/throttle-simple`.
- The ThrottlingFilter contains a request grouping policy that is blank. This means that all requests are in the same group.
- The rate defines the number of requests allowed to access the sample application in a given time.

3. Test the setup:

   1. In a terminal window, use a `curl` command similar to the route in a loop:

      ```
      $ curl -v \
      --cacert /path/to/secrets/ig.example.com-certificate.pem \
      https://ig.example.com:8443/home/throttle-simple/\[01-10\] \
      > /tmp/throttle-simple.txt 2>&1
      ```

   2. Search the output file for the result:

      ```
      $ grep "< HTTP/2" /tmp/throttle-simple.txt | sort | uniq -c

      6 < HTTP/2 200 OK
      4 < HTTP/2 429 Too Many Requests
      ```

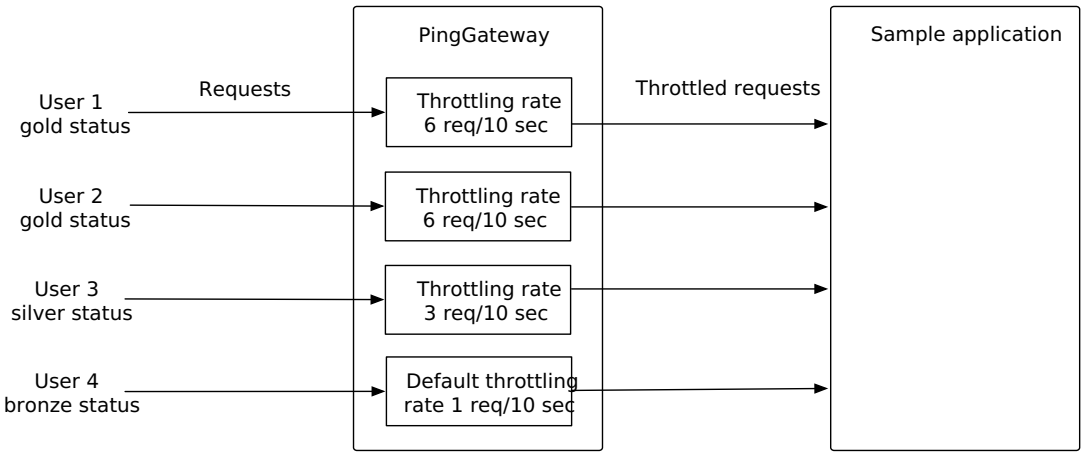
Notice that the first six requests returned a success response, and the following four requests returned an HTTP 429 `Too Many Requests`. This result demonstrates that the throttling filter has allowed only six requests to access the application, and has blocked the other requests.

Configure mapped throttling

This section describes how to configure a mapped throttling policy, where the grouping policy defines criteria to group requests, and the rate policy defines the criteria by which rates are mapped.

The following image illustrates how different throttling rates can be applied to users.

The following image illustrates how each user with a `gold` status has a throttling rate of 6 requests/10 seconds, and each user with a `silver` status has 3 requests/10 seconds. The `bronze` status is not mapped to a throttling rate, and so a user with the `bronze` status has the default rate.



Before you start, set up and test the example in Validate access tokens through the introspection endpoint.

1. In the AM console, select **</> Scripts** > **OAuth2 Access Token Modification Script**, and replace the default script as follows:

```
import org.forgerock.http.protocol.Request
import org.forgerock.http.protocol.Response

def attributes = identity.getAttributes(["mail", "employeeNumber"].toSet())
accessToken.setField("mail", attributes["mail"][0])
def mail = attributes['mail'][0]
if (mail.endsWith('@example.com')) {
  status = "gold"
} else if (mail.endsWith('@other.com')) {
  status = "silver"
} else {
  status = "bronze"
}
accessToken.setField("status", status)
```

   The AM script adds user profile information to the access token, and defines the content of the users `status` field according to the email domain.

2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/00-throttle-mapped.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-throttle-mapped.json
```

```
{
  "name": "00-throttle-mapped",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/throttle-mapped')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "token-resolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        }
```

```
        },
        {
          "name": "ThrottlingFilter-1",
          "type": "ThrottlingFilter",
          "config": {
            "requestGroupingPolicy": "${contexts.oauth2.accessToken.info.mail}",
            "throttlingRatePolicy": {
              "name": "MappedPolicy",
              "type": "MappedThrottlingPolicy",
              "config": {
                "throttlingRateMapper": "${contexts.oauth2.accessToken.info.status}",
                "throttlingRatesMapping": {
                  "gold": {
                    "numberOfRequests": 6,
                    "duration": "10 s"
                  },
                  "silver": {
                    "numberOfRequests": 3,
                    "duration": "10 s"
                  },
                  "bronze": {
                    "numberOfRequests": 1,
                    "duration": "10 s"
                  }
                },
                "defaultRate": {
                  "numberOfRequests": 1,
                  "duration": "10 s"
                }
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the IG route in Studio, refer to Mapped throttling filter in Structured Editor.

Notice the following features of the route:

- The route matches requests to `/home/throttle-mapped`.
- The OAuth2ResourceServerFilter validates requests with the AccessTokenResolver, and makes it available for downstream components in the `oauth2` context.
- The ThrottlingFilter bases the request grouping policy on the AM user's email. The throttling rate is applied independently to each email address.

  The throttling rate is mapped to the AM user's `status`, which is defined by the email domain, in the AM script.

3. Test the setup:

   1. In a terminal window, use a `curl` command similar to this to retrieve an access token:

```
$ mytoken=$(curl -s \
--user "client-application:password" \
--data "grant_type=password&username=demo&password=Ch4ng31t&scope=mail%20employeenumber" \
http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
```

2. Using the access token, access the route multiple times. The following example accesses the route 10 times and writes the output to a file:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/home/throttle-mapped/\[01-10\] \
> /tmp/throttle-mapped.txt 2>&1
```

3. Search the output file for the result:

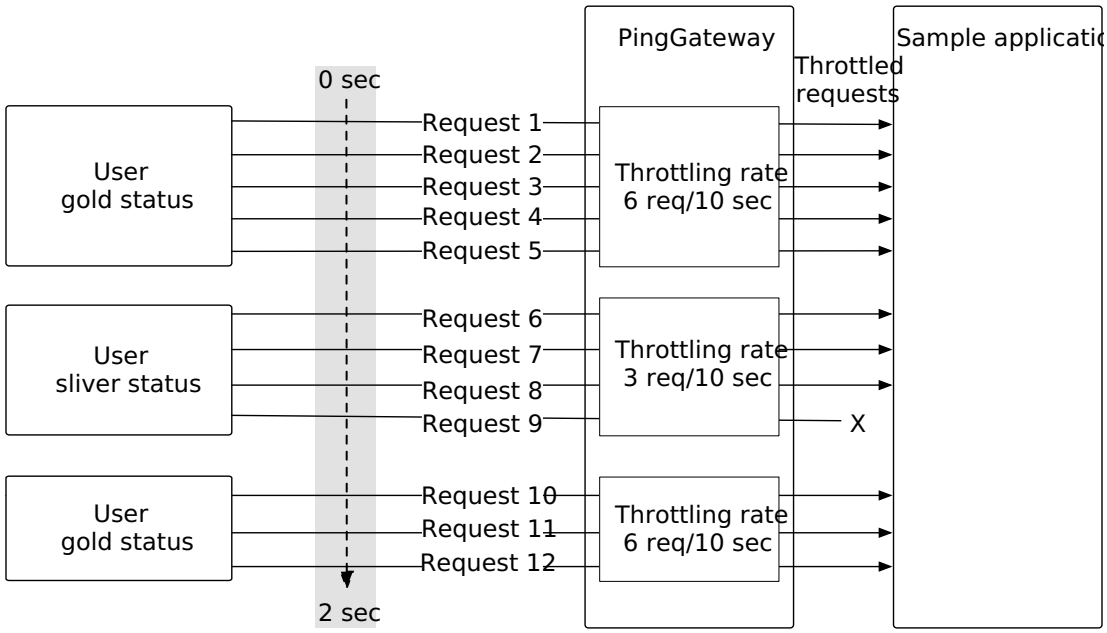
```
$ grep "< HTTP/2" /tmp/throttle-mapped.txt | sort | uniq -c

6 < HTTP/2 200
4 < HTTP/2 429
```

   Notice that with a `gold` status, the user can access the route 6 times in 10 seconds.

4. In AM, change the demo user's email to `demo@other.com`, and then run the last two steps again to find how the access is reduced.

Considerations for dynamic throttling

The following image illustrates what can happen when the throttling rate defined by `throttlingRateMapping` changes frequently or quickly:



In the image, the user starts out with a `gold` status. In a two second period, the users sends five requests, is downgraded to silver, sends four requests, is upgraded back to `gold`, and then sends three more requests.

After making five requests with a `gold` status, the user has almost reached their throttling rate. When his status is downgraded to silver, those requests are disregarded and the full throttling rate for `silver` is applied. The user can now make three more requests even though they have nearly reached their throttling rate with a `gold` status.

After making three requests with a `silver` status, the user has reached their throttling rate. When the user makes a fourth request, the request is refused.

The user is now upgraded back to `gold` and can now make six more requests even though they had reached his throttling rate with a `silver` status.

When you configure `requestGroupingPolicy` and `throttlingRateMapper`, bear in mind what happens when the throttling rate defined by the `throttlingRateMapper` is changed.

Configure scriptable throttling

This section builds on the example in Configure mapped throttling. It creates a scriptable throttling filter, where the script applies a throttling rate of 6 requests/10 seconds to requests from gold status users. For all other requests, the script returns `null`, and applies the default rate of 1 request/10 seconds.

Before you start, set up and test the example in Configure mapped throttling.

1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/00-throttle-scriptable.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-throttle-scriptable.json
```

```
{
  "name": "00-throttle-scriptable",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/throttle-scriptable')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "token-resolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
        }
```

```
        },
        {
          "name": "ThrottlingFilter-1",
          "type": "ThrottlingFilter",
          "config": {
            "requestGroupingPolicy": "${contexts.oauth2.accessToken.info.mail}",
            "throttlingRatePolicy": {
              "type": "DefaultRateThrottlingPolicy",
              "config": {
                "delegateThrottlingRatePolicy": {
                  "name": "ScriptedPolicy",
                  "type": "ScriptableThrottlingPolicy",
                  "config": {
                    "type": "application/x-groovy",
                    "source": [
                      "if (contexts.oauth2.accessToken.info.status == status) {",
                      "  return new ThrottlingRate(rate, duration)",
                      "} else {",
                      "  return null",
                      "}"
                    ],
                    "args": {
                      "status": "gold",
                      "rate": 6,
                      "duration": "10 seconds"
                    }
                  }
                },
                "defaultRate": {
                  "numberOfRequests": 1,
                  "duration": "10 s"
                }
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the IG route in Studio, refer to Scriptable throttling filter in Structured Editor.

Notice the following features of the route, compared to path] `00-throttle-mapped.json`:

- The route matches requests to `/home/throttle-scriptable`.
- The DefaultRateThrottlingPolicy delegates the management of throttling to the ScriptableThrottlingPolicy.
- The script applies a throttling rate to requests from users with gold status. For all other requests, the script returns null and the default rate is applied.

2. Test the setup:

    1. In a terminal window, use a `curl` command similar to this to retrieve an access token:

```
$ mytoken=$(curl -s \
--user "client-application:password" \
--data "grant_type=password&username=demo&password=Ch4ng31t&scope=mail%20employeenumber" \
http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
```

2. Using the access token, access the route multiple times. The following example accesses the route 10 times and writes the output to a file:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/home/throttle-scriptable/\[01-10\] \
> /tmp/throttle-script.txt 2>&1
```

3. Search the output file for the result:

```
$ grep "< HTTP/2" /tmp/throttle-script.txt | sort | uniq -c

6 < HTTP/2 200
4 < HTTP/2 429
```

   Notice that with a `gold` status, the user can access the route 6 times in 10 seconds.

4. In AM, change the user's email to `demo@other.com`, and then run the last two steps again to find how the access is reduced.

URI fragments in redirect

URI fragments are optional last parts of a URL for a document, typically used to identify or navigate to a particular part of the document. The fragment part follows the URL after a hash `#`, for example `https://www.rfc-editor.org/rfc/rfc1234#section5`.

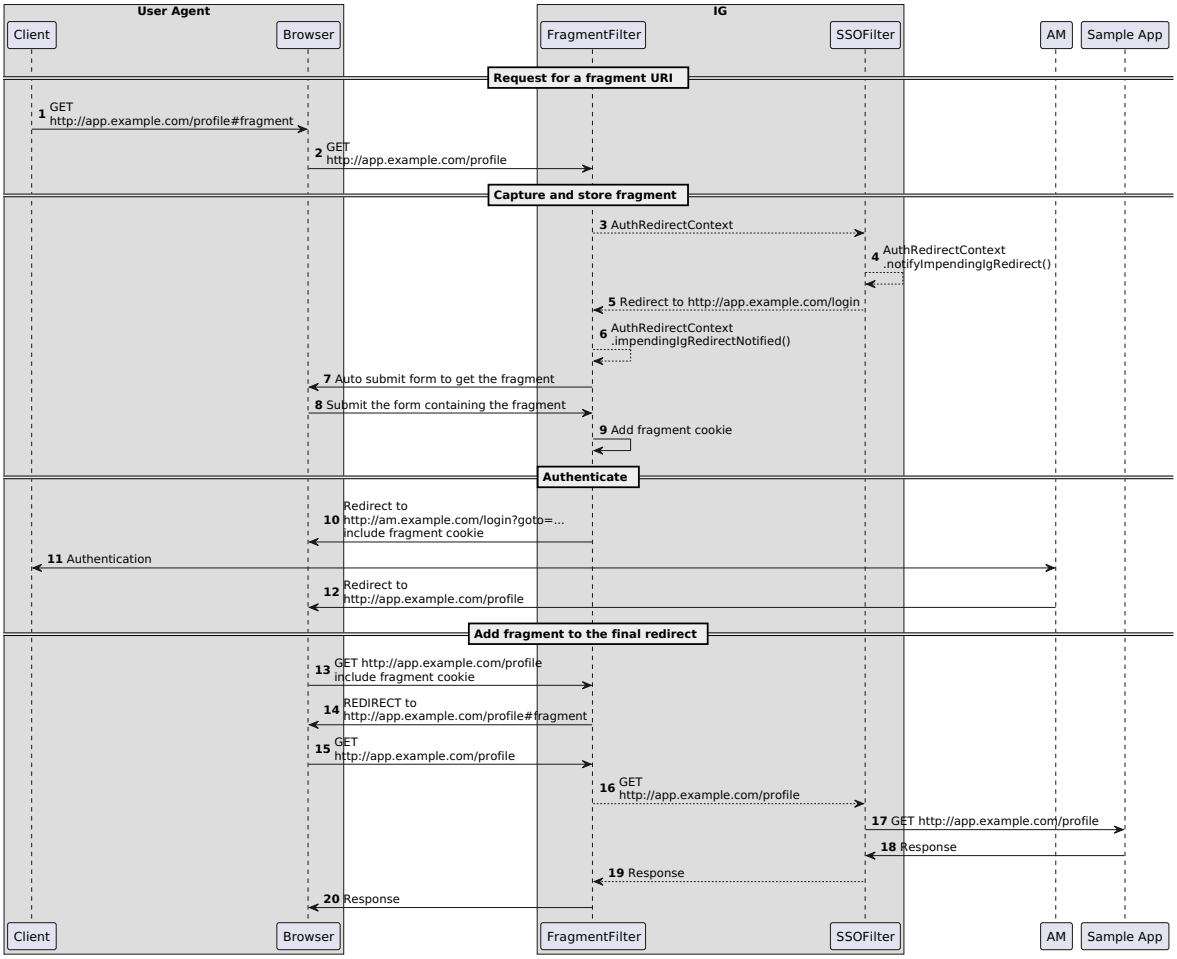
When an unauthenticated user requests a resource that includes a URI fragment, the user agent sends the URI but doesn't send the fragment. The fragment is lost during the authentication flow.

IG provides a FragmentFilter to track the fragment part of a URI when a request triggers a login redirect.

The FragmentFilter doesn't handle multiple fragment captures in parallel. If a fragment capture is in progress while IG performs another login redirect, a second fragment capture process isn't triggered and the fragment is lost.

When a browser request loads a favicon, it can cause the fragment part of a URI to be lost. Prevent problems by serving static resources with a separate route. As an example, use the route in Serve static resources.

The following image shows the flow of information when the FragmentFilter is included in the SSO authentication flow:



1-2. An unauthenticated client requests access to a fragment URL.

3. The FragmentFilter adds the AuthRedirectContext, so that downstream filters can mark the response as redirected.

4-5. The SingleSignOnFilter adds to the context to notify upstream filters that a redirect is pending, and redirects the request for authentication.

6-7. The FragmentFilter is notified by the context that a redirect is pending, and returns a new response object containing the response cookies, an autosubmit HTML form, and Javascript.

8. The user agent runs the Javascript or displays the form's submit button for the user to click on. This operation POSTs a form request back to a fragment endpoint URI, containing the following parts:

- Request URI path (`/profile`)
- Captured fragment (`#fragment`)
- Login URI (`http://am.example.com/login?goto=…`)

9. The FragmentFilter creates the fragment cookie.

10-12. The client authenticates with AM.

13. The FragmentFilter intercepts the request because it contains a fragment cookie, and its URI matches the original request URI.

The filter redirects the client to the original request URI containing the fragment. The fragment cookie then expires.

14-19. The client follows the final redirect to the original request URI containing the fragment, and the sample app returns the response.

This procedure shows how to persist a URI fragment in an SSO authentication. Before you start, set up and test the example in Authenticate with SSO through the default authentication service.

1. In IG, replace sso.json with the following route:

Linux

```
$HOME/.openig/config/routes/fragment.json
```

Windows

```
%appdata%\OpenIG\config\routes\fragment.json
```

```
{
  "name": "fragment",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/sso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "FragmentFilter-1",
          "type": "FragmentFilter",
          "config": {
            "fragmentCaptureEndpoint": "/home/sso"
          }
        },
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following feature of the route compared to `sso.json`:

- The `FragmentFilter` captures the fragment form data from the route condition endpoint.

2. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/sso#fragment.

      The SingleSignOnFilter redirects the request to AM for authentication.

   2. Log in to AM as user `demo`, password `Ch4ng31t`.

The SingleSignOnFilter passes the request to sample application, which returns the home page. Note that the URL of the page has preserved the fragment: `https://ig.example.com:8443/home/sso?_ig=true#fragment`

3. Remove the FragmentFilter from the route and test the route again.

   Note that this time the URL of the page hasn't preserved the fragment.

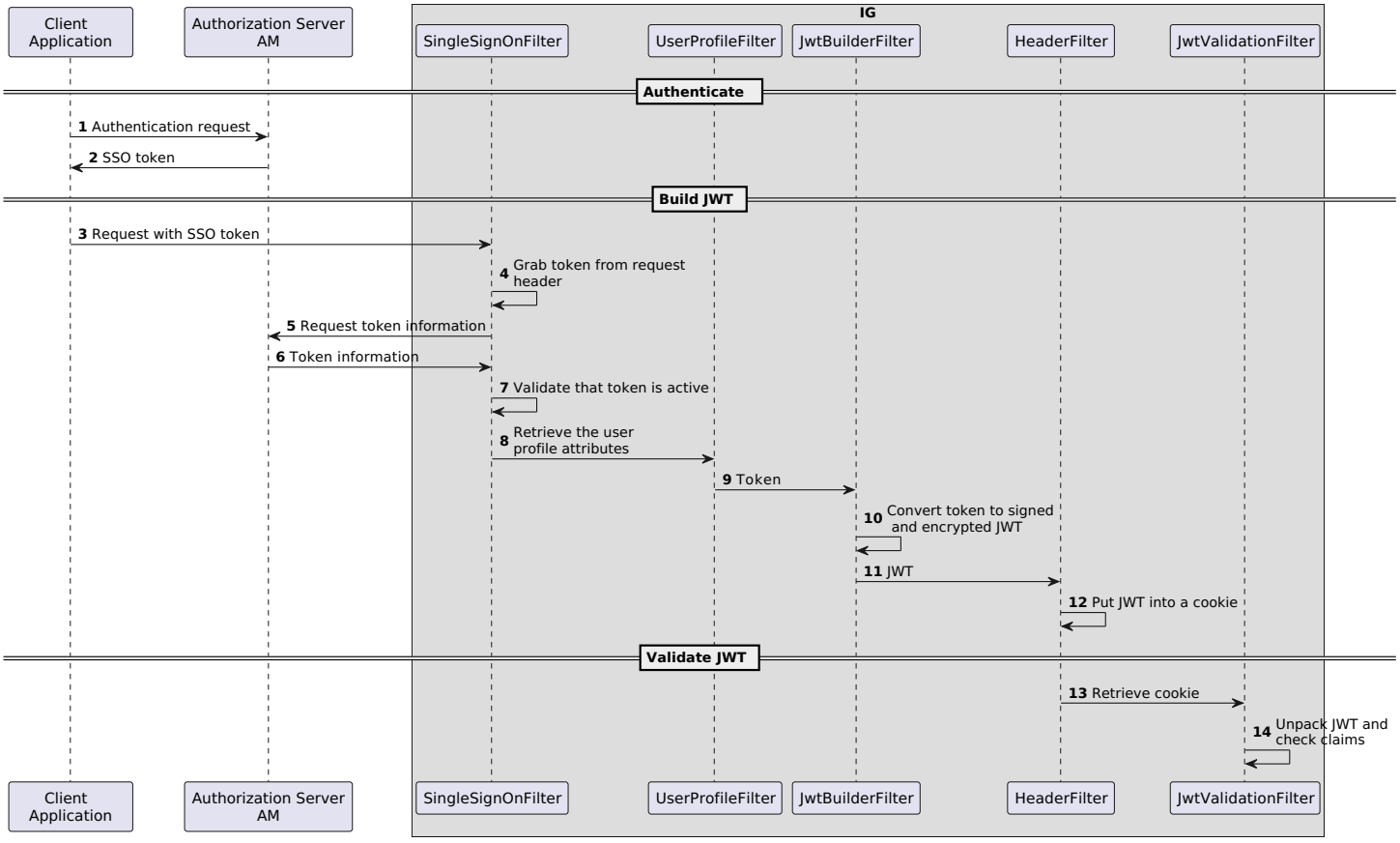
JWT validation

The following examples show how to use the JwtValidationFilter to validate signed and encrypted JWT.

The JwtValidationFilter can access JWTs in the request, provided in a header, query parameter, form parameter, cookie, or other way. If an upstream filter makes the JWT available in the request's attributes context, the JwtValidationFilter can access the JWT through the context, for example, at `${attributes.jwtToValidate}`.

For convenience, the JWT in this example is provided by the JwtBuilderFilter, and passed to the JwtValidationFilter in a cookie.

The following figure shows the flow of information in the example:



Before you start, set up and test the example in Pass runtime data in a JWT signed with PEM then encrypted with a symmetric key.

1. Add a second route to IG, replacing value of the property `secretsDir` with the directory for the PEM files:

Linux

```
$HOME/.openig/config/routes/jwt-validate.json
```

Windows

```
%appdata%\OpenIG\config\routes\jwt-validate.json
```

```
{
  "name": "jwt-validate",
  "condition": "${find(request.uri.path, '^/jwt-validate')}",
  "properties": {
    "secretsDir": "path/to/secrets"
  },
  "capture": "all",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore",
      "type": "SystemAndEnvSecretStore",
      "config": {
        "mappings": [{
          "secretId": "id.decrypted.key.for.signing.jwt",
          "format": "BASE64"
        }]
      }
    },
    {
      "name": "pemPropertyFormat",
      "type": "PemPropertyFormat",
      "config": {
        "decryptionSecretId": "id.decrypted.key.for.signing.jwt",
        "secretsProvider": "SystemAndEnvSecretStore"
      }
    },
    {
      "name": "FileSystemSecretStore-1",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "&{secretsDir}",
        "mappings": [{
          "secretId": "id.encrypted.key.for.signing.jwt.pem",
          "format": "pemPropertyFormat"
        }, {
          "secretId": "symmetric.key.for.encrypting.jwt",
          "format": {
            "type": "SecretKeyPropertyFormat",
            "config": {
              "format": "BASE64",
              "algorithm": "AES"
            }
          }
        }]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "type": "JwtValidationFilter",
        "config": {
          "jwt": "${request.cookies['my-jwt'][0].value}",
          "secretsProvider": "FileSystemSecretStore-1",
          "decryptionSecretId": "symmetric.key.for.encrypting.jwt",
          "customizer": {
            "type": "ScriptableJwtValidatorCustomizer",
            "config": {
```

```
              "type": "application/x-groovy",
              "source": [
                "builder.claim('name', JsonValue::asString, isEqualTo('demo'))",
                "builder.claim('email', JsonValue::asString, isEqualTo('demo@example.com'));"
              ]
            }
          },
          "failureHandler": {
            "type": "ScriptableHandler",
            "config": {
              "type": "application/x-groovy",
              "source": [
                "def response = new Response(Status.FORBIDDEN)",
                "response.headers['Content-Type'] = 'text/html; charset=utf-8'",
                "def errors = contexts.jwtValidationError.violations.collect{it.description}",
                "def display = \"<html>Can't validate JWT:<br> ${contexts.jwtValidationError.jwt} \"",
                "display <<=\"<br><br>For the following errors:<br> ${errors.join(\"<br>\")}</html>\"",
                "response.entity=display as String",
                "return response"
              ]
            }
          }
        }
      }],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": [
            "<html>",
            "  <h2>Validated JWT:</h2>",
            "    <p>${contexts.jwtValidation.value}</p>",
            "  <h2>JWT payload:</h2>",
            "    <p>${contexts.jwtValidation.info}</p>",
            "</html>"
          ]
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/jwt-validate`.
- The JwtValidationFilter takes the value of the JWT from `my-jwt`.
- The SystemAndEnvSecretStore, PemPropertyFormat, and FileSystemSecretStore objects in the heap are the same as those in the route to create the JWT. The JwtValidationFilter uses the same objects to validate the JWT.
- The JwtBuilderFilter `customizer` requires that the JWT claims match `name:demo` and `email:demo@example.com`.
- If the JWT is validated, the StaticResponseHandler displays the validated value. Otherwise, the FailureHandler displays the reason for the failed validation.

2. Test the setup:

    1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/jwtbuilder-sign-then-encrypt to build a JWT.
    2. Log in to AM as user `demo`, password `Ch4ng31t`. The sample application displays the signed JWT.
    3. Go to https://ig.example.com:8443/jwt-validate to validate the JWT. The validated JWT and its payload are displayed.
    4. Test the setup again, but log in to AM as a different user, or change the email address of the demo user in AM. The JWT isn't validated, and an error is displayed.

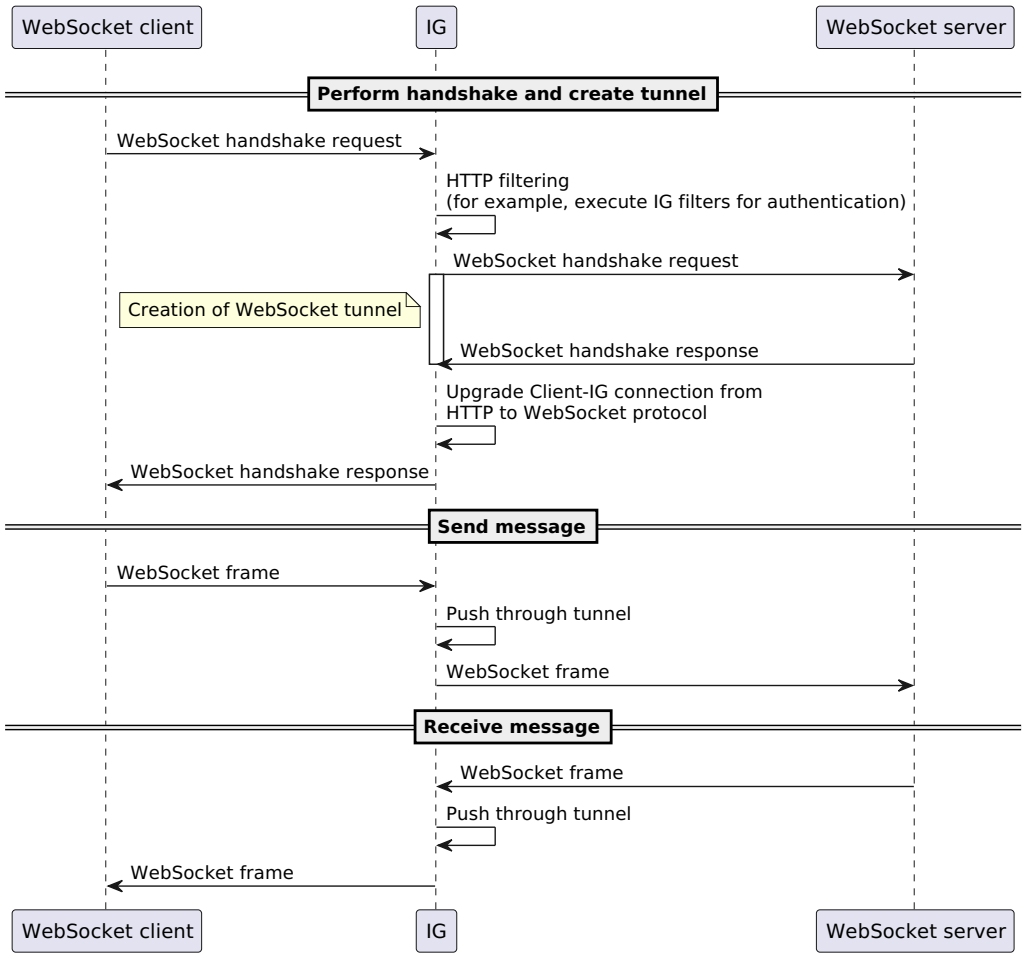
WebSocket traffic

When a user agent requests an upgrade from HTTP or HTTPS to the WebSocket protocol, IG detects the request and performs an HTTP handshake request between the user agent and the protected application.

If the handshake is successful, IG upgrades the connection and provides a dedicated tunnel to route WebSocket traffic between the user agent and the protected application. IG does not intercept messages to or from the WebSocket server.

The tunnel remains open until it is closed by the user agent or protected application. When the user agent closes the tunnel, the connection between IG and the protected application is automatically closed.

The following sequence diagram shows the flow of information when IG proxies WebSocket traffic:



To configure IG to proxy WebSocket traffic, configure the `websocket` property of ReverseProxyHandler. By default, IG does not proxy WebSocket traffic.

Proxy WebSocket traffic

1. Set up AM:
    1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
        - `https://ig.example.com:8443/*`
        - `https://ig.example.com:8443/*?*`
    2. Register an IG agent with the following values, as described in Register an IG agent in AM:
        - **Agent ID**: `ig_agent`
        - **Password**: `password`

        > **Important**
        >
        > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

    3. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

> **Important**
>
> IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.

2. Set up IG:

   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

   2. Set an environment variable for the IG agent password, and then restart IG:

      ```
      $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
      ```

      The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

   3. Add the following route to IG to serve the sample application .css and other static resources:

      Linux

      ```
      $HOME/.openig/config/routes/00-static-resources.json
      ```

      Windows

      ```
      %appdata%\OpenIG\config\routes\00-static-resources.json
      ```

      ```
      {
        "name" : "00-static-resources",
        "baseURI" : "http://app.example.com:8081",
        "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
      or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
        "handler": "ReverseProxyHandler"
      }
      ```

   4. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/websocket.json
```

Windows

```
%appdata%\OpenIG\config\routes\websocket.json
```

```
{
  "name": "websocket",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/websocket')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    },
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "config": {
        "websocket": {
          "enabled": true
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For information about how to set up the route in Studio, refer to Proxy for WebSocket traffic in Structured Editor.

Notice the following features of the route:

- The route matches requests to `/websocket`, the endpoint on the sample app that exposes a WebSocket server.
- The SingleSignOnFilter redirects unauthenticated requests to AM for authentication.

- The ReverserProxyHandler enables IG to proxy WebSocket traffic. After IG upgrades the HTTP connection to the WebSocket protocol, the ReverserProxyHandler passes the messages to the WebSocket server.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/websocket.
   2. Log in to AM as user `demo`, password `Ch4ng31t`.

      AM authenticates the user, creates an SSO token, and redirects the request back to the original URI, with the token in a cookie.

      The request then passes to the ReverseProxyHandler, which routes the request to the HTML page `/websocket/index.html` of the sample app. The page initiates the HTTP handshake for connecting to the WebSocket endpoint `/websocket/echo`.
   3. Enter text on the WebSocket echo screen and note that the text is echoed back.

Vert.x-specific configuration for WebSocket connections

Configure Vert.x-specific configuration for WebSocket connections, where IG does not provide its own first-class configuration. Vert.x options are described in HttpClientOptions.

The following example configures Vert.x options for Websocket connections:

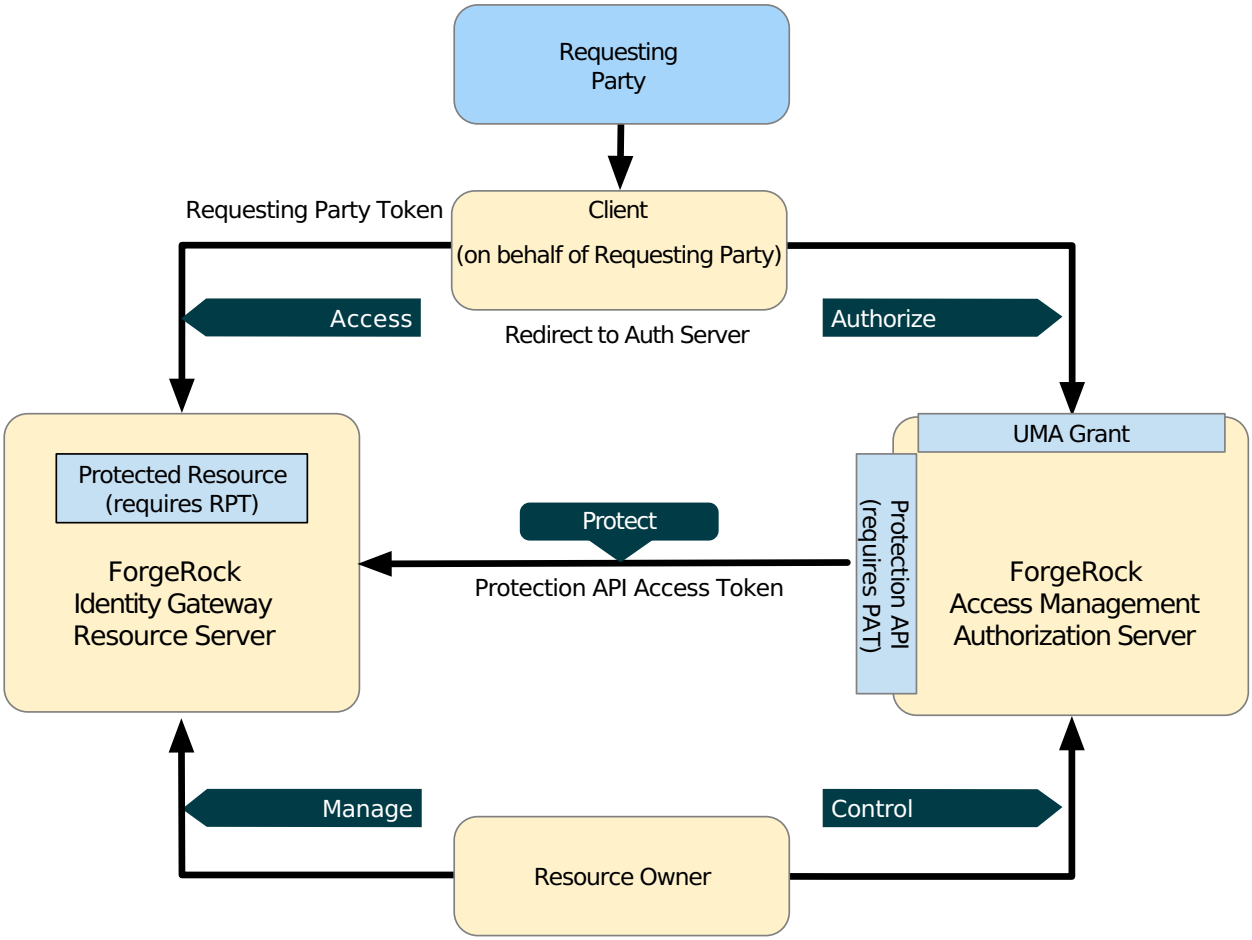
```
{
  "type": "ReverseProxyHandler",
  "config": {
    "websocket": {
      "enabled": true,
      "vertx": {
        "maxFrameSize": 200000000,
        "maxMessageSize": 200000000,
        "tryUsePerMessageCompression": true
      }
    }
  }
}
```

UMA support

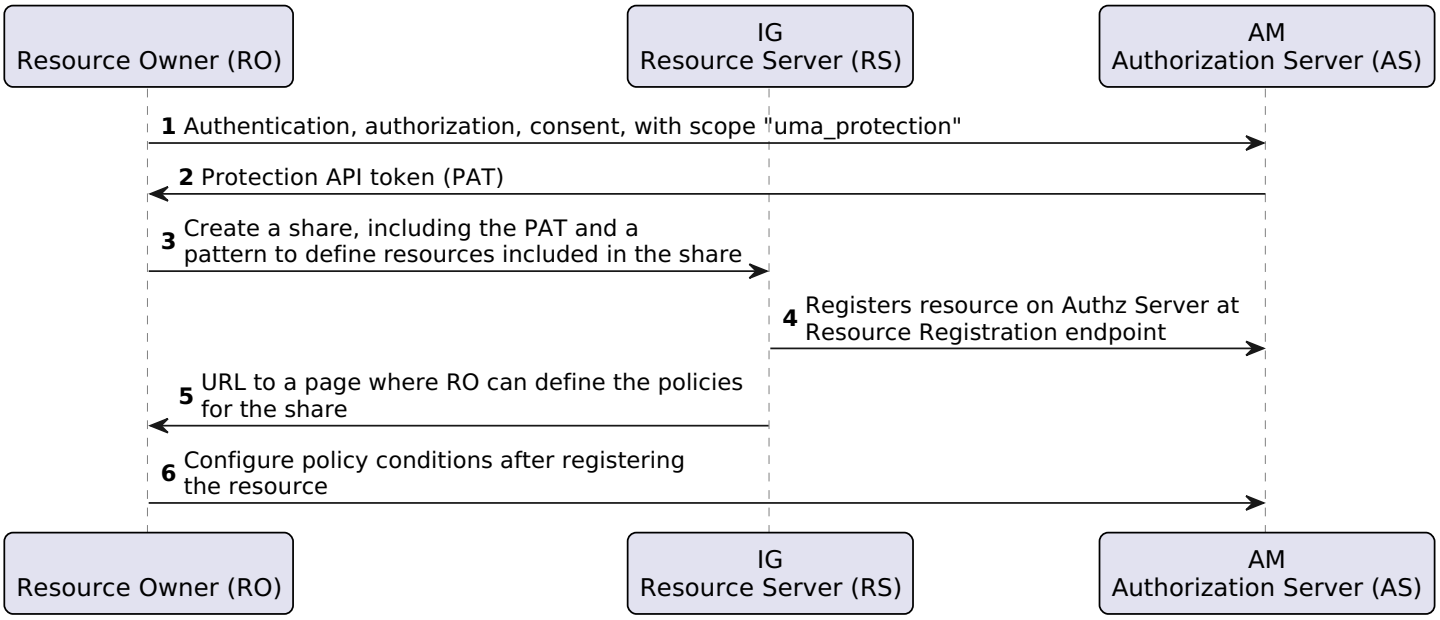
IG includes support for User-Managed Access (UMA) 2.0 Grant for OAuth 2.0 Authorization specifications.

About IG as an UMA resource server

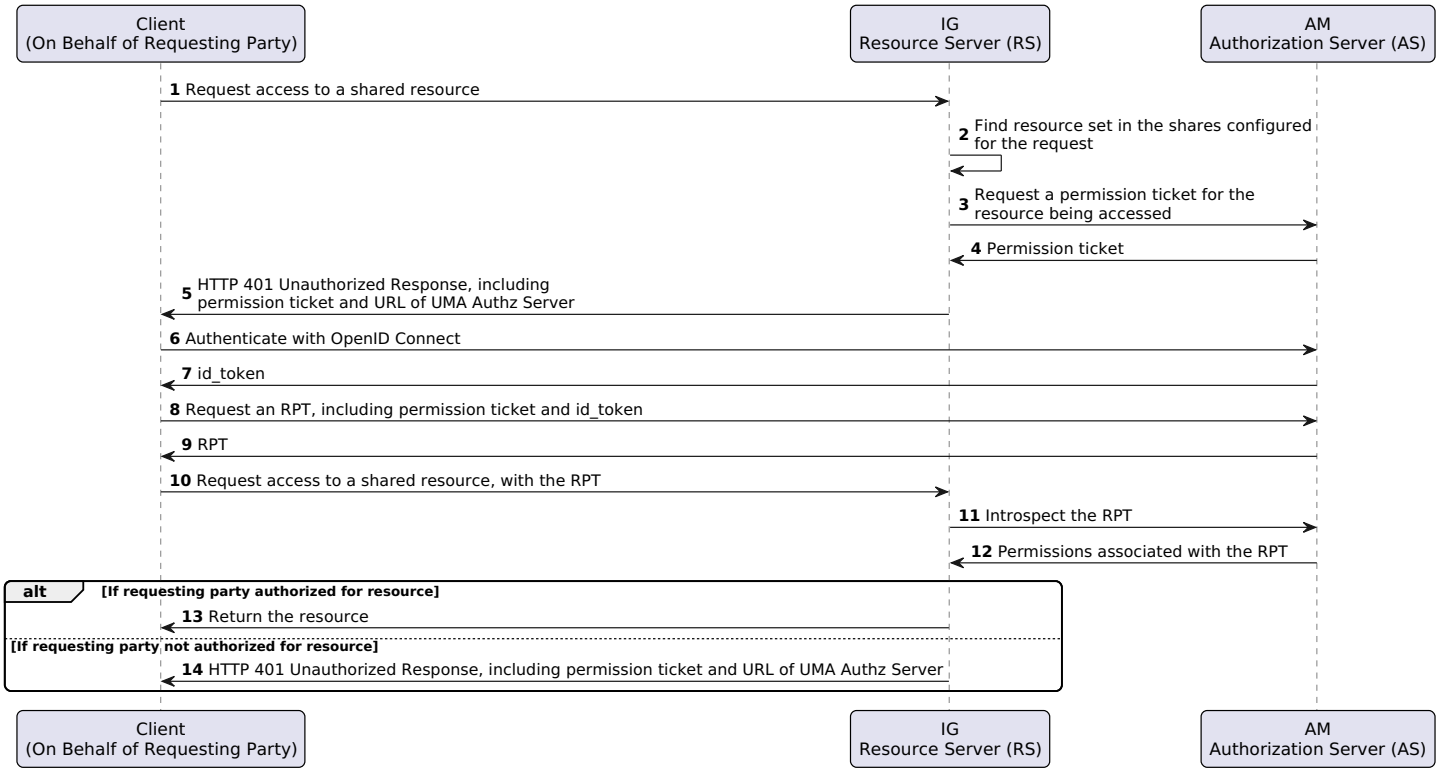
The following figure shows an UMA environment, with IG protecting a resource, and AM acting as an Authorization Server. For information about UMA, refer to AM's User-Managed Access (UMA) 2.0 guide.



The following figure shows the data flow when the resource owner registers a resource with AM, and sets up a share using a Protection API Token (PAT):



The following figure shows the data flow when the client accesses the resource, using a Requesting Party Token (RPT):



For information about CORS support, refer to Configure CORS support in AM's *Security guide*. This procedure describes how to modify the AM configuration to allow cross-site access.

Limitations of IG as an UMA resource server

When using IG as an UMA resource server, note the following points:

- IG depends on the resource owner for the PAT.

  When a PAT expires, no refresh token is available to IG. The resource owner must repeat the entire share process with a new PAT in order to authorize access to protected resources. The resource owner should delete the old resource and create a new one.

- Data about PATs and shared resources is held in memory.

  IG has no mechanism for persisting the data across restarts. When IG stops and starts again, the resource owner must repeat the entire share process.

- UMA client applications for sharing and accessing protected resources must deal with UMA error conditions and IG error conditions.

- By default, the REST API to manage share objects exposed by IG is protected only by CORS.

- When matching protected resource paths with share patterns, IG takes the longest match.

  For example, if resource owner Alice shares `/photos/.*` with Bob, and `/photos/vacation.png` with Charlie, and then Bob attempts to access `/photos/vacation.png`, IG applies the sharing permissions for Charlie, not Bob. As a result, Bob can be denied access.

Set up the UMA example

This section describes tasks to set up AM as an Authorization Server:

- Enabling cross-origin resource sharing (CORS) support in AM
- Configuring AM as an Authorization Server
- Registering UMA client profiles with AM
- Setting up a resource owner (Alice) and requesting party (Bob)

Caution

The settings in this section are suggestions for this tutorial. They are not intended as instructions for setting up AM CORS support on a server in production.
If you need to accept all origins, by allowing the use of `Access-Control-Allowed-Origin=*`, do not allow `Content-Type` headers. Allowing the use of both types of headers exposes AM to cross-site request forgery (CSRF) attacks.

Important

This procedure uses the *Resource Owner Password Credentials* grant type. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize other grant types whenever possible.

Before you start, prepare AM, IG, and the sample application as described in Example installation for this guide.

If you use different settings for the sample application, refer to Edit the example to match custom settings.

1. Set up AM:
    1. Find the name of the AM session cookie at the `/json/serverinfo/*` endpoint. This procedure assumes that you are using the default AM session cookie, `iPlanetDirectoryPro`.
    2. Create an OAuth 2.0 Authorization Server:
        1. Select **Services** > **Add a Service** > **OAuth2 Provider**.
        2. Add a service with the default values.
    3. Configure an UMA Authorization Server:
        1. Select **Services** > **Add a Service** > **UMA Provider**.
        2. Add a service with the default values.
    4. Add an OAuth 2.0 client for UMA protection:
        1. Select **Applications** > **OAuth 2.0** > **Clients**.
        2. Add a client with these values:
            - **Client ID** : `OpenIG`
            - **Client secret** : `password`
            - **Scope** : `uma_protection`

3. On the **Advanced** tab, select the following option:
    - **Grant Types** : `Resource Owner Password Credentials`

5. Add an OAuth 2.0 client for accessing protected resources:
    1. Select **Applications** > **OAuth 2.0** > **Clients**.
    2. Add a client with these values:
        - **Client ID** : `UmaClient`
        - **Client secret** : `password`
        - **Scope** : `openid`
    3. On the **Advanced** tab, select the following option:
        - **Grant Types** : `Resource Owner Password Credentials` and `UMA`
6. Select **Identities**, and add an identity for a resource owner, with the following values:
    - **ID** : `alice`
    - **Password** : `UMAexamp1e`
    - **Email Address** : `alice@example.com`
7. Select **Identities**, and add an identity for a requesting party, with the following values:
    - **ID** : `bob`
    - **Password** : `UMAexamp1e`
    - **Email Address** : `bob@example.com`
8. Enable the CORS filter on AM:
    1. In a terminal window, retrieve an SSO token from AM:

```
$ mytoken=$(curl --request POST \
--header "Accept-API-Version: resource=2.1" \
--header "X-OpenAM-Username: amadmin" \
--header "X-OpenAM-Password: password" \
--header "Content-Type: application/json" \
--data "{}"  \
http://am.example.com:8088/openam/json/authenticate | jq -r ".tokenId")
```

    2. Using the token retrieved in the previous step, enable the CORS filter on AM, by using the use the `/global-config/services/CorsService` REST endpoint:

```
$ curl  \
  --request PUT \
  --header "Content-Type: application/json" \
  --header "iPlanetDirectoryPro: $mytoken" http://am.example.com:8088/openam/json/global-config/
services/CorsService/configuration/CorsService \
  --data '{
     "acceptedMethods": [
       "POST",
       "GET",
       "PUT",
       "DELETE",
       "PATCH",
       "OPTIONS"
     ],
     "acceptedOrigins": [
       "http://app.example.com:8081",
       "http://ig.example.com:8080",
       "http://am.example.com:8088/openam"
     ],
     "allowCredentials": true,
     "acceptedHeaders": [
       "Authorization",
       "Content-Type",
       "iPlanetDirectoryPro",
       "X-OpenAM-Username",
       "X-OpenAM-Password",
       "Accept",
       "Accept-Encoding",
       "Connection",
       "Content-Length",
       "Host",
       "Origin",
       "User-Agent",
       "Accept-Language",
       "Referer",
       "Dnt",
       "Accept-Api-Version",
       "If-None-Match",
       "Cookie",
       "X-Requested-With",
       "Cache-Control",
       "X-Password",
       "X-Username",
       "X-NoSession"
     ],
     "exposedHeaders": [
       "Access-Control-Allow-Origin",
       "Access-Control-Allow-Credentials",
       "Set-Cookie",
       "WWW-Authenticate"
     ],
     "maxAge": 600,
     "enabled": true,
     "allowCredentials": true
  }'
```

A CORS configuration is diplayed.

Tip

To delete the CORS configuration and create another, first run the following command:

```
$ curl \
 --request DELETE \
 --header "X-Requested-With: XMLHttpRequest" \
 --header "iPlanetDirectoryPro: $mytoken" \
 http://am.example.com:8088/openam/json/global-config/services/CorsService/
CorsService/configuration/CorsService
```

2. Set up IG as an UMA resource server:

   1. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

   2. Add the following ClientHandler and ApiProtectionFilter to the heap in your `admin.json` configuration and restart IG:

```
{
  "prefix": "openig",
  "connectors": [
    { "port" : 8080 }
  ],
  "heap": [
    {
      "name": "ClientHandler",
      "type": "ClientHandler"
    },
    {
      "name": "ApiProtectionFilter",
      "type": "CorsFilter",
      "config": {
        "policies": [
          {
            "acceptedOrigins": [ "http://app.example.com:8081" ],
            "acceptedMethods": [ "GET", "POST", "DELETE" ],
            "acceptedHeaders": [ "Content-Type" ]
          }
        ]
      }
    }
  ]
}
```

Notice the following feature:

- The default ApiProtectionFilter is overridden by the CorsFilter, which allows requests from the origin `http://app.example.com:8081`.

3. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/00-uma.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-uma.json
```

```
{
  "name": "00-uma",
  "condition": "${request.uri.host == 'app.example.com'}",
  "heap": [
    {
      "name": "UmaService",
      "type": "UmaService",
      "config": {
        "protectionApiHandler": "ClientHandler",
        "wellKnownEndpoint": "http://am.example.com:8088/openam/uma/.well-known/uma2-configuration",
        "resources": [
          {
            "comment": "Protects all resources matching the following pattern.",
            "pattern": ".*",
            "actions": [
              {
                "scopes": [
                  "#read"
                ],
                "condition": "${request.method == 'GET'}"
              },
              {
                "scopes": [
                  "#create"
                ],
                "condition": "${request.method == 'POST'}"
              }
            ]
          }
        ]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "CorsFilter",
          "config": {
            "policies": [
              {
                "acceptedOrigins": [ "http://app.example.com:8081" ],
                "acceptedMethods": [ "GET" ],
                "acceptedHeaders": [ "Authorization" ],
                "exposedHeaders": [ "WWW-Authenticate" ],
                "allowCredentials": true
              }
            ]
          }
        },
        {
          "type": "UmaFilter",
          "config": {
            "protectionApiHandler": "ClientHandler",
            "umaService": "UmaService"
          }
        }
```

```
        ],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

Notice the following features of the route:

- The route matches requests from `app.example.com`.
- The UmaService describes the resources that a resource owner can share, using AM as the Authorization Server. It provides a REST API to manage sharing of resource sets.
- The CorsFilter defines the policy for cross-origin requests, listing the methods and headers that the request can use, the headers that are exposed to the frontend JavaScript code, and whether the request can use credentials.
- The UmaFilter manages requesting party access to protected resources, using the UmaService. Protected resources are on the sample application, which responds to requests on port 8081.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to http://app.example.com:8081/uma/.
   2. Share resources:
      1. Select **Alice shares resources**.
      2. On Alice's page, select **Share with Bob**. The following items are displayed:
         - The PAT that Alice receives from AM.
         - The metadata for the resource set that Alice registers through IG.
         - The result of Alice authenticating with AM in order to create a policy.
         - The successful result when Alice configures the authorization policy attached to the shared resource.

         If the step fails, run the following command to get an access token for Alice:

         ```
         $ curl -X POST \
         -H "Cache-Control: no-cache" \
         -H "Content-Type: application/x-www-form-urlencoded" \
         -d
         'grant_type=password&scope=uma_protection&username=alice&password=UMAexamp1e&client_id=Op
         enIG&client_secret=password' \
         http://am.example.com:8088/openam/oauth2/access_token
         ```

         If you fail to get an access token, check that AM is configured as described in this procedure. If you continue to have problems, make sure that your IG configuration matches that shown when you are running the test on http://app.example.com:8081/uma/.
   3. Access resources:
      1. Go back to the first page, and select **Bob accesses resources**.

2. On Bob's page, select **Get Alice's resources**. The following items are displayed:
   - The WWW-Authenticate Header.
   - The OpenID Connect Token that Bob gets to obtain the RPT.
   - The RPT that Bob gets in order to request the resource again.
   - The final response containing the body of the resource.

Edit the example to match custom settings

If you use a configuration that is different to that described in this chapter, consider the following tasks to adjust the sample to your configuration:

1. Unpack the UMA files from the sample application described in Use the sample application to temporary folder:

```
$ mkdir /tmp/uma
$ cd /tmp/uma
$ jar -xvf /path/to/IG-sample-application-2024.3.0-jar-with-dependencies.jar webroot-uma

created: webroot-uma/
inflated: webroot-uma/bob.html
inflated: webroot-uma/common.js
inflated: webroot-uma/alice.html
inflated: webroot-uma/index.html
inflated: webroot-uma/style.css
```

2. Edit the configuration in `common.js`, `alice.html`, and `bob.html` to match your settings.

3. Repack the UMA sample client files and then restart the sample application:

```
$ jar -uvf /path/to/IG-sample-application-2024.3.0-jar-with-dependencies.jar webroot-uma

adding: webroot-uma/(in = 0) (out= 0)(stored 0%)
adding: webroot-uma/bob.html(in = 26458) (out= 17273)(deflated 34%)
adding: webroot-uma/common.js(in = 3652) (out= 1071)(deflated 70%)
adding: webroot-uma/alice.html(in = 27775) (out= 17512)(deflated 36%)
adding: webroot-uma/index.html(in = 22046) (out= 16060)(deflated 27%)
adding: webroot-uma/style.css(in = 811) (out= 416)(deflated 48%)
updated module-info: module-info.class
```

4. If necessary, adjust the CORS settings for AM.

Understand the UMA API with an API descriptor

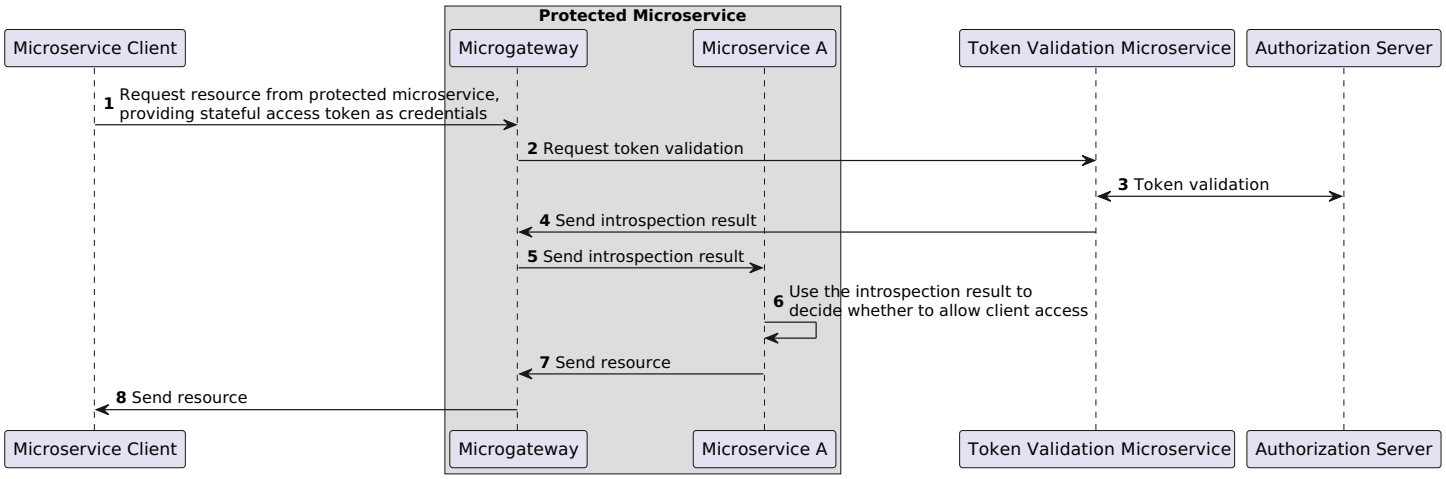
The UMA share endpoint serves API descriptors at runtime. When you retrieve an API descriptor for the endpoint, a JSON that describes the API for the endpoint is returned.

You can use the API descriptor with a tool such as Swagger UI to generate a web page that helps you to view and test the endpoint. For information, refer to API descriptors.

IG as a microgateway

This section describes how to use the ForgeRock Token Validation Microservice to resolve and cache OAuth 2.0 access tokens when protecting API resources. The section is based on the example in Introspecting stateful access tokens, in the Token Validation Microservice's *User guide*.

For information about the architecture, refer to IG as a microgateway. The following figure illustrates the flow of information when a client requests access to a protected microservice, providing a stateful access token as credentials:



Before you start, download and run the sample application as described in Use the sample application. The sample application acts as Microservice A.

1. Set up the example in Introspect stateful access tokens, in the Token Validation Microservice's *User guide*.
2. In AM, edit the microservice client to add a scope to access the protected microservice:
    1. Select **Applications** > **OAuth 2.0** > **Clients**.
    2. Select `microservice-client`, and add the scope `microservice-A`.
3. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/mgw.json
```

Windows

```
%appdata%\OpenIG\config\routes\mgw.json
```

```
{
  "properties": {
    "introspectOAuth2Endpoint": "http://mstokval.example.com:9090"
  },
  "capture": "all",
  "name": "mgw",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/mgw')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "requireHttps": false,
            "accessTokenResolver": {
              "name": "TokenIntrospectionAccessTokenResolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "endpoint": "&{introspectOAuth2Endpoint}/introspect",
                "providerHandler": "ForgeRockClientHandler"
              }
            },
            "scopes": ["microservice-A"]
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to IG on `http://ig.example.com:8080/home/mgw`, and rebases them to the sample application, on `http://app.example.com:8081`.
- The OAuth2ResourceServerFilter expects an OAuth 2.0 access token in the header of the incoming authorization request, with the scope `microservice-A`.
- If the filter successfully validates the access token, the ReverseProxyHandler passes the request to the sample application.

4. Test the setup:

    1. With AM, IG, the Token Validation Microservice, and the sample application running, get an access token from AM, using the scope `microservice-A`:

```
$ mytoken=$(curl -s \
--request POST \
--url http://am.example.com:8088/openam/oauth2/access_token \
--user microservice-client:password \
--data grant_type=client_credentials \
--data scope=microservice-A --silent | jq -r .access_token)
```

2. View the access token:

```
$ echo $mytoken
```

3. Call IG to access microservice A:

```
$ curl -v --header "Authorization: Bearer ${mytoken}" http://ig.example.com:8080/home/mgw
```

The home page of the sample application is displayed.

Identity Cloud

This guide provides examples of how to integrate your business application and APIs with Identity Cloud for Single Sign-On and API Security, with ForgeRock Identity Gateway. It is for ForgeRock Identity Cloud evaluators, administrators, and architects.

Example installation for this guide

Unless otherwise stated, the examples in this guide assume the following installation:

- Identity Gateway installed on `http://ig.example.com:8080`, as described in the Install .
- Sample application installed on `http://app.example.com:8081`, as described in Use the sample application.
- The ForgeRock Identity Cloud, with the default configuration, as described in the ForgeRock Identity Cloud Docs.

When using the ForgeRock Identity Cloud, you need to know the value of the following properties:

- The root URL of your ForgeRock Identity Cloud. For example, `https://myTenant.forgeblocks.com`.

  The URL of the Access Management component of the ForgeRock Identity Cloud is the root URL of your Identity Cloud followed by `/am`. For example, `https://myTenant.forgeblocks.com/am`.

- The realm where you work. The examples in this document use `alpha`.

  Prefix each realm in the hierarchy with the `realms` keyword. For example, `/realms/root/realms/alpha`.

If you use a different configuration, substitute in the procedures accordingly.

Authenticate an IG agent to Identity Cloud

> **Important**
>
> IG agents are automatically authenticated to Identity Cloud by a non-configurable authentication module.
> Authentication chains and modules are deprecated in Identity Cloud and replaced by journeys.
> You can now authenticate IG agents to Identity Cloud with a journey. The procedure is currently optional, but will be required when authentication chains and modules are removed in a future release of Identity Cloud.
> For more information, refer to Identity Cloud's Journeys.

This section describes how to create a journey to authenticate an IG agent to Identity Cloud. The journey has the following requirements:

- It must be called `Agent`
- Its nodes must pass the agent credentials to the Agent Data Store Decision node.

When you define a journey in Identity Cloud, that same journey is used for all instances of IG, Java agent, and Web agent. Consider this point if you change the journey configuration.

1. Log in to the Identity Cloud admin UI as an administrator.
2. Click **Journeys** > **New Journey**.
3. Add a journey with the following information and click **Create journey**:
   - **Name**: `Agent`

- **Identity Object**: The user or device to authenticate.
- (Optional) **Description**: Authenticate an IG agent to Identity Cloud

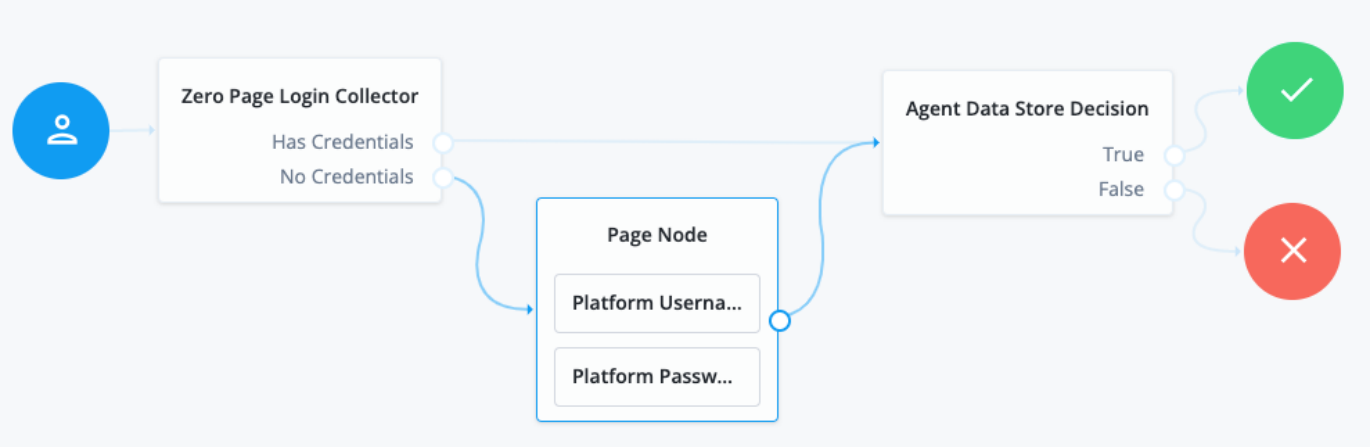
The journey designer is displayed, with the `Start` entry point connected to the `Failure` exit point, and a `Success` node.

4. Using the **Filter nodes** bar, find and then drag the following nodes from the **Components** panel into the designer area:
   - Zero Page Login Collector node to check whether the agent credentials are provided in the incoming authentication request, and use their values in the following nodes.

     This node is required for compatibility with Java agent and Web agent.
   - Page node to collect the agent credentials if they are not provided in the incoming authentication request, and use their values in the following nodes.
   - Agent Data Store Decision node to verify the agent credentials match the registered IG agent profile.

> **Important**
>
> Many nodes can be configured in the panel on the right side of the page. Unless otherwise stated, do not configure the nodes, and use only the default values.

5. Drag the following nodes from the **Components** panel into the Page node:
   - Platform Username node to prompt the user to enter their username.
   - Platform Password node to prompt the user to enter their password.
6. Connect the nodes as follows and save the journey:



Register an IG agent in Identity Cloud

This procedure registers an agent that acts on behalf of IG.

1. Log in to the Identity Cloud admin UI as an administrator.
2. Click **Gateways & Agents** > **New Gateway/Agent** > **Identity Gateway** > **Next**, and add an agent profile:
   - **ID**: agent-name

- **Password**: agent-password
- **Redirect URLs**: URL for CDSSO

> **Important**
>
> Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

3. Click **Save Profile** > **Done**. The agent profile page is displayed.
4. Click **Native Consoles** > **Access Management** and make the following optional changes in the AM admin UI.

Change	Action
Store the agent password in AM's secret service.	Set a **Secret Label Identifier**, and configure a mapping to the corresponding secret. If AM finds a matching secret in a secret store, it uses that secret instead of the agent password configured in Step 2. The secret label has the format `am.application.agents.identifier.secret`, where identifier is the **Secret Label Identifier**. The **Secret Label Identifier** can contain only characters `a-z`, `A-Z`, `0-9`, and periods (`.`). It can't start or end with a period. Note the following points: ◦ Set a **Secret Label Identifier** that clearly identifies the agent. ◦ If you update or delete the **Secret Label Identifier**, AM updates or deletes the corresponding mapping for the previous identifier provided no other agent shares the mapping. ◦ When you rotate a secret, update the corresponding mapping.
Direct login to a custom URL instead of the default AM login page.	Configure **Login URL Template for CDSSO**.
Apply a different introspection scope.	Click **Token Introspection** and select a scope from the drop-down list.

Set up a demo user in Identity Cloud

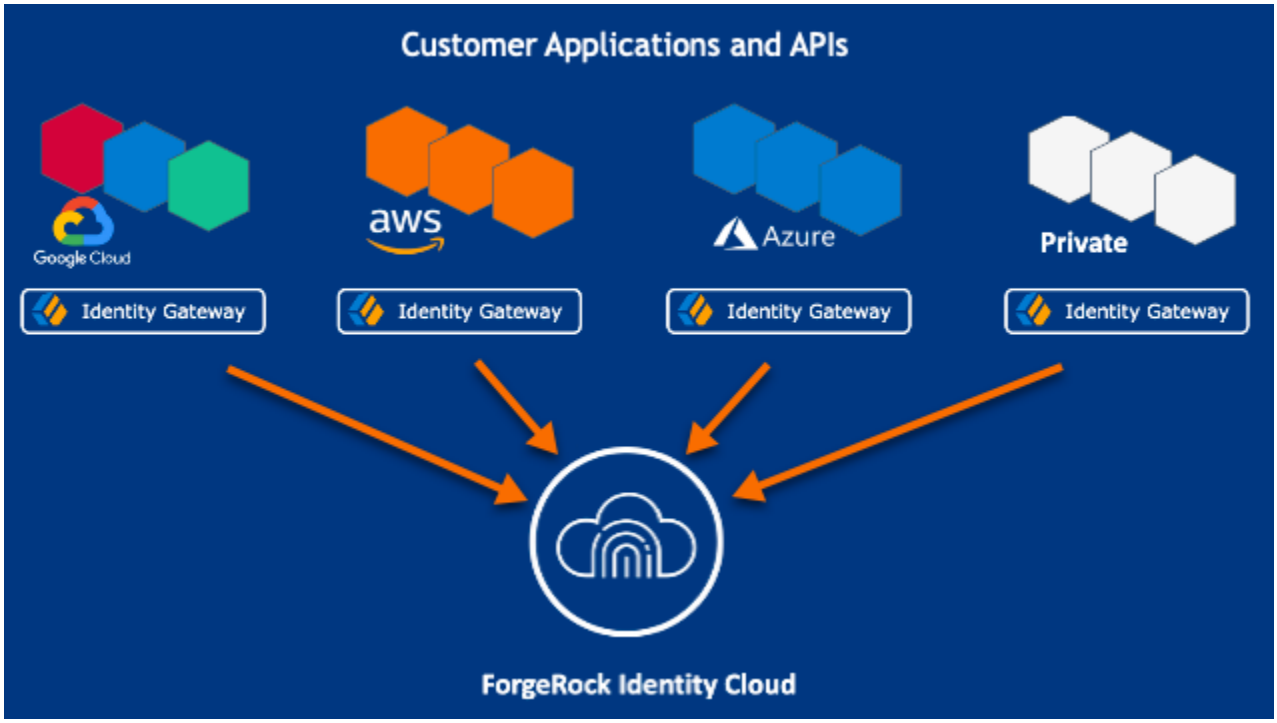
This procedure sets up a demo user in the alpha realm.

1. Log in to the Identity Cloud admin UI as an administrator.
2. Go to **Identities** > **Manage** > **Alpha realm - Users**, and add a user with the following values:
   - **Username**: `demo`
   - **First name**: `demo`
   - **Last name**: `user`
   - **Email Address**: `demo@example.com`
   - **Password**: `Ch4ng3!t`

About Identity Gateway and the ForgeRock Identity Cloud

ForgeRock Identity Cloud simplifies the consumption of ForgeRock as an Identity Platform. However, many organizations have business web applications and APIs deployed across multiple clouds, or on-premise.

Identity Gateway facilitates non-intrusive integration of your web applications and APIs with the Identity Cloud, for SSO and API Security. The following image illustrates how Identity Gateway bridges your business to the ForgeRock Identity Cloud:



For information about the ForgeRock Identity Cloud, refer to the ForgeRock Identity Cloud Docs.

OAuth 2.0

This example sets up OAuth 2.0, using the standard introspection endpoint, where ForgeRock Identity Cloud is the Authorization Server, and Identity Gateway is the resource server.

For more information about Identity Gateway as an OAuth 2.0 resource server, refer to Validate access tokens through the introspection endpoint.

> **Important**
>
> This procedure uses the *Resource Owner Password Credentials* grant type. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize other grant types whenever possible.

Before you start, prepare Identity Cloud, IG, and the sample application as described in Example installation for this guide.

1. Set up Identity Cloud:
    1. Log in to the Identity Cloud admin UI as an administrator.

2. Make sure you are managing the `alpha` realm. If not, click the current realm at the top of the screen, and switch realm.
3. Go to **Identities** > **Manage** > **Alpha realm - Users**, and add a user with the following values:
   - **Username**: `demo`
   - **First name**: `demo`
   - **Last name**: `user`
   - **Email Address**: `demo@example.com`
   - **Password**: `Ch4ng3!t`
4. Go to **Applications** > **+ Custom Application** > **OIDC - OpenId Connect** > **Web** and add a web application with the following values:
   - **Name**: `oauth2-client`
   - **Owners**: `demo user`
   - **Client Secret**: `password`
   - **Sign On** > **Grant Types**: `Authorization Code`, `Resource owner Password Credentials`
   - **Sign On** > **Scopes**: `mail`

     For more information, refer to Identity Cloud's Application management.
5. Register an IG agent with the following values, as described in Register an IG agent in Identity Cloud:
   - **ID**: `ig_agent`
   - **Password**: `password`
6. (Optional) Authenticate the agent to Identity Cloud as described in Authenticate an IG agent to Identity Cloud.

   Important

   IG agents are automatically authenticated to Identity Cloud by a deprecated authentication module in Identity Cloud. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of Identity Cloud.

2. Set up Identity Gateway:
   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
   2. Set an environment variable for the IG agent password, and then restart IG:

      ```
      $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
      ```

      The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.
   3. Add the following route to Identity Gateway, replacing the value for the property `amInstanceUrl`:

Linux

```
$HOME/.openig/config/routes/oauth2rs-idc.json
```

Windows

```
%appdata%\OpenIG\config\routes\oauth2rs-idc.json
```

```
{
  "name": "oauth2rs-idc",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/oauth2rs-idc')}",
  "properties": {
    "amInstanceUrl": "https://myTenant.forgeblocks.com/am"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "&{amInstanceUrl}",
        "realm": "/alpha",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "TokenIntrospectionAccessTokenResolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
```

```
              }
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></
body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route compared to `rs-introspect.json` in Validate access tokens through the introspection endpoint, where a local Access Management instance is the Authorization Server:

- The AmService `URL` points to Access Management in the Identity Cloud.
- The AmService `realm` points to the realm where you have configured your web application and the IG agent.

3. Test the setup:

   1. In a terminal, export an environment variable for the URL of Access Management in Identity Cloud:

      ```
      $ export amInstanceUrl='myAmInstanceUrl'
      ```

   2. Use a `curl` command similar to the following to retrieve an access token:

      ```
      $ mytoken=$(curl -s \
      --user "oauth2-client:password" \
      --data 'grant_type=password&username=demo&password=Ch4ng3!t&scope=mail' \
      $amInstanceUrl/oauth2/realms/alpha/access_token | jq -r ".access_token")
      ```

   3. Validate the access token returned in the previous step:

```
$ curl -v \
--cacert /path/to/secrets/ig.example.com-certificate.pem \
--header "Authorization: Bearer ${mytoken}" \
https://ig.example.com:8443/oauth2rs-idc

{
  active = true,
  scope = mail,
  realm = /alpha,
  client_id = oauth2-client,
  ...
}
```

Identity Cloud as an OpenID Connect provider

This example sets up ForgeRock Identity Cloud as an OpenID Connect identity provider, and Identity Gateway as a relying party.

For more information about Identity Gateway and OpenID Connect, refer to OpenID Connect.

Before you start, prepare Identity Cloud, IG, and the sample application as described in Example installation for this guide.

1. Set up Identity Cloud:
    1. Log in to the Identity Cloud admin UI as an administrator.
    2. Make sure you are managing the `alpha` realm. If not, click the current realm at the top of the screen, and switch realm.
    3. Go to **Identities** > **Manage** > **Alpha realm - Users**, and add a user with the following values:
        - **Username**: `demo`
        - **First name**: `demo`
        - **Last name**: `user`
        - **Email Address**: `demo@example.com`
        - **Password**: `Ch4ng3!t`
    4. Go to **Applications** > **+ Custom Application** > **OIDC - OpenId Connect** > **Web** and add a web application with the following values:
        - **Name**: `oidc_client`
        - **Owners**: `demo user`
        - **Client Secret**: `password`
        - **Sign On** > **Sign-in URLs**: `https://ig.example.com:8443/home/id_token/callback`
        - **Sign On** > **Grant Types**: `Authorization Code`
        - **Sign On** > **Scopes**: `openid`, `profile`, `email`

- **Show advanced settings** > **Authentication** > **Implied Consent**: `On`

For more information, refer to Identity Cloud's Application management.

2. Set up Identity Gateway:

   1. Set an environment variable for the `oidc_client` password, and then restart IG:

   ```
   $ export OIDC_SECRET_ID='cGFzc3dvcmQ='
   ```

   1. Add the following route to IG to serve the sample application .css and other static resources:

   Linux

   ```
   $HOME/.openig/config/routes/00-static-resources.json
   ```

   Windows

   ```
   %appdata%\OpenIG\config\routes\00-static-resources.json
   ```

   ```
   {
     "name" : "00-static-resources",
     "baseURI" : "http://app.example.com:8081",
     "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
   or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
     "handler": "ReverseProxyHandler"
   }
   ```

   2. Add the following route to Identity Gateway, replacing the value for the property `amInstanceUrl`:

   Linux

   ```
   $HOME/.openig/config/routes/oidc-idc.json
   ```

Windows

```
%appdata%\OpenIG\config\routes\oidc-idc.json
```

```
{
  "name": "oidc-idc",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/id_token')}",
  "properties": {
    "amInstanceUrl": "https://myTenant.forgeblocks.com/am"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AuthenticatedRegistrationHandler-1",
      "type": "Chain",
      "config": {
        "filters": [
          {
            "name": "ClientSecretBasicAuthenticationFilter-1",
            "type": "ClientSecretBasicAuthenticationFilter",
            "config": {
              "clientId": "oidc_client",
              "clientSecretId": "oidc.secret.id",
              "secretsProvider": "SystemAndEnvSecretStore-1"
            }
          }
        ],
        "handler": "ForgeRockClientHandler"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "AuthorizationCodeOAuth2ClientFilter-1",
          "type": "AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/home/id_token",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "status": 500,
                "headers": {
                  "Content-Type": [
                    "text/plain"
                  ]
                },
                "entity": "Error in OAuth 2.0 setup."
              }
            },
            "registrations": [
              {
                "name": "oauth2-client",
                "type": "ClientRegistration",
                "config": {
                  "clientId": "oidc_client",
                  "issuer": {
                    "name": "Issuer",
```

```
                    "type": "Issuer",
                    "config": {
                      "wellKnownEndpoint": "&{amInstanceUrl}/oauth2/realms/alpha/.well-known/openid-
configuration"
                    }
                  },
                  "scopes": [
                    "openid",
                    "profile",
                    "email"
                  ],
                  "authenticatedRegistrationHandler": "AuthenticatedRegistrationHandler-1"
                }
              }
            ],
            "requireHttps": false,
            "cacheExpiration": "disabled"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Compared to `07-openid.json` in AM as a single OpenID Connect provider, where Access Management is running locally, the ClientRegistration `wellKnownEndpoint` points to Identity Cloud.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/id_token.

      The Identity Cloud login page is displayed.

   2. Log in to Identity Cloud as user `demo`, password `Ch4ng3!t`. The home page of the sample application is displayed.

Cross-domain single sign-on

For organizations relying on AM's session and policy services with SSO, consider cross-Domain Single Sign-On (CDSSO) as an alternative to SSO through OpenID Connect.

This example sets up ForgeRock Identity Cloud as an SSO authentication server for requests processed by Identity Gateway. For more information about about Identity Gateway and CDSSO, refer to Authenticate with CDSSO.

Before you start, prepare Identity Cloud, IG, and the sample application as described in Example installation for this guide.

1. Set up Identity Cloud:

   1. Log in to the Identity Cloud admin UI as an administrator.

   2. Make sure you are managing the `alpha` realm. If not, click the current realm at the top of the screen, and switch realm.

   3. Go to **Identities** > **Manage** > **Alpha realm - Users**, and add a user with the following values:

      - **Username**: `demo`

- **First name**: `demo`
- **Last name**: `user`
- **Email Address**: `demo@example.com`
- **Password**: `Ch4ng3!t`

4. Register an IG agent with the following values, as described in Register an IG agent in Identity Cloud:
   - **ID**: `ig_agent`
   - **Password**: `password`
   - **Redirect URLs**: `https://ig.ext.com:8443/home/cdsso/redirect`
5. (Optional) Authenticate the agent to Identity Cloud as described in Authenticate an IG agent to Identity Cloud.

> **Important**
>
> IG agents are automatically authenticated to Identity Cloud by a deprecated authentication module in Identity Cloud. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of Identity Cloud.

6. Add a Validation Service:
   1. In Identity Cloud, select **Native Consoles** > **Access Management**. The AM admin UI is displayed.
   2. Select **Services**, and add a validation service with the following **Valid goto URL Resources**:
      - `https://ig.ext.com:8443/*`
      - `https://ig.ext.com:8443/*?*`

2. Set up Identity Gateway:
   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
   2. Add the following `session` configuration to `admin.json`, to ensure that the browser passes the session cookie in the form-POST to the redirect endpoint (step 6 of Information flow during CDSSO):

```
{
  "connectors": […],
  "session": {
    "cookie": {
      "sameSite": "none",
      "secure": true
    }
  },
  "heap": […]
}
```

This step is required for the following reasons:

- When `sameSite` is `strict` or `lax`, the browser does not send the session cookie, which contains the nonce used in validation. If IG doesn't find the nonce, it assumes that the authentication failed.
- When `secure` is `false`, the browser is likely to reject the session cookie.

  For more information, refer to admin.json.

3. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

4. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

5. Add the following route to Identity Gateway, and correct the value for the property `amInstanceUrl`:

Linux

```
$HOME/.openig/config/routes/cdsso-idc.json
```

Windows

```
%appdata%\OpenIG\config\routes\cdsso-idc.json
```

```
{
  "name": "cdsso-idc",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/cdsso')}",
  "properties": {
    "amInstanceUrl": "https://myTenant.forgeblocks.com/am"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "&{amInstanceUrl}",
        "realm": "/alpha",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "sessionCache": {
          "enabled": false
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "CrossDomainSingleSignOnFilter-1",
          "type": "CrossDomainSingleSignOnFilter",
          "config": {
            "redirectEndpoint": "/home/cdsso/redirect",
            "authCookie": {
              "path": "/home",
              "name": "ig-token-cookie"
            },
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route compared to `cdsso.json` in CDSSO for IG in standalone mode, where Access Management is running locally:

- The AmService `URL` points to Access Management in the Identity Cloud.
- The AmService `realm` points to the realm where you configure your IG agent.

6. Restart IG.

3. Test the setup:

1. In your browser's privacy or incognito mode, go to https://ig.ext.com:8443/home/cdsso.

   The Identity Cloud login page is displayed.

2. Log in to Identity Cloud as user `demo`, password `Ch4ng3!t`.

   Access Management calls `/home/cdsso/redirect`, and includes the CDSSO token. The CrossDomainSingleSignOnFilter passes the request to sample app.

Policy enforcement

The following procedure gives an example of how to request and enforce policy decisions from Identity Cloud.

Enforce a simple policy

Before you start, set up and test the example in Cross-domain single sign-on.

1. Set up Identity Cloud:

1. In the Identity Cloud admin UI, select **Native Consoles** > **Access Management**. The AM admin UI is displayed.
2. Select **Authorization** > **Policy Sets** > **New Policy Set**, and add a policy set with the following values:
   - **Id** : `PEP-CDSSO`
   - **Resource Types** : `URL`
3. In the new policy set, add a policy with the following values:
   - **Name** : `CDSSO`
   - **Resource Type** : `URL`
   - **Resource pattern** : `*://*:*/*`
   - **Resource value** : `http://app.example.com:8081/home/cdsso`

     This policy protects the home page of the sample application.
4. On the **Actions** tab, add an action to allow HTTP `GET`.
5. On the **Subjects** tab, remove any default subject conditions, add a subject condition for all `Authenticated Users`.

2. Set up IG:

    1. Replace `cdsso-idc.json` with the following route, and correct the value for the property amInstanceUrl:

Linux

```
$HOME/.openig/config/routes/pep-cdsso-idc.json
```

Windows

```
%appdata%\OpenIG\config\routes\pep-cdsso-idc.json
```

```
{
  "name": "pep-cdsso-idc",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/cdsso')}",
  "properties": {
    "amInstanceUrl": "https://myTenant.forgeblocks.com/am"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "&{amInstanceUrl}",
        "realm": "/alpha",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "sessionCache": {
          "enabled": false
        }
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "CrossDomainSingleSignOnFilter-1",
          "type": "CrossDomainSingleSignOnFilter",
          "config": {
            "redirectEndpoint": "/home/cdsso/redirect",
            "authCookie": {
              "path": "/home",
              "name": "ig-token-cookie"
            },
            "amService": "AmService-1"
          }
        },
        {
          "name": "PolicyEnforcementFilter-1",
          "type": "PolicyEnforcementFilter",
          "config": {
            "application": "PEP-CDSSO",
            "ssoTokenSubject": "${contexts.cdsso.token}",
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Note the following feature of the route compared to `cdsso-idc.json`:

- The CrossDomainSingleSignOnFilter is followed by a PolicyEnforcementFilter to enforce the policy `PEP-CDSSO`.

3. Test the setup:

   1. Go to https://ig.ext.com:8443/home/cdsso.

      If you have warnings that the site is not secure respond to the warnings to access the site.

      IG redirects you to Identity Cloud for authentication.

   2. Log in to Identity Cloud as user `demo`, password `Ch4ng3!t`.

      Identity Cloud redirects you back to the request URL, and IG requests a policy decision. Identity Cloud returns a policy decision that grants access to the sample application.

Step up authorization for a transaction

Before you start, set up and test the example in Enforce a simple policy.

1. In the Identity Cloud admin UI, select **Scripts** > **Auth Scripts** > **New Script** > **Journey Decision Node** > **Next**, and add a default Journey Decision Node Script script called `TxTestPassword`:

```
/*
  - Data made available by nodes that have already executed are available in the sharedState variable.
  - The script should set outcome to either "true" or "false".
 */

var givenPassword = nodeState.get("password").asString()

if (givenPassword.equals("7890")) {
  outcome = "true"
} else {
  outcome = "false"
}
```

2. Configure a journey:

   1. Click **Journeys** and add a journey with the following configuration:

      - **Name**: `Tx01_Tree`
      - **Identity Object**: `Alpha realm users`

        The journey canvas is displayed.

   2. In **Nodes** > **Basic Authentication**, drag a **Password Collector** node onto the canvas.

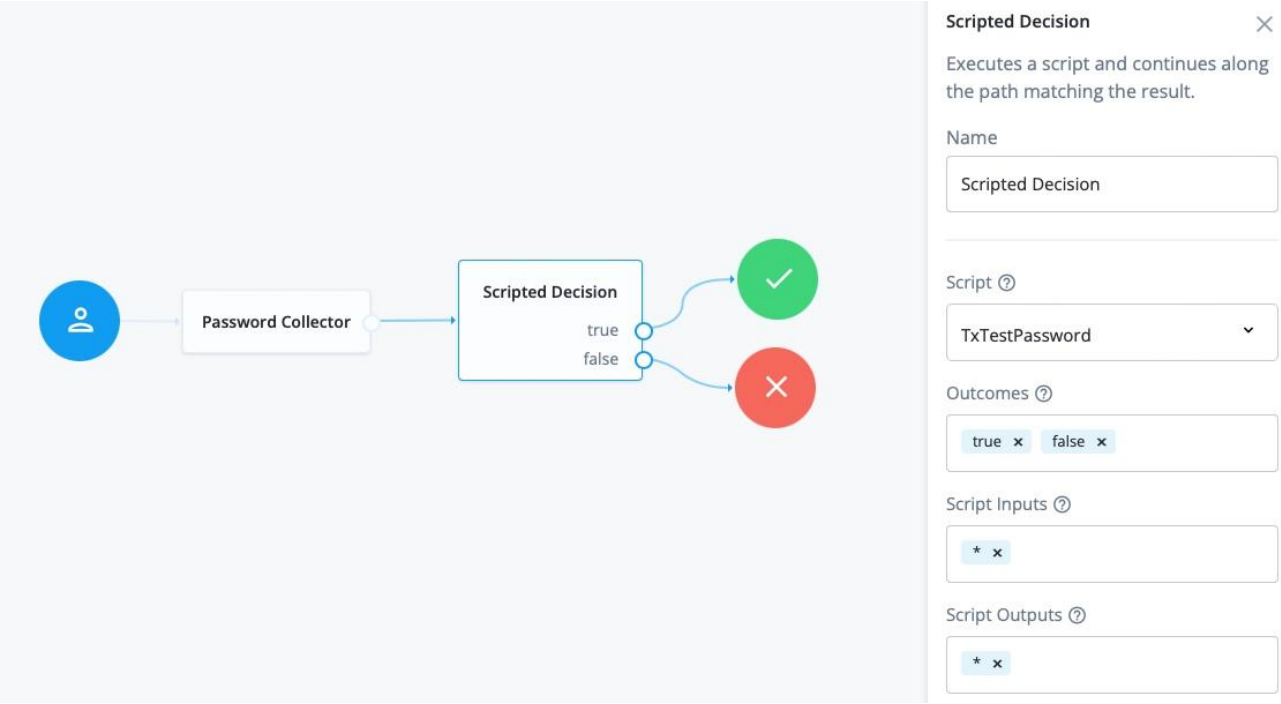
   3. In **Nodes** > **Utilities**, drag a **Scripted decision** node onto the canvas.

   4. Configure the scripted decision node as follows:

      - **Script**: select `TxTestPassword`

- **Outcomes**: enter `true` and `false`

5. Connect the nodes as shown:



For information about configuring trees, refer to ForgeRock Identity Cloud Docs.

3. Edit the authorization policy:

1. In the Identity Cloud admin UI, select **Native Consoles** > **Access Management**. The AM admin UI is displayed.
2. Select **Authorization** > **Policy Sets** > **PEP-CDSSO**, and add the following environment condition to the `CDSSO` policy:

- `All of`
- **Type**: `Transaction`
- **Script name**: `Authenticate to tree`
- **Strategy Specifier**: `Tx01_Tree`

4. Test the setup:

1. In a browser, go to https://ig.ext.com:8443/home/cdsso.

   If you have not previously authenticated to Identity Cloud, the CrossDomainSingleSignOnFilter redirects the request to Identity Cloud for authentication.

2. Log in to Identity Cloud as user `demo`, password `Ch4ng3!t`.
3. Enter the password `7890` required by the script `TxTestPassword`.

   Identity Cloud redirects you back to the request URL, and IG requests a policy decision. Identity Cloud returns a policy decision based on the authentication journey.

Pass runtime data downstream in a JWT

This example sets up Identity Cloud as an identity provider, to pass identity or other runtime information downstream, in a JWT signed with a PEM.

For more information about using runtime data, refer to Passing data along the chain. To help with development, the sample application includes a `/jwt` endpoint to display the JWT, verify its signature, and decrypt it.

Before you start, prepare Identity Cloud, IG, and the sample application as described in Example installation for this guide.

1. Set up secrets:

    1. Locate a directory for secrets, and go to it:

    ```
    $ cd /path/to/secrets
    ```

    2. Create the following secret key and certificate pair as PEM files:

    ```
    $ openssl req \
    -newkey rsa:2048 \
    -new \
    -nodes \
    -x509 \
    -days 3650 \
    -subj "/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
    -keyout ig.example.com-key.pem \
    -out ig.example.com-certificate.pem
    ```

    Two PEM files are created, one for the secret key, and another for the associated certificate.

    3. Map the key and certificate to the same secret ID in IG:

    ```
    $ cat ig.example.com-key.pem ig.example.com-certificate.pem > key.manager.secret.id.pem
    ```

    4. Generate PEM files to sign and verify the JWT:

    ```
    $ openssl req \
    -newkey rsa:2048 \
    -new \
    -nodes \
    -x509 \
    -days 3650 \
    -subj "/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
    -keyout id.key.for.signing.jwt.pem \
    -out id.key.for.verifying.jwt.pem
    ```

5. Make sure the following files have been added to your secrets directory:
   - `id.key.for.signing.jwt.pem`
   - `id.key.for.verifying.jwt.pem`
   - `key.manager.secret.id.pem`
   - `ig.example.com-certificate.pem`
   - `ig.example.com-key.pem`

2. Set up Identity Cloud:
   1. Log in to the Identity Cloud admin UI as an administrator.
   2. Go to **Identities** > **Manage** > **Alpha realm - Users**, and add a user with the following values:
      - **Username**: `demo`
      - **First name**: `demo`
      - **Last name**: `user`
      - **Email Address**: `demo@example.com`
      - **Password**: `Ch4ng3!t`
   3. Register an IG agent with the following values, as described in Register an IG agent in Identity Cloud:
      - **ID**: `ig_agent_jwt`
      - **Password**: `password`
      - **Redirect URLs**: `https://ig.example.com:8443/jwt/redirect`
   4. (Optional) Authenticate the agent to Identity Cloud as described in Authenticate an IG agent to Identity Cloud.

      Important

      IG agents are automatically authenticated to Identity Cloud by a deprecated authentication module in Identity Cloud. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of Identity Cloud.

   5. Add a Validation Service:
      1. In Identity Cloud, select **Native Consoles** > **Access Management**. The AM admin UI is displayed.
      2. Select **Services**, and add a validation service with the following **Valid goto URL Resources**:
         - `https://ig.example.com:8443/*`
         - `https://ig.example.com:8443/*?*`

3. Set up IG:
   1. Set up TLS by adding the following file to IG, replacing the value for the property `secretsDir`:

Linux

```
$HOME/.openig/config/admin.json
```

Windows

```
%appdata%\OpenIG\config\admin.json
```

```
{
  "mode": "DEVELOPMENT",
  "properties": {
    "secretsDir": "/path/to/secrets"
  },
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443,
      "tls": "ServerTlsOptions-1"
    }
  ],
  "session": {
    "cookie": {
      "sameSite": "none",
      "secure": true
    }
  },
  "heap": [
    {
      "name": "ServerTlsOptions-1",
      "type": "ServerTlsOptions",
      "config": {
        "keyManager": {
          "type": "SecretsKeyManager",
          "config": {
            "signingSecretId": "key.manager.secret.id",
            "secretsProvider": "ServerIdentityStore"
          }
        }
      }
    },
    {
      "name": "ServerIdentityStore",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "&{secretsDir}",
        "suffix": ".pem",
        "mappings": [{
          "secretId": "key.manager.secret.id",
          "format": {
            "type": "PemPropertyFormat"
          }
        }]
      }
    }
  ]
}
```

2. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.

3. Add the following route to IG to serve the sample application .css and other static resources:

Linux

```
$HOME/.openig/config/routes/00-static-resources.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-static-resources.json
```

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$')
or matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

4. Add the following route to IG, replacing the value for the properties `secretsDir` and `amInstanceUrl`:

Linux

```
$HOME/.openig/config/routes/jwt-idc.json
```

Windows

```
%appdata%\OpenIG\config\routes\jwt-idc.json
```

```
{
  "name": "jwt-idc",
  "condition": "${find(request.uri.path, '/jwt')}",
  "baseURI": "http://app.example.com:8081",
  "properties": {
    "secretsDir": "/path/to/secrets",
    "amInstanceUrl": "https://myTenant.forgeblocks.com/am"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "&{amInstanceUrl}",
        "realm": "/alpha",
        "agent": {
          "username": "ig_agent_jwt",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "sessionCache": {
          "enabled": false
        }
      }
    },
    {
      "name": "pemPropertyFormat",
      "type": "PemPropertyFormat"
    },
    {
      "name": "FileSystemSecretStore-1",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "PLAIN",
        "directory": "&{secretsDir}",
        "suffix": ".pem",
        "mappings": [{
          "secretId": "id.key.for.signing.jwt",
          "format": "pemPropertyFormat"
        }]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "CrossDomainSingleSignOnFilter-1",
          "type": "CrossDomainSingleSignOnFilter",
          "config": {
            "redirectEndpoint": "/jwt/redirect",
            "authCookie": {
              "path": "/jwt",
              "name": "ig-token-cookie"
            },
```

```
            "amService": "AmService-1"
          }
        },
        {
          "name": "UserProfileFilter",
          "type": "UserProfileFilter",
          "config": {
            "username": "${contexts.ssoToken.info.uid}",
            "userProfileService": {
              "type": "UserProfileService",
              "config": {
                "amService": "AmService-1"
              }
            }
          }
        },
        {
          "name": "JwtBuilderFilter-1",
          "type": "JwtBuilderFilter",
          "config": {
            "template": {
              "name": "${contexts.userProfile.commonName}",
              "email": "${contexts.userProfile.rawInfo.mail[0]}"
            },
            "secretsProvider": "FileSystemSecretStore-1",
            "signature": {
              "secretId": "id.key.for.signing.jwt",
              "algorithm": "RS512"
            }
          }
        },
        {
          "name": "HeaderFilter-1",
          "type": "HeaderFilter",
          "config": {
            "messageType": "REQUEST",
            "add": {
              "x-openig-user": ["${contexts.jwtBuilder.value}"]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

4. Test the setup:

    1. Go to https://ig.example.com:8443/jwt.

       If you receive warnings that the site is not secure, respond to the warnings to access the site. The Identity Cloud login page is displayed.

    2. Log in to Identity Cloud as user `demo`, password `Ch4ng3!t`. The sample app displays the signed JWT along with its header and payload.

    3. In `USE PEM FILE`, enter the absolute path to `id.key.for.verifying.jwt.pem` to verify the JWT signature.

Secure the OAuth 2.0 access token endpoint

This section uses a GrantSwapJwtAssertionOAuth2ClientFilter to transform requests for OAuth 2.0 access tokens into secure JWT bearer grant type requests. It propagates the transformed requests to Identity Cloud to obtain an access token.

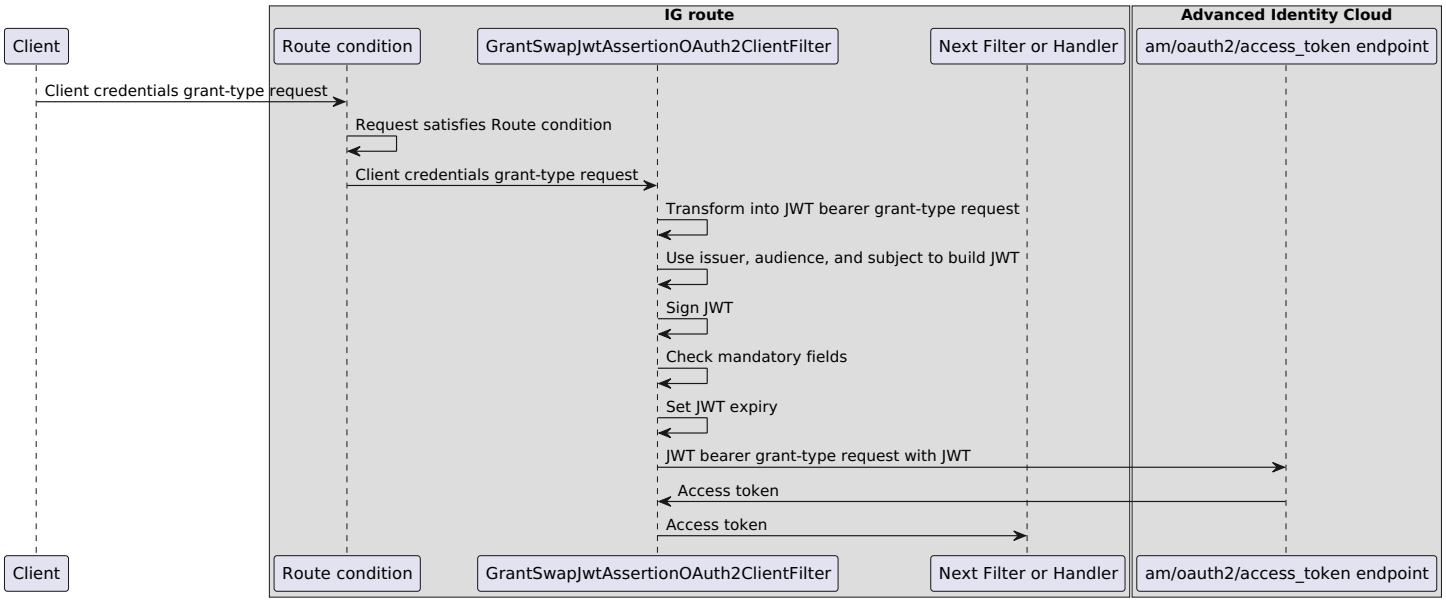
Use GrantSwapJwtAssertionOAuth2ClientFilter to increase the security of less-secure grant-type requests, such as Client credentials grant requests or Resource owner password credentials grant requests.

Caution

The GrantSwapJwtAssertionOAuth2ClientFilter obtains access tokens from the `/oauth2/access_token` endpoint. To prevent unwanted or malicious access to the endpoint, make sure only a well-defined set of clients can use this filter. Consider the following options to secure access to the GrantSwapJwtAssertionOAuth2ClientFilter:

- Deploy IG on a trusted network.
- Use mutual TLS (mTLS) and X.509 certificates for authentication between clients and IG. For more information, refer to OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens.
- Configure an AllowOnlyFilter in front of the GrantSwapJwtAssertionOAuth2ClientFilter to control access within a route.
- Define restrictive Route conditions to allow access only for expected grant-type requests. For example, define a route condition that requires a specific client ID, grant-type, or scope.
- Configure a ScriptableFilter in front of the GrantSwapJwtAssertionOAuth2ClientFilter to validate requests.

The following figure shows the flow of information for a grant swap:



Before you start, prepare Identity Cloud, IG, and the sample application as described in Example installation for this guide.

1. Set up Identity Cloud:

    1. Log in to the Identity Cloud admin UI as an administrator.

2. Create a service account with the following values, as described in Create a new service account:

   - **Name**: `myServiceAccount`
   - **Scopes**: `fr:idm:* All Identity Management APIs`

     The service account ID is displayed and you are prompted to download the private key. The public key is held in Identity Cloud.

     For more information, refer to Service accounts.

3. Make a note of the service account ID and download the private key to your secrets directory.
4. Rename the key to match the regex format `(\.[a-zA-Z0-9])*`. For example, rename `myServiceAccount_privateKey.jwk` to `privateKey.jwk`.

2. Set up IG:

   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
   2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/grant-swap.json
```

Windows

```
%appdata%\OpenIG\config\routes\grant-swap.json
```

```
{
  "name" : "grant-swap",
  "properties": {
    "idcInstanceUrl": "https://myTenant.forgeblocks.com",
    "issuer": "service-account-id",
    "secretsDir": "path-to-secrets",
    "privateKeyFilename": "privateKey.jwk"
  },
  "condition" : "#{find(request.uri.path, '^/am/oauth2/access_token') &&
request.entity.form['grant_type'][0] == 'client_credentials'}",
  "baseURI" : "&{idcInstanceUrl}:443/",
  "heap" : [ {
    "name": "JwkPropertyFormat-01",
    "type": "JwkPropertyFormat"
  },
    {
      "name": "FileSystemSecretStore-01",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "JwkPropertyFormat-01",
        "directory": "&{secretsDir}",
        "mappings": [ {
          "secretId": "&{privateKeyFilename}",
          "format": "JwkPropertyFormat-01"
        }
        ]
      }
    }
  ],
  "handler" : {
    "type" : "Chain",
    "capture" : "all",
    "config" : {
      "filters" : [
        {
          "name" : "GrantSwapJwtAssertionOAuth2ClientFilter-01",
          "description": "access /access_token endpoint with jwt-bearer-profile",
          "type" : "GrantSwapJwtAssertionOAuth2ClientFilter",
          "capture" : "all",
          "config" : {
            "clientId" : "service-account",
            "assertion" : {
              "issuer" : "&{issuer}",
              "audience" : "&{idcInstanceUrl}/am/oauth2/access_token",
              "subject" : "&{issuer}",
              "expiryTime": "2 minutes"
            },
            "signature": {
              "secretId": "&{privateKeyFilename}",
              "includeKeyId": false
            },
            "secretsProvider": "FileSystemSecretStore-01",
            "scopes" : {
              "type": "RequestFormResourceAccess"
            }
          }
        }
```

```
        ],
        "handler" : "ForgeRockClientHandler"
      }
    }
  }
```

3. In the route, replace the values for the following properties with your values:
   - `idcInstanceUrl`: The root URL of your Identity Cloud.
   - `issuer`: The ID of the service account created in Identity Cloud
   - `secretsDir`: The directory containing the downloaded private key
   - `privateKeyFilename`: The filename of the downloaded private key
4. Notice the following features of the route:
   - The condition intercepts only `client_credentials` grant-type requests on the path `/am/oauth2/access_token`. A more secure condition can be set on the client ID.
   - Requests are rebased to the Identity Cloud URL.
   - A FileSystemSecretStore loads the private-key JWK used to sign the JWT.
   - The GrantSwapJwtAssertionOAuth2ClientFilter:
     - Requires the core JWT claims `issuer`, `subject`, `audience`, and `expiryTime`.
     - Uses RequestFormResourceAccess to extract scopes from the inbound request for inclusion in the JWT-assertion grant-type request propagated to AM.
     - Signs the JWT with the JWK provided by the service account.
   - The GrantSwapJwtAssertionOAuth2ClientFilter `clientId` refers to the OAuth 2.0 client ID created by AM. The value must be `service-account`.
5. Add the following route to IG to return a standard OAuth 2.0 error response if the request fails the route condition:

Linux

```
$HOME/.openig/config/routes/zz-returns-invalid-request.json
```

Windows

```
%appdata%\OpenIG\config\routes\zz-returns-invalid-request.json
```

```
{
  "name" : "zz-returns-invalid-request",
  "handler" : {
    "type" : "StaticResponseHandler",
    "capture" : "all",
    "config" : {
      "status": 400,
      "headers": {"Content-Type": ["application/json; charset=UTF-8"]},
      "entity": "{\"error\": \"Invalid_request\", \"error_description\": \"Invalid request\"}"
    }
  }
}
```

3. Test the setup by accessing the route with a `curl` command similar to this:

```
$ curl  \
    --cacert /path/to/secrets/ig.example.com-certificate.pem \
    --location \
    --request POST 'https://ig.example.com:8443/am/oauth2/access_token' \
    --header 'Content-Type: application/x-www-form-urlencoded' \
    --data-urlencode 'client_id=myServiceAccount' \
    --data-urlencode 'grant_type=client_credentials' \
    --data-urlencode 'scope=fr:idm:*'

{"access_token":"eyJ...","scope":"fr:idm:*","token_type":"Bearer","expires_in":899}
```

The command makes a `client_credentials` grant-type request on the path `/am/oauth2/access_token`, supplying the client ID and scopes. IG transforms the request into a JWT-assertion grant-type request and propagates it to Identity Cloud.

Because the service account in Identity Cloud supports the requested scope, the GrantSwapJwtAssertionOAuth2ClientFilter returns an access token.

IG Studio

This guide gives an overview of how to use IG Studio to design and develop routes to protect applications.

Start with Studio

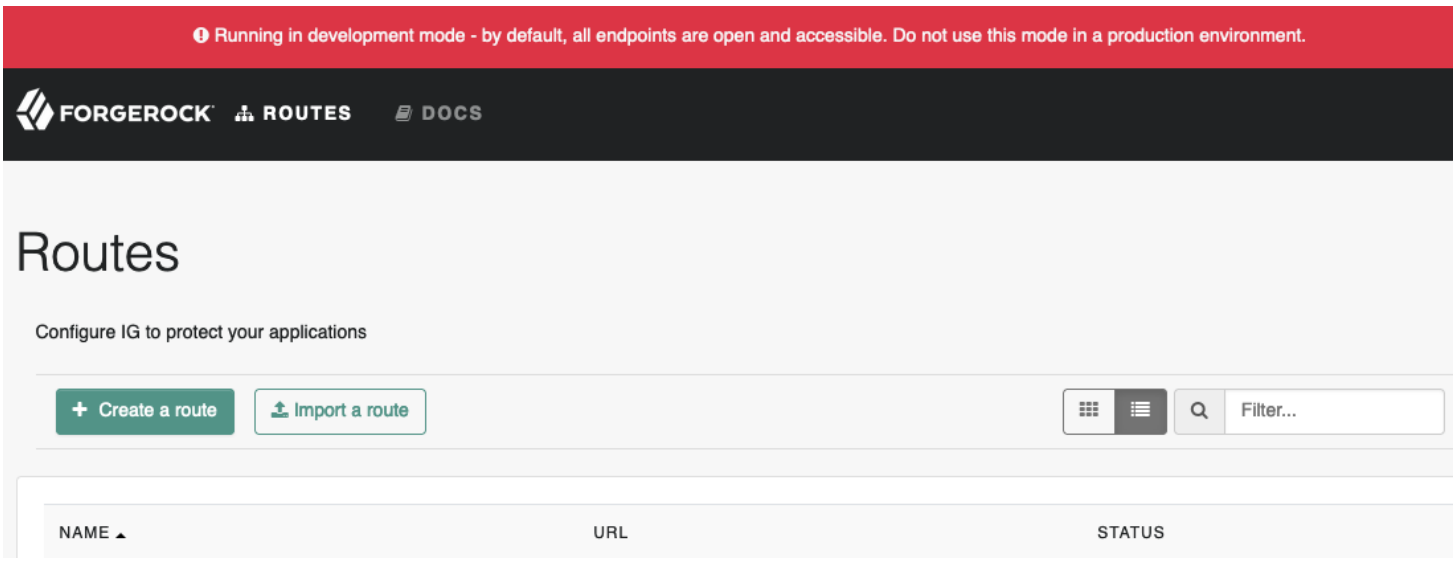
IG Studio is a user interface to help you build and deploy your IG configuration. There are two ways to create routes in Studio:

- With the *structured editor* (deprecated) to build simple routes by using predefined menus and templates. The structured editor presents valid options and default values as you add filters, decorators, and other objects to your configuration.
- With the *FreeForm Designer* to design complex, multi-branched routes. Drag handlers and filters from a side bar onto the canvas to begin designing the route. The FreeForm Designer helps you to visualize the chain, and track the path of requests, responses, and contexts.

After installation, IG is by default in production mode. The `/routes` endpoint is not exposed or accessible, and Studio is effectively disabled. To access Studio, switch to development mode as described in Operating modes.

If you provide a custom `config.json`, include a main router named `_router`. If a custom `config.json` is not provided, IG includes this router by default.

When IG is installed and running in development mode, as described in Quick install, access Studio on http://ig.example.com:8080/openig/studio. The **Routes** screen is displayed:



During IG upgrade, routes that were previously created in Studio are automatically transferred to the new version of IG. Where possible, IG replaces deprecated settings with the newer evolved setting. If IG needs additional information to upgrade the route, the route status becomes ⚠ **Compatibility update required**. Select the route, and provide the requested information.

> **Important**
>
> SecretsProviders can't be configured in Studio. Documentation examples generated with Studio might refer to SecretsProviders that must be configured separately in `config.json`.

Upgrade from an earlier version of Studio

From IG 2024.3, Studio manages the migration of deprecated objects for routes created and managed in earlier versions of Studio.

IG doesn't manage migration for the following:

- Routes in Studio editor mode
- Custom filters in routes

Migration for routes containing secrets

IG automatically:

- Updates the route to use SecretsProvider and SecretIds
- Removes references to the password

You must manually create the required SecretsProvider in `config.json` and create its referenced secrets.

Migration for routes containing Splunk or ElasticSearch audit event handlers

IG automatically deletes the Splunk or ElasticSearch audit event handlers from the route.

Create and edit routes with Structured Editor (deprecated)

This section describes basic tasks for creating and deploying routes in the structured editor of Studio.

> **Important**
>
> The structured editor of Studio is deprecated. For more information, refer to the Deprecated section of the *Release Notes*.

Creating simple routes

1. In IG Studio, create a route:
    1. Go to http://ig.example.com:8080/openig/studio, and select + **Create a route**.
    2. Select **Structured** to use the structured editor.
2. Enter the URL of the application you want to protect, followed by a path condition to access the route. For example, enter `http://app.example.com:8081/my-basic-route`.

   The route is created, and menus to add configuration objects to the route are displayed.
3. On the top-right of the screen, select and **Display** to review the route.

   A route similar to this is displayed, where the path condition is used for the route name:

```
{
  "name": "my-basic-route",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/my-basic-route')}",
  "handler": "ReverseProxyHandler"
}
```

Change the basic settings of a route

1. In Studio, select **ROUTES**, and then select a route with the icon.
2. On the top-right of the screen select **Route settings**.
3. Using the on-screen hints for guidance, change the name, condition, or other features of the route, and save the changes.
4. On the top-right of the screen, select and **Display** to review the route.

Adding configuration to a route

After creating a route in the structured editor, you can add filters, decorators, scripts, and other configuration to the route.

Add other configuration to a route

1. In Studio, select **ROUTES**, and then select a route with the icon.
2. Select one of the configuration options, and follow the on-screen hints to select configuration settings.

   For routes to test with the examples in the Gateway guide, refer to Example routes created with Structured Editor (deprecated).

Add other filters to a route

Use this procedure to add any filter type to the configuration.

1. In Studio, select **ROUTES**, and then select a route with the icon.
2. Select **Other filters** > **New filter** > **Other filter**.
3. In **Create filter**, select a filter type from the list, enter a name, and optionally enter a configuration for the filter.

   > **Note**
   >
   > Studio checks that the JSON is valid, but doesn't check that the configuration of the filter is valid. If the filter configuration isn't valid, when you deploy the route it fails to load.

   When you save, the filter is added to the list of other filters but is not added to the configuration.
4. Enable the filter to add it to the configuration.

   If you disable the filter again, it is removed from the route's chain but the configuration is saved. Simply enable the filter again to add it back in the chain.

Managing the route chain

The **Chain** view lists the filters in the order that they appear in the configuration.

Some filters have a natural position in the chain. For example, so that an authenticated user is given the correct permissions, an authentication filter must come before an authorization filter. Similarly, so that an authorization token is transformed, an authorization filter always comes before a token transformation filter.

Other filters have a flexible position in the chain. For example, an AssignmentFilter can be used before a request is handled or after a response is handled.

When the position of a filter is fixed, it is automatically placed in the correct position in the chain; you cannot change the position. When the position of a filter is flexible, the icon is displayed, and you can drag the filter into a different position in the chain.

Select **Chain** to view and manage the filters in the chain as follows:

- When the icon is displayed, drag a filter up or down the chain.
- select to edit a filter.
- Select **Realm Settings** to disable and remove a filter from the chain.

For information about chains, refer to Chain.

Deploy and undeploy routes

Deploy a Route

1. In Studio, select **ROUTES**, and then select a route created with the structured editor (with the icon).
2. On the top-right of the screen, select and **Display** to review the route.
3. If the route is okay, select **Deploy** to push the route to the IG configuration.

   > **Important**
   >
   > If the route configuration is not valid, or if a service that the route relies on, such as an AM service, is not available, the route fails to deploy.

   If the route deploys successfully, **Deployed** is displayed, and the **Deploy** button is greyed out.
4. Check the `$HOME/.openig/config/routes` folder in your IG configuration to see that the route is there.

   By default, routes are loaded automatically into the IG configuration. You don't need to stop and restart IG. For more information, refer to Prevent the reload of routes.
5. Check the system log to confirm that the route was loaded successfully into the configuration. For information about logs, refer to Manage logs.

Undeploy a Route

1. In Studio, select **ROUTES** and then select a route with the status **Deployed**.
2. On the top-right of the screen, select and **Undeploy**, and then confirm your request.

The route is removed from the IG configuration. On the Studio screen, the route status **Deployed** is no longer displayed, and the **Deploy** option is active again.

Create and edit routes with Freeform Designer

The following sections describe how to create a simple route in the FreeForm Designer of Studio, and then add configuration to the route. For examples of routes created with the FreeForm Designer that can be tested with the examples in the Gateway guide, refer to Example routes created with Freeform Designer.

Create a simple route

1. In IG Studio, create a route:
    1. Go to `http://ig.example.com:8080/openig/studio`, and then select **Create a route**.
    2. Select **Freeform** to use the FreeForm Designer.
2. Select **Basic** to create a route from a blank template.
3. Enter a URL for the application you want to protect, followed by a path condition to access the route. For example, enter `http://app.example.com:8081/my-basic-route`.

    The route is displayed on the **Flow** tab of the canvas.
4. On the top-right of the screen, select and **Display** to review the route.

```
{
  "name": "my-basic-route",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/my-basic-route')}",
  "handler": "ReverseProxyHandler",
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler"
    },
    {
      "type": "BaseUriDecorator",
      "name": "baseUri"
    },
    {
      "type": "TimerDecorator",
      "name": "timer",
      "config": {
        "timeUnit": "ms"
      }
    },
    {
      "type": "CaptureDecorator",
      "name": "capture",
      "config": {
        "captureEntity": false,
        "captureContext": false,
        "maxEntityLength": 524288
      }
    }
  ]
}
```

Change the basic settings of a route

1. Using the route created in Create a simple route, on the top-right of the screen select **Route settings**.
2. Using the on-screen hints for guidance, change the name, condition, or other features of the route, and save the changes.
3. On the top-right of the screen, select ⋮ and **Display** to review the route.

Add objects to a route heap

1. Using the route created in Create a simple route, select **All Objects** > **Create Object**.
2. In **Node Type**, select an object type from the drop down list. For example, create an AmService object, using the following values:
   - **Name**: `AmService-1`
   - **URI**: `http://am.example.com:8088/openam/`
   - **Agent**:
     - **Agent**: `ig-agent`

- **Password**: `password`

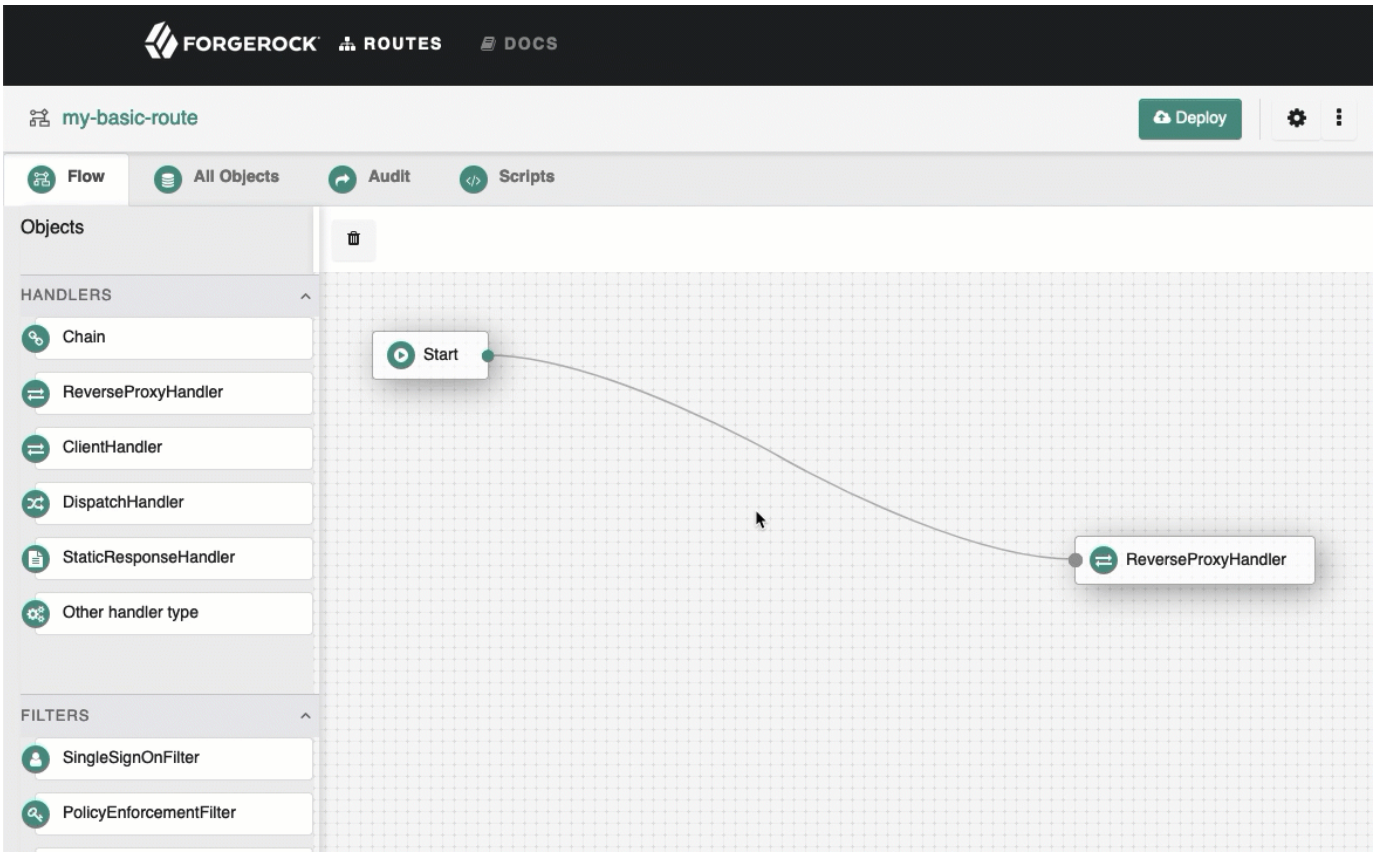
> **Important**
>
> Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.

When you save, the object is added to route heap but is not used in the route.

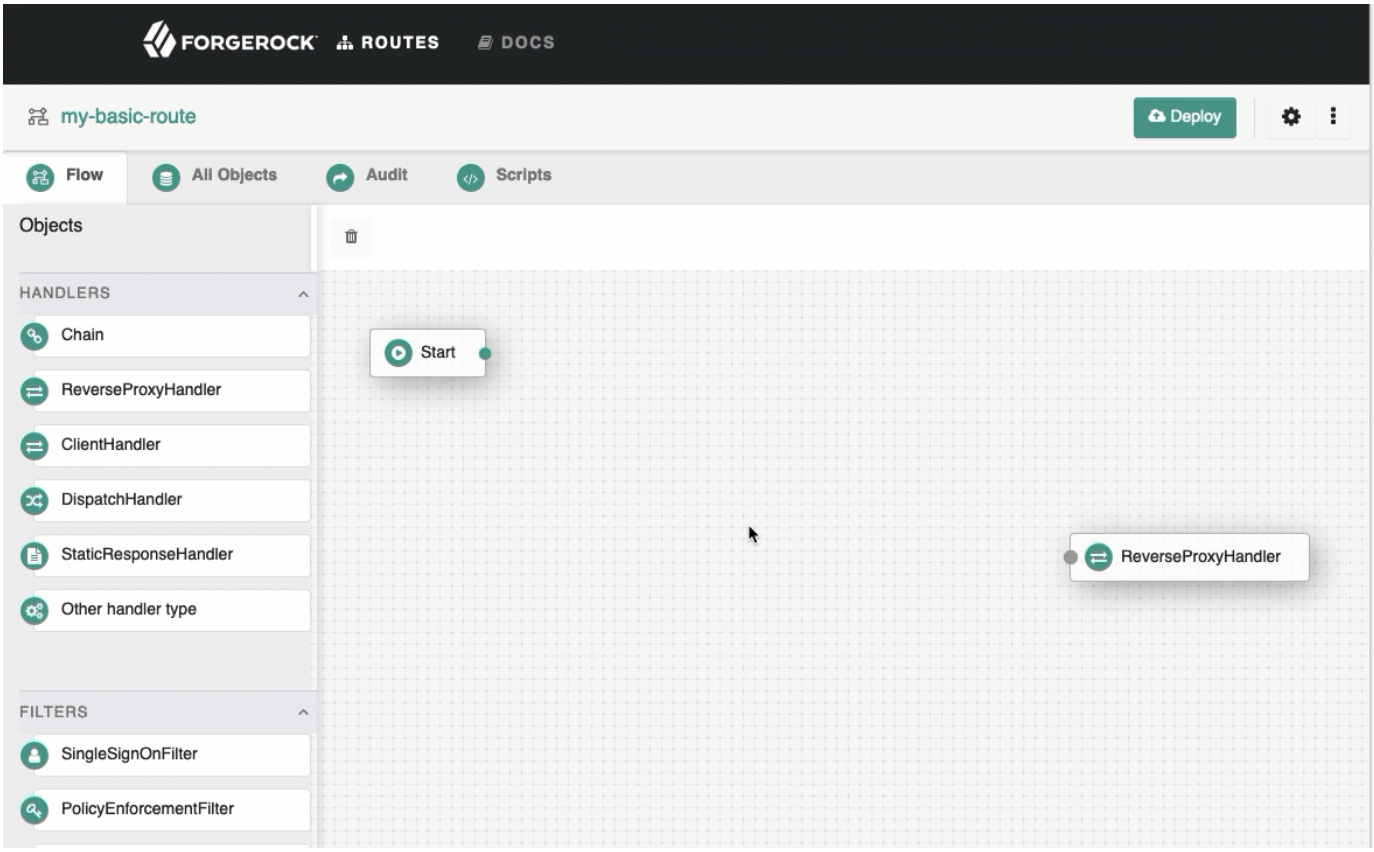
3. On the top-right of the screen, select ⋮ and **Display** to review the route.

Add configuration to a route

1. Using the route created in Create a simple route, select the **Flow** tab, and delete the connector between **Start** and **ReverseProxyHandler**.

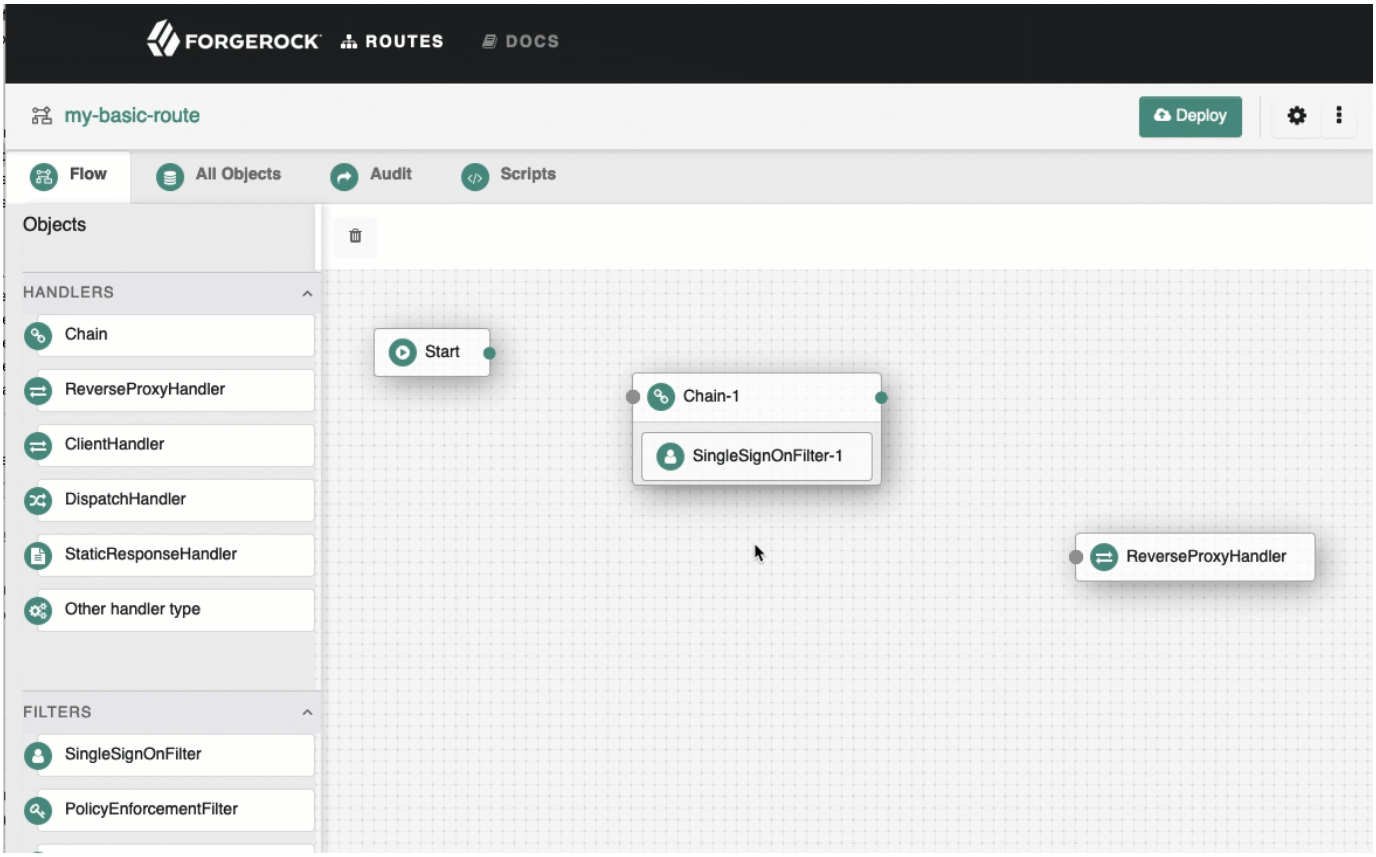


2. Drag a **Chain** from the side bar onto the canvas, and then drag a SingleSignOnFilter into the chain.



3. In the menu for the SingleSignOnFilter, enter the name of the AmService object you created in Add objects to a route heap, `AmService-1`. The filter uses the object previously defined in the heap.

4. Connect **Start** to **Chain-1**, and **Chain-1** to **ReverseProxyHandler**.



5. On the top-right of the screen, select ⋮ and **Display** to review the route.

Decorate objects in the route

1. Using the route created in Create a simple route, select the **All Objects** tab.

   A list of objects in the route is displayed. By default, all available decorators are included in the route heap, but they do not decorate any objects.

2. For the ReverseProxyHandler or filter, select ✎, select the **Decorations** tab, and then enable one or more of the decorators.

3. On the top-right of the screen, select ⋮ and **Display** to review the route.

Edit and import routes

The following sections describe basic tasks for Edit and import routes in Studio:

Edit routes in editor mode

After creating a route in Studio, you can edit it by using the options offered in Studio, or by switching to editor mode and using the JSON editor.

Routes created only in the menus of structured editor have the icon ☰. Routes created only in the menus of FreeForm Designer have the icon 🖧. Imported routes and routes edited in editor mode have the icon **{ }**.

Important

When you go into editor mode, you can use the JSON editor to manually edit the route, but can no longer use the full Studio interface to add to or edit the configuration.

1. In Studio, select **ROUTES**, and then select a route created with the structured editor (with the icon).
2. Edit the route in Studio or manually:
   - To edit in Studio, select options offered in Studio.
   - To edit manually, select and **Editor mode**, and use the JSON editor to edit the route.

   If the route status is **Deployed**, it changes to **Changes pending**.
3. Deploy the route as described in Deploying and undeploying routes.

Import routes into Studio

When you import a route into Studio, it is imported in editor mode. You can use the JSON editor to manually edit the route, but can't use the full Studio interface to add or edit filters.

Routes created only in the menus of structured editor have the icon . Routes created only in the menus of FreeForm Designer have the icon . Imported routes and routes edited in editor mode have the icon **{ }**.

1. In Studio, select **ROUTES** and then **Import a route**.
2. Click in the window to import a route, or drag a route from your filesystem.

   If the route has a `name` property, the name is automatically used for the **Name** and **ID** fields in Studio.
3. If necessary, make the following changes, and then select **Import**:
   - If the **Name** and **ID** fields are empty, enter a unique name and ID for the route.
   - If the **Name** and **ID** fields are outlined in red, the route name or ID already exists in Studio. Change the name and ID to be unique.
   - If an error message is displayed, the route is not valid JSON. Fix the route and then try again to import it.

   The route is added to the list of routes on the **ROUTES** page.
4. Deploy the route as described in Deploying and undeploying routes.

View and search for routes in your configuration

All of the routes that exist in your backend configuration are displayed on the **ROUTES** page, including imported routes and routes created outside of Studio.

To search for a route, select **ROUTES**, and type part of the route name in the search box. Matching routes are displayed as you enter the search criteria.

Restrict access to Studio

When IG is running in development mode, by default the Studio endpoint is open and accessible. To allow only specific users to access Studio, configure a StudioProtectionFilter with a SingleSignOnFilter or CrossDomainSingleSignOnFilter.

The following example uses a SingleSignOnFilter to require users to authenticate with AM before they can access Studio, and protects the request from Cross Site Request Forgery (CSRF) attacks.

1. Set up AM:
    1. Select **Services** > **Add a Service** and add a **Validation Service** with the following **Valid goto URL Resources**:
        - `https://ig.example.com:8443/*`
        - `https://ig.example.com:8443/*?*`
    2. Register an IG agent with the following values, as described in Register an IG agent in AM:
        - **Agent ID**: `ig_agent`
        - **Password**: `password`

        > **Important**
        >
        > Use secure passwords in a production environment. Consider using a password manager to generate secure passwords.
    3. (Optional) Authenticate the agent to AM as described in Authenticate an IG agent to AM.

        > **Important**
        >
        > IG agents are automatically authenticated to AM by a deprecated authentication module in AM. This step is currently optional, but will be required when authentication chains and modules are removed in a future release of AM.
2. Set up IG:
    1. Set an environment variable for the IG agent password, and then restart IG:

        ```
        $ export AGENT_SECRET_ID='cGFzc3dvcmQ='
        ```

        The password is retrieved by a SystemAndEnvSecretStore, and must be base64-encoded.
    2. Add the following `admin.json` configuration to IG:

```
{
  "prefix": "openig",
  "mode": "DEVELOPMENT",
  "properties": {
    "SsoTokenCookieOrHeader": "iPlanetDirectoryPro"
  },
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443
    }
  ],
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent" : {
          "username" : "ig_agent",
          "passwordSecretId" : "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/",
        "ssoTokenHeader": "&{SsoTokenCookieOrHeader}"
      }
    },
    {
      "name": "StudioProtectionFilter",
      "type": "ChainOfFilters",
      "config": {
        "filters": [
          {
            "type": "SingleSignOnFilter",
            "config": {
              "amService": "AmService-1"
            }
          },
          {
            "type": "CsrfFilter",
            "config": {
              "cookieName": "&{SsoTokenCookieOrHeader}",
              "failureHandler": {
                "type": "StaticResponseHandler",
                "config": {
                  "status": 403,
                  "headers": {
                    "Content-Type": [
                      "text/plain"
                    ]
                  },
                  "entity": "Request forbidden"
                }
              }
            }
```

```
            }
          ]
        }
      }
    ]
  }
```

Notice the following features of the configuration:

- The `prefix` sets the base of the administrative route to the default value `/openig`. The Studio endpoint is therefore `/openig/studio`.
- The `mode` is `development`, so by default the Studio endpoint is open and unfiltered.
- The `properties` object sets a configuration parameter for the value of the SSO token cookie or header, which is used in AmService and CorsFilter.
- The AmService uses the IG agent in AM for authentication.

  The agent password for AmService is provided by a SystemAndEnvSecretStore in the heap.
- The StudioProtectionFilter calls the SingleSignOnFilter to redirect unauthenticated requests to AM, and uses the CsrfFilter to protect requests from CSRF attacks. For more information, refer to SingleSignOnFilter and CsrfFilter.

3. Restart IG to take into account the changes to `admin.json`.

3. Test the setup:

   1. If you are logged in to AM, log out and clear any cookies.
   2. Go to http://ig.example.com:8080/openig/studio. The SingleSignOnFilter redirects the request to AM for authentication.
   3. Log in to AM with user `demo`, password `Ch4ng31t`. The Studio **Routes** screen is displayed.

Example routes created with Structured Editor (deprecated)

The following sections give examples of how to set up some of the routes used in the Gateway guide by using the structured editor of Studio.

Important

The structured editor of Studio is deprecated. For more information, refer to the Deprecated section of the *Release Notes*.

Important

SecretsProviders can't be configured in Studio. Documentation examples generated with Studio might refer to SecretsProviders that must be configured separately in `config.json`.

Single sign-on in Structured Editor

This section describes how to set up SSO in the structured editor of Studio. For more information about setting up SSO, refer to Authentication.

1. In IG Studio, create a route:
    1. Go to `http://ig.example.com:8080/openig/studio`, and then select + **Create a route**.
    2. Select ☰ **Structured** to use the structured editor.
2. Select **Advanced options** on the right, and create a route with the following options:
    - **Base URI**: `http://app.example.com:8081`
    - **Condition: Path**: `/home/sso-studio`
    - **Name** : `sso-studio`
3. Configure authentication:
    1. Select **Authentication**.
    2. Select **Single Sign-On**, and enter the following information:
        - **AM service** : Configure an AM service to use for authentication:
            - **URI:** `http://am.example.com:8088/openam`
            - **Secrets Provider:** `SystemAndEnvSecretStore-1`
            - **Agent** :
                - **Username** : `ig_agent`
                - **Password Secret ID** : `password.secret.id`

    Leave all other values as default.
4. On the top-right of the screen, select ⋮ and **Display** to review the route.
5. Select **Deploy** to push the route to the IG configuration.

    You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

Policy enforcement in Structured Editor

This section describes how to set up IG as a policy enforcement point in the structured editor of Studio. For more information about setting up policy enforcement, refer to Enforce policy decisions from AM.

1. In IG Studio, create a route:
    1. Go to `http://ig.example.com:8080/openig/studio`, and then select + **Create a route**.
    2. Select ☰ **Structured** to use the structured editor.

2. Select **Advanced options** on the right, and create a route with the following options:
   - **Base URI**: `http://app.example.com:8081`
   - **Condition: Path**: `/home/pep-sso`
   - **Name** : `pep-sso`

   The structured editor is displayed.
3. Configure authentication:
   1. Select **Authentication**.
   2. Select **Single Sign-On**, and enter the following information:
      - **AM service** : Configure an AM service to use for authentication:
        - **URI:** `http://am.example.com:8088/openam`
        - **Secrets Provider:** `SystemAndEnvSecretStore-1`
        - **Agent** : The credentials of the agent you created in AM.
          - **Username** : `ig_agent`
          - **Password Secret ID** : `password.secret.id`

   Leave all other values as default.
4. Configure a PolicyEnforcementFilter:
   1. Select **Authorization**.
   2. Select **AM Policy Enforcement**, and then select the following options:
      - Access Management configuration:
        - **AM service** : `http://am.example.com:8088/openam (/)` .
      - Access Management policies:
        - **Policy set** : `PEP-SSO`
        - **AM SSO token** : `${contexts.ssoToken.value}`

      Leave all other values as default.
5. On the top-right of the screen, select ⋮ and **Display** to review the route.
6. Select **Deploy** to push the route to the IG configuration.

   You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

Policy enforcement for CDSSO in Structured Editor

This section describes how to set up IG as a policy enforcement point for CDSSO in the structured editor of Studio. For more information about how to set up SSO, refer to Enforce AM Policy decisions in different domains.

1. In IG Studio, create a route:
    1. Go to `http://ig.example.com:8080/openig/studio`, and then select + **Create a route**.
    2. Select ☰ **Structured** to use the structured editor.
2. Select **Advanced options** on the right, and create a route with the following options:
    - **Base URI**: `http://app.example.com:8081`
    - **Condition: Path**: `/home/pep-cdsso`
    - **Name** : `pep-cdsso`
3. Configure authentication:
    1. Select 👤 **Authentication**.
    2. Select **Cross-Domain Single Sign-On**, and enter the following information:
        - **AM service** :
            - **URI**: `http://am.example.com:8088/openam`
            - **Secrets Provider:** `SystemAndEnvSecretStore-1`
            - **Agent** : The credentials of the agent you created in AM.
                - **Username** : `ig_agent_cdsso`
                - **Password Secret ID** : `password.secret.id`
        - **Redirect endpoint** : `/home/pep-cdsso/redirect`
        - **Authentication cookie** :
            - **Path** : `/home`

    Leave all other values as default.
4. Configure a PolicyEnforcementFilter:
    1. Select 🔑 **Authorization**.
    2. Select **AM Policy Enforcement**, and select the following options to reflect the configuration of the IG agent in AM:
        - Access Management configuration:
            - **AM service** : `http://am.example.com:8088/openam (/)`.
        - Access Management policies:
            - **Policy set** : `PEP-CDSSO`

- **AM SSO token ID** : `${contexts.cdsso.token}`

Leave all other values as default.

5. On the top-right of the screen, select ⋮ and **Display** to review the route.
6. Select **Deploy** to push the route to the IG configuration.

You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

Token validation using the introspection endpoint in Structured Editor

This section sets up IG as an OAuth 2.0 resource server, using the introspection endpoint, in the structured editor of Studio.

1. Set up AM as described in Validate access tokens through the introspection endpoint. In addition, create an OAuth 2.0 Client authorized to introspect tokens, with the following values:
   - **Client ID** : `resource-server`
   - Client secret `password`
   - **Scope(s)** : `am-introspect-all-tokens`
2. In IG Studio, create a route:
   1. Go to `http://ig.example.com:8080/openig/studio`, and then select + **Create a route**.
   2. Select **Structured** to use the structured editor.
   3. Create a route with the following option:
      - Application URL: `http://app.example.com:8081/rs-introspect-se`
3. Configure authorization:
   1. Select **Authorization** > **OAuth 2.0 Resource Server**, and then select the following options:
      - **Token resolver configuration**:
        - **Access token resolver**: `OAuth 2.0 introspection endpoint`
        - **Introspection endpoint URI**: `http://am.example.com:8088/openam/oauth2/introspect`
        - **Client name** and **Client secret** : `resource-server` and `password`

          This is the name and password of the OAuth 2.0 client with the scope to examine (introspect) tokens, configured in AM.
      - Scope configuration:
        - **Evaluate scopes**: `Statically`
        - **Scopes**: `mail`, `employeenumber`
      - **OAuth 2.0 Authorization settings**:
        - **Require HTTPS**: Deselect this option

- **Enable cache**: Deselect this option

Leave all other values as default.

4. Add a StaticResponseHandler:

   1. On the top-right of the screen, select ⋮ and **Editor mode** to switch into editor mode.

      > **Warning**
      >
      > After switching to Editor mode, you cannot go back. You will be able to use the JSON file editor to manually edit the route, but will no longer be able use the full Studio interface to add or edit filters.

   2. Replace the last ReverseProxyHandler in the route with the following StaticResponseHandler, and then save the route:

      ```
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
         "status": 200,
         "headers": {
           "Content-Type": [ "text/html; charset=UTF-8" ]
         },
         "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></body></
      html>"
        }
      }
      ```

5. On the top-right of the screen, select ⋮ and **Display** to review the route.

6. Select **Deploy** to push the route to the IG configuration.

   You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

OpenID Connect in Structured Editor

This section describes how to set up IG as an OpenID Connect relying party in the structured editor of Studio. For more information, refer to AM as a single OpenID Connect provider.

1. In IG Studio, create a route:

   1. Go to `http://ig.example.com:8080/openig/studio`, and then select + **Create a route**.

   2. Select **Structured** to use the structured editor.

2. Select **Advanced options** on the right, and create a route with the following options:

   - **Base URI**: `http://app.example.com:8081`
   - **Condition: Path**: `/home/id_token`
   - **Name**: `07-openid`

3. Configure authentication:
    1. Select **Authentication**.
    2. Select **OpenID Connect**, and then select the following options:
        - Client Filter:
            - **Client Endpoint**: `/home/id_token`
            - **Require HTTPS**: Deselect this option
        - **Client Registration**:
            - **Client ID**: `oidc_client`
            - **Client secret**: `password`
            - **Scopes**: `openid`, `profile`, and `email`
            - **Basic authentication**: Select this option
        - **Issuer**:
            - **Well-known Endpoint**: `http://am.example.com:8088/openam/oauth2/.well-known/openid-configuration`

    Leave all other values as default.
4. On the top-right of the screen, select ⋮ and **Display** to review the route.
5. Select **Deploy** to push the route to the IG configuration.

    You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

Token transformation in Structured Editor

This section describes how to set up token transformation in the structured editor of Studio. For more information about setting up token transformation, refer to Transform OpenID Connect ID tokens into SAML assertions.

1. In IG Studio, create a route:
    1. Go to `http://ig.example.com:8080/openig/studio`, and then select + **Create a route**.
    2. Select **Structured** to use the structured editor.
2. Select **Advanced options** on the right, and create a route with the following options:
    - **Base URI**: `http://app.example.com:8081`
    - **Condition**: Path: `/home/id_token`
    - **Name** : `50-idtoken`
3. Configure authentication:
    1. Select **Authentication**.

2. Select **OpenID Connect**, and enter the following information:
    - **Client Filter** :
        - **Client Endpoint**: `/home/id_token`
        - **Require HTTPS**: Deselect this option
    - **Client Registration** :
        - **Client ID** : `oidc_client`
        - **Client secret** : `password`
        - **Scopes**: `openid` , `profile` , and `email`
        - **Basic authentication**: Select this option
    - **Issuer** :
        - Well-known endpoint: `http://am.example.com:8088/openam/oauth2/.well-known/openid-configuration`

Leave all other values as default, and save your settings.

4. Set up token transformation:
    1. Select and enable **Token transformation**.
    2. Enter the following information:
        - **AM service** : Configure an AM service to use for authentication and REST STS requests.
            - **URI**: `http://am.example.com:8088/openam`
            - **Secrets Provider:** `SystemAndEnvSecretStore-1`
            - **Agent** : The credentials of the agent you created in AM.
                - **Username** : `ig_agent`
                - **Password Secret ID** : `password.secret.id`
        - **Username** : `oidc_client`
        - **Password** : `password`
        - **id_token** : `${attributes.openid.id_token}`
        - **Instance** : `openig`
5. Add a StaticResponseHandler:
    1. On the top-right of the screen, select ⋮ and **Editor mode** to switch into editor mode.

> **Warning**
>
> After switching to Editor mode, you cannot go back. You will be able to use the JSON file editor to manually edit the route, but will no longer be able use the full Studio interface to add or edit filters.

2. Replace the last ReverseProxyHandler in the route with the following StaticResponseHandler, and then save the route:

```
"handler": {
  "type": "StaticResponseHandler",
  "config": {
    "status": 200,
    "headers": {
      "Content-Type": [ "text/plain; charset=UTF-8" ]
    },
    "entity": "{\"id_token\":\n\"${attributes.openid.id_token}\"} \n\n\n{\"saml_assertions\":\n\"$
{contexts.sts.issuedToken}\"}"
  }
}
```

6. On the top-right of the screen, select ⋮ and **Display** to review the route.
7. Select **Deploy** to push the route to the IG configuration.

   You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

Simple throttling filter in Structured Editor

This section describes how to set up a simple throttling filter in the structured editor of Studio. For more information about how to set up throttling, refer to Configure simple throttling.

1. In IG Studio, create a route:
   1. Go to `http://ig.example.com:8080/openig/studio`, and then select + **Create a route**.
   2. Select **Structured** to use the structured editor.
2. Select **Advanced options** on the right, and create a route with the following options:
   - **Base URI**: `http://app.example.com:8081`
   - **Condition: Path**: `/home/throttle-simple`
   - **Name** : `00-throttle-simple`
3. Select and enable **Throttling**.
4. In **GROUPING POLICY**, apply the rate to a single group.

   All requests are grouped together, and the default throttling rate is applied to the group. By default, no more than 100 requests can access the sample application each second.
5. In **RATE POLICY**, select **Fixed**, and allow 6 requests each 10 seconds.
6. On the top-right of the screen, select ⋮ and **Display** to review the route.
7. Select **Deploy** to push the route to the IG configuration.

   You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

Mapped throttling filter in Structured Editor

This section describes how to set up a mapped throttling filter in the structured editor of Studio. For more information about how to set up throttling, refer to Configure mapped throttling.

1. Set up AM as described in Validate access tokens through the introspection endpoint. In addition, create an OAuth 2.0 Client authorized to introspect tokens, with the following values:
   - **Client ID** : `resource-server`
   - Client secret `password`
   - **Scope(s)** : `am-introspect-all-tokens`
2. In IG Studio, create a route:
   1. Go to `http://ig.example.com:8080/openig/studio`, and then select + **Create a route**.
   2. Select ☰ **Structured** to use the structured editor.
3. Select **Advanced options** on the right, and create a route with the following options:
   - **Base URI**: `http://app.example.com:8081`
   - **Condition: Path**: `/home/throttle-mapped-se`
   - **Name** : `00-throttle-mapped-se`
4. Configure authorization:
   1. Select 🔑 **Authorization** > **OAuth 2.0 Resource Server**, and then select the following options:
      - **Token resolver configuration**:
        - **Access token resolver**: `OAuth 2.0 introspection endpoint`
        - **Introspection endpoint URI**: `http://am.example.com:8088/openam/oauth2/introspect`
        - **Client name** and **Client secret** : `resource-server` and `password`

          This is the name and password of the OAuth 2.0 client with the scope to examine (introspect) tokens, configured in AM.
      - Scope configuration:
        - **Evaluate scopes**: `Statically`
        - **Scopes**: `mail`, `employeenumber`
      - **OAuth 2.0 Authorization settings**:
        - **Require HTTPS**: Deselect this option
        - **Enable cache**: Deselect this option

   Leave all other values as default.

5. Configure throttling:
    1. Select and enable **Throttling**.
    2. Set up the grouping policy:
        1. In **GROUPING POLICY**, apply the rate to independent groups of requests.

            Requests are split into different groups according to criteria, and the throttling rate is applied to each group.
        2. Select to group requests by custom criteria.

            Enter `${contexts.oauth2.accessToken.info.mail}` as the custom expression. This expression defines the subject in the OAuth2Context.
    3. Set up the rate policy:
        1. In **RATE POLICY**, select **Mapped**.
        2. Select to map requests by custom criteria.
        3. Enter the custom expression `${contexts.oauth2.accessToken.info.status}`.
        4. In **Default Rate**, select Edit and change default rate to 1 request each 10 seconds.
        5. In **Mapped Rates**, add the following rate for `gold` status:
            - **Match Value** : `gold`
            - **Number of requests** : `6`
            - **Period** : `10 seconds`
        6. Add a different rate for `silver` status:
            - **Match Value** : `silver`
            - **Number of requests** : `3`
            - **Period** : `10 seconds`
        7. Add a different rate for `bronze` status:
            - **Match Value** : `bronze`
            - **Number of requests** : `1`
            - **Period** : `10 seconds`
        8. Save the rate policy.
6. Select **Chain**, and change the order of the filters so that **Throttling** comes after **Authorization**.
7. On the top-right of the screen, select and **Display** to review the route.
8. Select **Deploy** to push the route to the IG configuration.

    You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

Scriptable throttling filter in Structured Editor

This section describes how to set up a scriptable throttling filter in the structured editor of Studio. For more information about how to set up throttling, refer to Configure scriptable throttling.

1. Set up AM as described in Validate access tokens through the introspection endpoint. In addition, create an OAuth 2.0 Client authorized to introspect tokens, with the following values:
   - **Client ID**: `resource-server`
   - **Client secret**: `password`
   - **Scope(s)**: `am-introspect-all-tokens`
2. In IG Studio, create a route:
   1. Go to `http://ig.example.com:8080/openig/studio`, and then select + **Create a route**.
   2. Select ☰ **Structured** to use the structured editor.
3. Select **Advanced options** on the right, and create a route with the following options:
   - **Base URI**: `http://app.example.com:8081`
   - **Condition: Path**: `/home/throttle-scriptable-se`
   - **Name**: `00-throttle-scriptable-se`
4. Configure authorization:
   1. Select 🔑 **Authorization** > **OAuth 2.0 Resource Server**, and then select the following options:
      - **Token resolver configuration**:
        - **Access token resolver**: `OAuth 2.0 introspection endpoint`
        - **Introspection endpoint URI**: `http://am.example.com:8088/openam/oauth2/introspect`
        - **Client name** and **Client secret** : `resource-server` and `password`

          This is the name and password of the OAuth 2.0 client with the scope to examine (introspect) tokens, configured in AM.
      - Scope configuration:
        - **Evaluate scopes**: `Statically`
        - **Scopes**: `mail`, `employeenumber`
      - **OAuth 2.0 Authorization settings**:
        - **Require HTTPS**: Deselect this option
        - **Enable cache**: Deselect this option

   Leave all other values as default.

5. Configure throttling:

    1. Select and enable **Throttling**.
    2. Set up the grouping policy:
        1. In **GROUPING POLICY**, apply the rate to independent groups of requests.

           Requests are split into different groups according to criteria, and the throttling rate is applied to each group.
        2. Select to group requests by custom criteria.
        3. Enter `${contexts.oauth2.accessToken.info.mail}` as the custom expression.
    3. Set up the rate policy:
        1. In **RATE POLICY**, select **Scripted**.
        2. Select to create a new script, and name it `X-User-Status`. So that you can easily identify the script, use a name that describes the content of the script.
        3. Add the following argument/value pairs:
            - **argument**: `status`, **value**: `gold`
            - **argument**: `rate`, **value**: `6`
            - **argument**: `duration`, **value**: `10 seconds`
                - Replace the default script with the content of a valid Groovy script. For example, enter the following script:

                ```
                if (contexts.oauth2.accessToken.info.status == status) {
                  return new ThrottlingRate(rate, duration)
                } else {
                  return null
                }
                ```

                Alternatively, skip the step to define arguments, and add the following script instead:

                ```
                if (contexts.oauth2.accessToken.info.status == 'gold') {
                  return new ThrottlingRate(6, '10 seconds')
                } else {
                  return null
                }
                ```

                > **Note**
                >
                > Studio does not check the validity of the Groovy script.
        4. Enable the default rate, and set it to 1 request each 10 seconds.
        5. Save the rate policy. The script is added to the list of reference scripts available to use in scriptable throttling filters.

6. Select **Chain**, and change the order of the filters so that **Throttling** comes after **Authorization**.
7. On the top-right of the screen, select ⋮ and **Display** to review the route.
8. Select **Deploy** to push the route to the IG configuration.

   You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

Proxy for websocket traffic in Structured Editor

This section describes how to set up IG to proxy WebSocket traffic, in the structured editor of Studio. For more information about how to set up proxying for WebSocket traffic, refer to WebSocket traffic.

1. In IG Studio, create a route:
   1. Go to `http://ig.example.com:8080/openig/studio`, and then select + **Create a route**.
   2. Select **Structured** to use the structured editor.
2. Select **Advanced options** on the right, and create a route with the following options:
   - **Base URI**: `http://app.example.com:8081`
   - **Condition: Path**: `/websocket-se`
   - **Name** : `websocket-se`
   - **Enable WebSocket**: Select this option
     1. Select **Authentication**.
     2. Select **Single Sign-On**, and enter the following information:
        - **AM service** : Configure an AM service to use for authentication:
          - **URI:** `http://am.example.com:8088/openam`
          - **Secrets Provider:** `SystemAndEnvSecretStore-1`
          - **Agent** :
            - **Username** : `ig_agent`
            - **Password Secret ID** : `password.secret.id`

   Leave all other values as default.
3. On the top-right of the screen, select ⋮ and **Display** to review the route.
4. Select **Deploy** to push the route to the IG configuration.

   You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

Example routes created with Freeform Designer

The following sections give examples of how to use the templates provided by the FreeForm Designer:

> **Important**
>
> SecretsProviders can't be configured in Studio. Documentation examples generated with Studio might refer to SecretsProviders that must be configured separately in `config.json`.

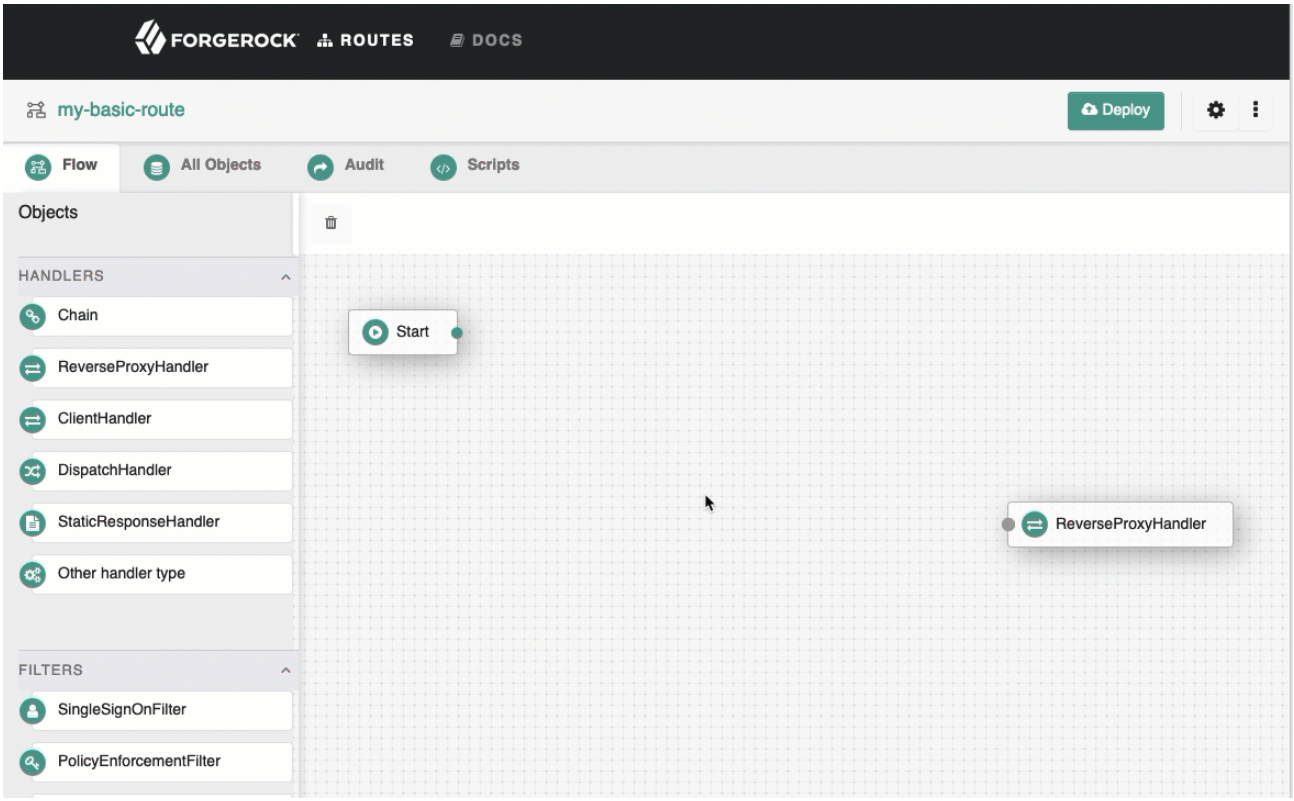
Use a basic template in FreeForm Designer

This section describes how to use a basic template in FreeForm Designer to set up SSO. For more information about setting up and testing SSO, refer to Authentication.

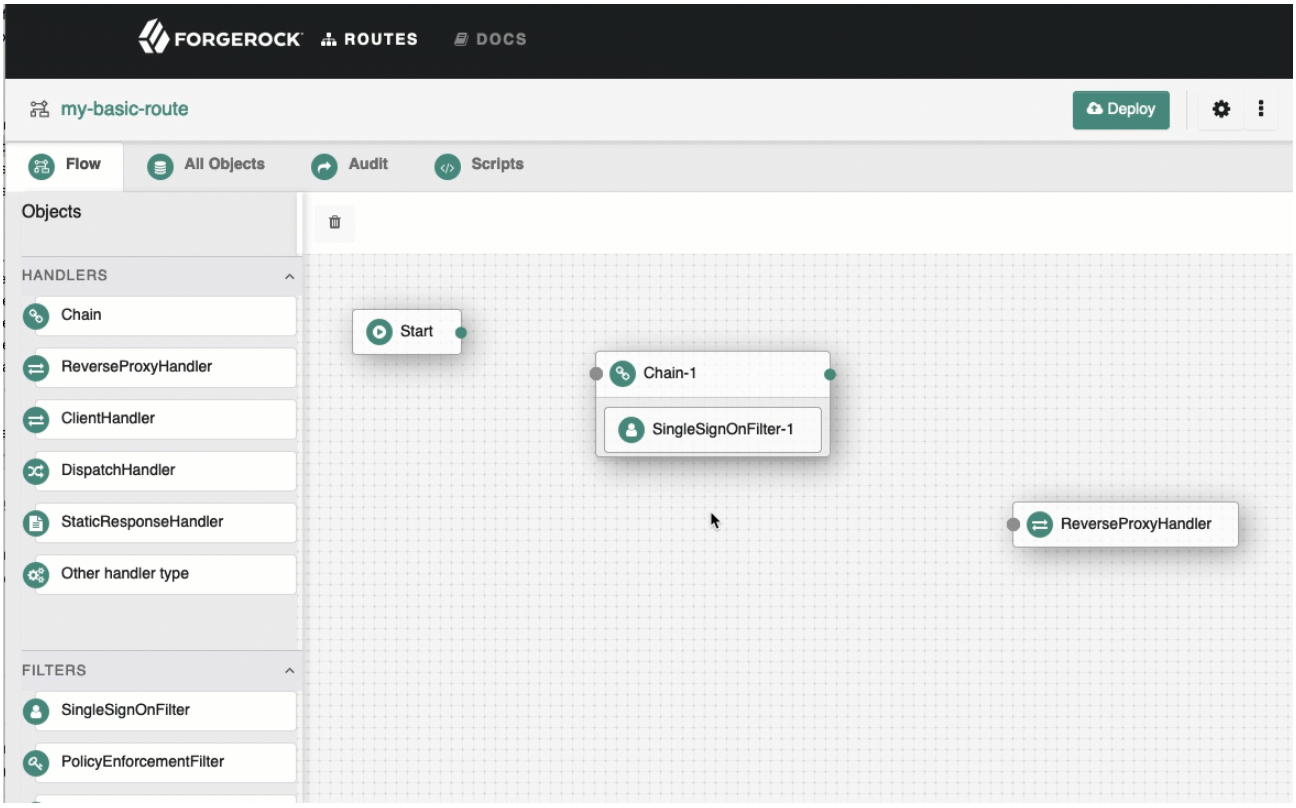
1. In IG Studio, create a route:
    1. Go to `http://ig.example.com:8080/openig/studio`, and then select + **Create a route**.
    2. Select **Freeform** to use the FreeForm Designer.
2. Select **Basic** to create a route from a blank template.
3. Select **Advanced options** on the right, and create a route with the following options:
    - **Base URI**: `http://app.example.com:8081`
    - **Condition: Path**: `/home/sso-ff`
    - **Name**: `sso-ff`

      The route is displayed on the **Flow** tab of the canvas. Select the **All Objects** tab to view a list of objects in the route.

      Double-click on any object to review or edit it. After double-clicking on an object, select the **Decorations** tab to decorate it.
4. Configure authentication with a SingleSignOnFilter:
    1. In the **Flow** tab, delete the connector between **Start** and **ReverseProxyHandler**.
    2. From the side bar, drag a **Chain** onto the canvas, and then drag a **SingleSignOnFilter** into the chain.



3. In the **Edit SingleSignOnFilter** page, click +, and create an AM service, with the following values:
    - **URI:** `http://am.example.com:8088/openam`
    - **Secrets Provider:** `SystemAndEnvSecretStore-1`
    - **Agent**:
        - **Username**: `ig_agent`
        - **Password Secret ID** : `agent.secret.id`
4. Connect **Start** to **Chain-1**, and **Chain-1** to **ReverseProxyHandler**.



5. On the top-right of the screen, select ⋮ and **Display** to review the route.

```
{
  "name": "sso-ff",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/sso-ff')}",
  "handler": "Chain-1",
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler"
    },
    {
      "type": "BaseUriDecorator",
      "name": "baseUri"
    },
    {
      "type": "TimerDecorator",
      "name": "timer",
      "config": {
        "timeUnit": "ms"
      }
    },
    {
      "type": "CaptureDecorator",
      "name": "capture",
      "config": {
        "captureEntity": false,
        "captureContext": false,
        "maxEntityLength": 524288
      }
    },
    {
      "name": "Chain-1",
      "type": "Chain",
      "config": {
        "handler": "ReverseProxyHandler",
        "filters": [
          "SingleSignOnFilter-1"
        ]
      }
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "sessionCache": {
          "enabled": false
        }
      }
    },
    {
      "name": "SingleSignOnFilter-1",
      "type": "SingleSignOnFilter",
      "config": {
```

```
          "amService": "AmService-1"
        }
      }
    ]
  }
```

6. Select **Deploy** to push the route to the IG configuration.

   You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

Protect a web app with Freeform Designer

This section describes how to use FreeForm Designer to protect a web app, using AM for single sign-on and policy enforcement.

The generated route contains a chain of objects to authenticate the user, enforce an AM authorization policy, retrieve the user's profile, insert it into the request, and, finally, forward the request to the web app.

Before you start, set up AM as described in Enforce policy decisions from AM.

1. In IG Studio, create a route:
   1. Go to `http://ig.example.com:8080/openig/studio`, and then select **+ Create a route**.
   2. Select **Freeform** to use the FreeForm Designer.
2. Select **Web SSO** to use the template for protecting web apps.
3. Select **Advanced options** on the right, and create a route with the following options:
   - **Base URI** : `http://app.example.com:8081`
   - **Condition: Path** : `/home/pep-sso-ff`
   - **Name** : `pep-sso-ff`
   - **AM Configuration** :
     - **URI** : `http://am.example.com:8088/openam`
     - **Secrets Provider:** `SystemAndEnvSecretStore-1`
     - **Username** : `ig_agent`
     - **Password Secret ID** : `agent.secret.id`

       The route is displayed on the **Flow** tab of the canvas. Select the **All Objects** tab to view a list of objects in the route.

       Double-click on any object to review or edit it. After double-clicking on an object, select the **Decorations** tab to decorate it.
4. On the **Flow** tab, double-click the Policy Enforcement object, and add a policy set with the following values:
   - **Policy set** : `PEP-SSO`
   - **AM SSO token** : `${contexts.ssoToken.value}`

Leave all other values as default.

5. On the top-right of the screen, select ⋮ and **Display** to review the route.

```
{
  "name": "pep-sso-ff",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/pep-sso-ff')}",
  "handler": "Chain",
  "heap": [
    {
      "name": "Chain",
      "type": "Chain",
      "config": {
        "handler": "ReverseProxyHandler",
        "filters": [
          "SSO",
          "Policy Enforcement",
          "GetEmail",
          "InjectEmail"
        ]
      }
    },
    {
      "name": "SSO",
      "type": "SingleSignOnFilter",
      "config": {
        "amService": "AmService"
      }
    },
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler"
    },
    {
      "name": "AmService",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "sessionCache": {
          "enabled": false
        }
      }
    },
    {
      "name": "Policy Enforcement",
      "type": "PolicyEnforcementFilter",
      "config": {
        "amService": "AmService",
        "ssoTokenSubject": "${contexts.ssoToken.value}",
        "cache": {
          "enabled": false
        },
        "application": "PEP-SSO"
      }
    },
    {
      "name": "GetEmail",
```

```
        "type": "UserProfileFilter",
        "config": {
          "username": "${contexts.ssoToken.info.uid}",
          "userProfileService": {
            "type": "UserProfileService",
            "config": {
              "amService": "AmService"
            }
          }
        }
      },
      {
        "name": "InjectEmail",
        "type": "HeaderFilter",
        "config": {
          "messageType": "REQUEST",
          "add": {
            "Email": [
              "${contexts.userProfile.username}"
            ]
          }
        }
      },
      {
        "type": "BaseUriDecorator",
        "name": "baseUri"
      },
      {
        "type": "TimerDecorator",
        "name": "timer",
        "config": {
          "timeUnit": "ms"
        }
      },
      {
        "type": "CaptureDecorator",
        "name": "capture",
        "config": {
          "captureEntity": false,
          "captureContext": false,
          "maxEntityLength": 524288
        }
      }
    ]
  }
```

6. Select **Deploy** to push the route to the IG configuration.

   You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

Protect an API with Freeform Designer

This section describes how to use FreeForm Designer to protect APIs, using AM as an OAuth 2.0 Authorization Server.

The generated route contains a chain of objects to authenticate the user, throttle the rate of requests to the API, and, finally, forward the request to the sample app.

Before you start, set up AM as described in Validate access tokens through the introspection endpoint. In addition, create an OAuth 2.0 Client authorized to introspect tokens, with the following values:

- **Client ID** : `resource-server`
- **Client secret** : `password`
- **Scope(s)** : `am-introspect-all-tokens`

1. In IG Studio, create a route:
    1. Go to `http://ig.example.com:8080/openig/studio`, and then select + **Create a route**.
    2. Select **Freeform** to use the FreeForm Designer.
2. Select **API Security**.
3. Select **Advanced options** on the right, and create a route with the following options:
    - **Base URI** : `http://app.example.com:8081`
    - **Condition: Path** : `/home/rs-introspect-ff`
    - **Name** : `rs-introspect-ff`
    - **AM Configuration** :
        - **URI** : `http://am.example.com:8088/openam`
        - **Secrets Provider:** `SystemAndEnvSecretStore-1`
        - **Username** : `ig_agent`
        - **Password Secret ID** : `agent.secret.id`
        - **Scopes** : `mail` , `employeenumber`

        The route is displayed on the **Flow** tab of the canvas.

        Notice that the **Start**, **Chain**, and **ReverseProxyHandler** objects are connected by solid lines, but other objects, such as **Authenticate to Am Chain**, are connected by a fading line. Objects connected by a fading line are used by other objects in the route.

        Select the **All Objects** tab to view a list of objects in the route. Double-click on any object to review or edit it. After double-clicking on an object, select the **Decorations** tab to decorate it.
4. On the **Flow** tab, double-click the OAuth2RS object, and edit it as follows:
    - **Require HTTPS** : Deselect this option
    - **Realm** : `OpenIG`

    Leave the other values as they are.
5. On the top-right of the screen, select ⋮ and **Display** to review the route.

```
{
  "name": "rs-introspect-ff",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/rs-introspect-ff')}",
  "handler": "Chain",
  "properties": {
    "amSecretsProvider": "SystemAndEnvSecretStore-1",
    "amUsername": "ig_agent",
    "amPasswordSecretId": "agent.secret.id"
  },
  "heap": [
    {
      "name": "ClientHandler",
      "type": "ClientHandler"
    },
    {
      "name": "Chain",
      "type": "Chain",
      "config": {
        "handler": "ReverseProxyHandler",
        "filters": [
          "OAuth2RS",
          "Throttling"
        ]
      }
    },
    {
      "type": "OAuth2ResourceServerFilter",
      "name": "OAuth2RS",
      "config": {
        "requireHttps": false,
        "realm": "OpenIG",
        "scopes": [
          "mail",
          "employeenumber"
        ],
        "accessTokenResolver": "TokenIntrospectionAccessTokenResolver"
      }
    },
    {
      "type": "TokenIntrospectionAccessTokenResolver",
      "name": "TokenIntrospectionAccessTokenResolver",
      "config": {
        "amService": "AmService",
        "providerHandler": "Authenticate to AM Chain"
      }
    },
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler"
    },
    {
      "name": "AmService",
      "type": "AmService",
      "config": {
        "url": "http://am.example.com:8088/openam",
        "realm": "/",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "agent": {
          "username": "ig_agent",
```

```
            "passwordSecretId": "agent.secret.id"
          },
          "sessionCache": {
            "enabled": false
          }
        }
      },
      {
        "name": "Authenticate to AM Chain",
        "type": "Chain",
        "config": {
          "handler": "ClientHandler",
          "filters": [
            "Authenticate to AM Filter"
          ]
        }
      },
      {
        "name": "Authenticate to AM Filter",
        "type": "HttpBasicAuthenticationClientFilter",
        "config": {
          "username": "ig_agent",
          "passwordSecretId": "password.secret.id",
          "secretsProvider": "SystemAndEnvSecretStore-1"
        }
      },
      {
        "name": "Throttling",
        "type": "ThrottlingFilter",
        "config": {
          "requestGroupPolicy": "${contexts.oauth2.info.sub}",
          "rate": {
            "numberOfRequests": 60,
            "duration": "60 s"
          }
        }
      },
      {
        "type": "BaseUriDecorator",
        "name": "baseUri"
      },
      {
        "type": "TimerDecorator",
        "name": "timer",
        "config": {
          "timeUnit": "ms"
        }
      },
      {
        "type": "CaptureDecorator",
        "name": "capture",
        "config": {
          "captureEntity": false,
          "captureContext": false,
          "maxEntityLength": 524288
        }
      }
    ]
}
```

6. Select **Deploy** to push the route to the IG configuration.

   You can check the `$HOME/.openig/config/routes` folder to see that the route is there.

7. Test the setup:

   1. In a terminal window, use a `curl` command similar to the following to retrieve an access token:

      ```
      $ mytoken=$(curl -s \
      --user "client-application:password" \
      --data "grant_type=password&username=demo&password=Ch4ng31t&scope=mail%20employeenumber" \
      http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
      ```

   2. Validate the access token returned in the previous step:

      ```
      $ curl -v http://ig.example.com:8080/home/rs-introspect-ff --header "Authorization: Bearer ${mytoken}"
      ```

      The HTML of the sample application is returned.

Summary of tasks, route status, and icons

The following tables summarize the basic tasks that you can do in Studio, and summarizes the icons and status displayed in Studio:

Task reference

To do this	Do this
Create a new route	Select **ROUTES**, **+ Create a route**.
Select a route	Select **ROUTES**, and then select a route to view.
Display the config of a selected route	Select a route, and then select ⋮ and **Display**.
Deploy a selected route	Select a route, and then select **Deploy**.
Undeploy a selected route	Select a deployed route, and then select ⋮ and **✕ Undeploy**.
Change the basic config of a route	Select a route, and then select **Route settings**. Edit the route and save the changes.

Route status

Status	Description	Action
Undeployed	The route is saved in Studio but is not deployed to the backend.	Deploy the route. The status changes to **Deployed**.
Deployed	The route is saved in Studio and deployed to the backend.	None. The route has the same configuration in Studio and the backend.
Changes pending	The route has been deployed and then subsequently changed in Studio.	Deploy the route. The status changes to **Deployed**.
Out of sync	The route has been deployed and then subsequently changed in the backend, or in both Studio and the backend.	Select **Deploy**. A message informs you that a different version of the route is deployed in the backend. Select an option: • **Deploy**: The version in Studio overwrites the backend. • **Import a route**: The version in the backend overwrites Studio. When you import a route into Studio you go into **editor mode**. You can use the JSON editor to manually edit the route, but can no longer use the full Studio interface to add or edit filters.
Compatibility update required	The route was created in Studio in an earlier version of IG. Some information is needed to complete the upgrade.	Enter the information as prompted, and then select **Deploy** to deploy the route.

Icons

Icon	Mode	Description
	Structured editor	The route was created and edited using the menus and options of structured editor.
{ }	Editor mode	The route was imported into Studio, or was created in Studio and then edited in editor mode.
	Freeform designer	The route was created on the canvas of FreeForm Designer.

Maintenance

This guide describes tasks and configurations you might repeat throughout the life cycle of a deployment in your organization. It is for people who maintain IG services for their organization.

Audit the deployment

The following sections describe how to set up auditing for your deployment. For information about how to include user ID in audit logs, refer to Recording User ID in Audit Events.

For information about the audit framework and each event handler, refer to Audit framework.

Record access audit events in CSV

This section describes how to record access audit events in a CSV file, using tamper-evident logging. For information about the CSV audit event handler, refer to CsvAuditEventHandler.

> **Important**
>
> The CSV handler does not sanitize messages when writing to CSV log files.
> Do not open CSV logs in spreadsheets or other applications that treat data as code.

Before you start, prepare IG and the sample application as described in the Quick install.

1. Set up secrets for tamper-evident logging:

    1. Locate a directory for secrets, and go to it:

        ```
        $ cd /path/to/secrets
        ```

    2. Generate a key pair in the keystore.

        The CSV event handler expects a JCEKS-type keystore with a key alias of `signature` for the signing key, where the key is generated with the `RSA` key algorithm and the `SHA256withRSA` signature algorithm:

        ```
        $ keytool \
         -genkeypair \
         -keyalg RSA \
         -sigalg SHA256withRSA \
         -alias "signature" \
         -dname "CN=ig.example.com,O=Example Corp,C=FR" \
         -keystore audit-keystore \
         -storetype JCEKS \
         -storepass password \
         -keypass password
        ```

        > **Note**
        >
        > Because keytool converts all characters in its key aliases to lowercase, use only lowercase in alias definitions of a keystore.

3. Generate a secret key in the keystore.

   The CSV event handler expects a JCEKS-type keystore with a key alias of `csv-key-2` for the symmetric key, where the key is generated with the `HmacSHA256` key algorithm and 256-bit key size:

   ```
   $ keytool \
    -genseckey \
    -keyalg HmacSHA256 \
    -keysize 256 \
    -alias "password" \
    -keystore audit-keystore \
    -storetype JCEKS \
    -storepass password \
    -keypass password
   ```

4. Verify the content of the keystore:

   ```
   $ keytool \
    -list \
    -keystore audit-keystore \
    -storetype JCEKS \
    -storepass password

   Keystore type: JCEKS
   Keystore provider: SunJCE

   Your keystore contains 2 entries

   password, ... SecretKeyEntry,
   signature, ... PrivateKeyEntry,
   Certificate fingerprint (SHA1): 4D:...:D1
   ```

2. Set up IG

   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

   2. Add the following route to IG, replacing `/path/to/secrets/audit-keystore` with your path:

      Linux

      ```
      $HOME/.openig/config/routes/30-csv.json
      ```

      Windows

      ```
      %appdata%\OpenIG\config\routes\30-csv.json
      ```

```
{
  "name": "30-csv",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/csv-audit')}",
  "heap": [
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.csv.CsvAuditEventHandler",
            "config": {
              "name": "csv",
              "logDirectory": "/tmp/logs",
              "security": {
                "enabled": "true",
                "filename": "/path/to/secrets/audit-keystore",
                "password": "password",
                "signatureInterval": "1 day"
              },
              "topics": [
                "access"
              ]
            }
          }
        ],
        "config": { }
      }
    }
  ],
  "auditService": "AuditService",
  "handler": "ForgeRockClientHandler"
}
```

The route calls an audit service configuration for publishing log messages to the CSV file, `/tmp/logs/access.csv`.

When a request matches `audit`, audit events are logged to the CSV file.

The route uses the `ForgeRockClientHandler` as its handler, to send the `X-ForgeRock-TransactionId` header with its requests to external services.

3. Test the setup:

    1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/csv-audit.

        The home page of the sample application is displayed, and the file `/tmp/logs/tamper-evident-access.csv` is updated.

Record access audit events with a JMS audit event handler

> **Important**
>
> This procedure is an example of how to record access audit events with a JMS audit event handler configured to use the ActiveMQ message broker. This example is not tested on all configurations, and can be more or less relevant to your configuration.

For information about configuring the JMS event handler, refer to JmsAuditEventHandler.

Before you start, prepare IG as described in the Quick install.

1. Download the following files:
   - ActiveMQ binary. IG is tested with ActiveMQ Classic 5.15.11.
   - ActiveMQ Client. Use a version that corresponds to your ActiveMQ version.
   - Apache Geronimo J2EE management bundle.
   - hawtbuf-1.11 JAR.
2. Add the files to the configuration:
   - Create the directory `$HOME/.openig/extra`, where `$HOME/.openig` is the instance directory, and add .jar files to the directory.
3. Create a consumer that subscribes to the `audit` topic.

   From the ActiveMQ installation directory, run the following command:

   ```
   $ ./bin/activemq consumer --destination topic://audit
   ```

4. Set up IG
   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
   2. Add the following route to IG:

      Linux

      ```
      $HOME/.openig/config/routes/30-jms.json
      ```

Windows

```
%appdata%\OpenIG\config\routes\30-jms.json
```

```
{
  "name": "30-jms",
  "MyCapture" : "all",
  "baseURI": "http://app.example.com:8081",
  "condition" : "${request.uri.path == '/activemq_event_handler'}",
  "heap": [
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers" : [
          {
            "class" : "org.forgerock.audit.handlers.jms.JmsAuditEventHandler",
            "config" : {
              "name" : "jms",
              "topics": [ "access" ],
              "deliveryMode" : "NON_PERSISTENT",
              "sessionMode" : "AUTO",
              "jndi" : {
                "contextProperties" : {
                  "java.naming.factory.initial" :
"org.apache.activemq.jndi.ActiveMQInitialContextFactory",
                  "java.naming.provider.url" : "tcp://am.example.com:61616",
                  "topic.audit" : "audit"
                },
                "topicName" : "audit",
                "connectionFactoryName" : "ConnectionFactory"
              }
            }
          }
        ],
        "config" : { }
      }
    }
  ],
  "auditService": "AuditService",
  "handler" : {
    "type" : "StaticResponseHandler",
    "config" : {
      "status" : 200,
      "headers" : {
        "Content-Type" : [ "text/plain; charset=UTF-8" ]
      },
      "entity" : "Message from audited route"
    }
  }
}
```

When a request matches the `/activemq_event_handler` route, this configuration publishes JMS messages containing audit event data to an ActiveMQ managed JMS topic, and the StaticResponseHandler displays a message.

5. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/activemq_event_handler.

      Depending on how ActiveMQ is configured, audit events are displayed on the ActiveMQ console or written to file.

Record access audit events with a JSON audit event handler

This section describes how to record access audit events with a JSON audit event handler. For information about configuring the JSON event handler, refer to JsonAuditEventHandler.

1. Set up IG

   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

   2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/30-json.json
```

Windows

```
%appdata%\OpenIG\config\routes\30-json.json
```

```
{
  "name": "30-json",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/json-audit')}",
  "heap": [
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
            "config": {
              "name": "json",
              "logDirectory": "/tmp/logs",
              "topics": [
                "access"
              ],
              "rotationRetentionCheckInterval": "1 minute",
              "buffering": {
                "maxSize": 100000,
                "writeInterval": "100 ms"
              }
            }
          }
        ]
      }
    }
  ],
  "auditService": "AuditService",
  "handler": "ReverseProxyHandler"
}
```

Notice the following features of the route:

- The route calls an audit service configuration for publishing log messages to the JSON file, `/tmp/audit/access.audit.json`. When a request matches `/home/json-audit`, a single line per audit event is logged to the JSON file.
- The route uses the `ForgeRockClientHandler` as its handler, to send the `X-ForgeRock-TransactionId` header with its requests to external services.

2. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/json-audit.

      The home page of the sample application is displayed and the file `/tmp/logs/access.audit.json` is created or updated with a message. The following example message is formatted for easy reading, but it is produced as a single line for each event:

```
{
  "_id": "830...-41",
  "timestamp": "2019-...540Z",
  "eventName": "OPENIG-HTTP-ACCESS",
  "transactionId": "830...-40",
  "client": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 51666
  },
  "server": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 8080
  },
  "http": {
    "request": {
      "secure": false,
      "method": "GET",
      "path": "http://ig.example.com:8080/home/json-audit",
      "headers": {
        "accept": ["text/html,application/xhtml+xml,application/xml;q=0.9,/;q=0.8"],
        "host": ["ig.example.com:8080"],
        "user-agent": ["Mozilla/5.0 ... Firefox/66.0"]
      }
    }
  },
  "response": {
    "status": "SUCCESSFUL",
    "statusCode": "200",
    "elapsedTime": 212,
    "elapsedTimeUnits": "MILLISECONDS"
  },
  "ig": {
    "exchangeId": "b3f...-29",
    "routeId": "30-json",
    "routeName": "30-json"
  }
}
```


Record access audit events to standard output

This section describes how to record access audit events to standard output. For more information about the event handler, refer to JsonStdoutAuditEventHandler.

Before you start, prepare IG and the sample application as described in the Quick install.

1. Set up IG

    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

    2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/30-jsonstdout.json
```

Windows

```
%appdata%\OpenIG\config\routes\30-jsonstdout.json
```

```
{
  "name": "30-jsonstdout",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/jsonstdout-audit')}",
  "heap": [
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.json.stdout.JsonStdoutAuditEventHandler",
            "config": {
              "name": "jsonstdout",
              "elasticsearchCompatible": false,
              "topics": [
                "access"
              ]
            }
          }
        ],
        "config": {}
      }
    }
  ],
  "auditService": "AuditService",
  "handler": "ReverseProxyHandler"
}
```

Notice the following features of the route:

- The route matches requests to `/home/jsonstdout-audit`.
- The route calls the audit service configuration for publishing access log messages to standard output. When a request matches `/home/jsonstdout-audit`, a single line per audit event is logged.

2. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/jsonstdout-audit.

   The home page of the sample application is displayed, and a message like this is published to standard output:

```
{
  "_id": "830...-61",
  "timestamp": "2019-...89Z",
  "eventName": "OPENIG-HTTP-ACCESS",
  "transactionId": "830...-60",
  "client": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 51876
  },
  "server": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 8080
  },
  "http": {
    "request": {
      "secure": false,
      "method": "GET",
      "path": "http://ig.example.com:8080/home/jsonstdout-audit",
      "headers": {
        "accept": ["text/html,application/xhtml+xml,application/xml;q=0.9,/;q=0.8"],
        "host": ["ig.example.com:8080"],
        "user-agent": ["Mozilla/5.0 ... Firefox/66.0"]
      }
    }
  },
  "response": {
    "status": "SUCCESSFUL",
    "statusCode": "200",
    "elapsedTime": 10,
    "elapsedTimeUnits": "MILLISECONDS"
  },
  "ig": {
    "exchangeId": "b3f...-41",
    "routeId": "30-jsonstdout",
    "routeName": "30-jsonstdout"
  },
  "source": "audit",
  "topic": "access",
  "level": "INFO"
}
```


Trust transaction IDs from other products

Each audit event is identified by a unique transaction ID that can be communicated across products and recorded for each local event. By using the transaction ID, requests can be tracked as they traverse the platform, making it easier to monitor activity and to enrich reports.

The `X-ForgeRock-TransactionId` header is automatically set in all outgoing HTTP calls from one ForgeRock product to another. Customers can also set this header themselves from their own applications or scripts that call into the ForgeRock Identity Platform.

To reduce the risk of malicious attacks, by default IG does not trust transaction ID headers from client applications.

If you trust the transaction IDs sent by your client applications, consider setting Java system property `org.forgerock.http.TrustTransactionHeader` to `true`.

Add the following system property in `env.sh`:

```
# Specify a JVM option
TX_HEADER_OPT="-Dorg.forgerock.http.TrustTransactionHeader=true"

# Include it into the JAVA_OPTS environment variable
export JAVA_OPTS="${TX_HEADER_OPT}"
```

All incoming `X-ForgeRock-TransactionId` headers are trusted, and monitoring or reporting systems that consume the logs can allow requests to be correlated as they traverse multiple servers.

Safelist audit event fields for the logs

To prevent logging of sensitive data for an audit event, the Common Audit Framework uses a safelist to specify which audit event fields appear in the logs.

By default, only safelisted audit event fields are included in the logs. For information about how to include non-safelisted audit event fields, or exclude safelisted audit event fields, refer to Including or excluding audit event fields in logs.

Audit event fields use JSON pointer notation, and are taken from the JSON schema for the audit event content. The following event fields are safelisted:

- `/_id`
- `/timestamp`
- `/eventName`
- `/transactionId`
- `/trackingIds`
- `/userId`
- `/client`
- `/server`
- `/ig/exchangeId`
- `/ig/routeId`
- `/ig/routeName`
- `/http/request/secure`
- `/http/request/method`
- `/http/request/path`
- `/http/request/headers/accept`
- `/http/request/headers/accept-api-version`
- `/http/request/headers/content-type`

- `/http/request/headers/host`
- `/http/request/headers/user-agent`
- `/http/request/headers/x-forwarded-for`
- `/http/request/headers/x-forwarded-host`
- `/http/request/headers/x-forwarded-port`
- `/http/request/headers/x-forwarded-proto`
- `/http/request/headers/x-original-uri`
- `/http/request/headers/x-real-ip`
- `/http/request/headers/x-request-id`
- `/http/request/headers/x-requested-with`
- `/http/request/headers/x-scheme`
- `/request`
- `/response`

Include or exclude audit event fields in logs

The safelist is designed to prevent logging of sensitive data for audit events by specifying which audit event fields appear in the logs. You can add or remove messages from the logs as follows:

- To include audit event fields in logs that are not safelisted, configure the `includeIf` property of AuditService.

  > **Important**
  >
  > Before you include non-safelisted audit event fields in the logs, consider the impact on security. Including some headers, query parameters, or cookies in the logs could cause credentials or tokens to be logged, and allow anyone with access to the logs to impersonate the holder of these credentials or tokens.

- To exclude safelisted audit event fields from the logs, configure the `excludeIf` property of AuditService. For an example, refer to Exclude safelisted audit event fields from logs.

Exclude safelisted audit event fields from logs

Before you start, set up and test the example in Recording access audit events in JSON. Note the audit event fields in the log file `access.audit.json`.

1. Replace `30-json.json` with the following route:

Linux

```
$HOME/.openig/config/routes/30-json-excludeif.json
```

Windows

```
%appdata%\OpenIG\config\routes\30-json-excludeif.json
```

```
{
  "name": "30-json-excludeif",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/json-audit-excludeif$')}",
  "heap": [
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "config": {
          "filterPolicies": {
            "field": {
              "excludeIf": [
                "/access/http/request/headers/host",
                "/access/http/request/path",
                "/access/server",
                "/access/response"
              ]
            }
          }
        },
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
            "config": {
              "name": "json",
              "logDirectory": "/tmp/logs",
              "topics": [
                "access"
              ],
              "rotationRetentionCheckInterval": "1 minute",
              "buffering": {
                "maxSize": 100000,
                "writeInterval": "100 ms"
              }
            }
          }
        ]
      }
    }
  ],
  "auditService": "AuditService",
  "handler": "ReverseProxyHandler"
}
```

Notice that the AuditService is configured with an `excludeIf` property to exclude audit event fields from the logs.

2. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/json-audit-excludeif.

   The home page of the sample application is displayed and the file `/tmp/logs/access.audit.json` is updated:

```
{
  "_id": "830...-41",
  "timestamp": "2019-...540Z",
  "eventName": "OPENIG-HTTP-ACCESS",
  "transactionId": "830...-40",
  "client": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 51666
  },
  "http": {
    "request": {
      "secure": false,
      "method": "GET",
      "headers": {
        "accept": ["text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"],
        "user-agent": ["Mozilla/5.0 ... Firefox/66.0"]
      }
    }
  },
  "ig": {
    "exchangeId": "b3f...-56",
    "routeId": "30-json-excludeif",
    "routeName": "30-json-excludeif"
  }
}
```


3. Compare the audit event fields in `access.audit.json` with those produced in Recording access audit events in JSON, and note that the audit event fields specified by the `excludeIf` property no longer appear in the logs.

Record user ID in audit events

The following sections provide examples of how to capture the AM user ID in audit logs.

Sample scripts are available in the `openig-samples.jar` file, to capture the user ID after SSO, CDSSO, OpenID, or SAML authentication. The scripts inject the user ID into the RequestAuditContext so that it is available when the audit event is written.

Using the notes in the sample scripts, adapt the script for your deployment. For example, configure which `user_info` field to capture in the audit event.

The audit service in these examples use a JsonStdoutAuditEventHandler, which writes audit events to standard output, but can be any other audit service.

Record user ID in audit logs after SSO authentication

Before you start, set up and test the example in Authenticating with SSO.

1. Add the following script to IG:

Linux

```
$HOME/.openig/scripts/groovy/InjectUserIdSso.groovy
```

Windows

```
%appdata%\OpenIG\scripts\groovy\InjectUserIdSso.groovy
```

```
package scripts.groovy

import org.forgerock.openig.openam.SsoTokenContext
import org.forgerock.services.context.RequestAuditContext

/**
 * Sample ScriptableFilter implementation to capture the user id from the session
 * and inject it into the RequestAuditContext for later use when the audit event
 * is written.
 *
 * This ScriptableFilter should be added in the filter chain at whatever point the
 * desired user id is available - e.g. on the session after SSO.
 *
 * "handler": {
 *   "type": "Chain",
 *   "config": {
 *     "filters": [ {
 *        "name": "SingleSignOnFilter-1",
 *         "type": "SingleSignOnFilter",
 *         "config": {
 *           "amService": "AmService-1"
 *         }
 *       }, {
 *         "type" : "ScriptableFilter",
 *         "config" : {
 *           "file" : "InjectUserIdSso.groovy",
 *           "type": "application/x-groovy"
 *         }
 *       }
 *     ],
 *     "handler" : "ReverseProxyHandler",
 * }
 *
 * When using the SSO/ CDSSO flow then the SsoTokenContext is guaranteed to exist and
 * be populated if there was no error. The RequestAuditContext is also guaranteed to
 * be available. Note also that if the SessionInfoFilter is present in the route then
 * a SessionInfoContext would be available in the context chain and could be queried
 * for user info.
 *
 * Implementors may decide which user id field to capture in the audit event:
 * - The sessionInfo universalId - 'universalId' - is always available as
 *   provided by AM and resembles -
 *   e.g. "id=bonnie,ou=user,o=myrealm,ou=services,dc=openam,dc=forgerock,dc=org".
 * - The sessionInfo username - mapped to 'username') resembles - e.g. "bonnie".
 *   Field 'username' should be preferred to 'uid', which also points to 'username'.
 *
 * Additional error handling may be required.
 *
 * @see RequestAuditContext
 * @see SsoTokenContext
 * @see org.forgerock.openig.openam.SessionInfoContext
 */

def requestAuditContext = context.asContext(RequestAuditContext.class)
def ssoTokenContext = context.asContext(SsoTokenContext.class)
```

```
// The sessionInfo 'universalId' is always available, though 'username' may be unknown
requestAuditContext.setUserId(ssoTokenContext.universalId)

// Propagate the request to the next filter/ handler in the chain
next.handle(context, request)
```

The script captures the user ID after SSO or CDSSO authentication, and injects it into the RequestAuditContext so that it is available when the audit event is written.

2. Replace `sso.json` with the following route:

Linux

```
$HOME/.openig/config/routes/audit-sso.json
```

Windows

```
%appdata%\OpenIG\config\routes\audit-sso.json
```

```
{
  "name": "audit-sso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/audit-sso$')}",
  "heap": [
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.json.stdout.JsonStdoutAuditEventHandler",
            "config": {
              "name": "jsonstdout",
              "elasticsearchCompatible": false,
              "topics": [
                "access"
              ]
            }
          }
        ],
        "config": {}
      }
    },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "auditService": "AuditService",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "type" : "ScriptableFilter",
          "config" : {
            "file" : "InjectUserIdSso.groovy",
            "type": "application/x-groovy"
          }
        }
```

```
        ],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

Notice the following features of the route compared to `sso.json`:

- The route matches requests to `/home/audit-sso`.
- An audit service is included to publish access log messages to standard output.
- The chain includes a scriptable filter that refers to `InjectUserIdSso.groovy`.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/audit-sso. The SingleSignOnFilter redirects the request to AM for authentication.
   2. Log in to AM as user `demo`, password `Ch4ng31t`, and then allow the application to access user information.

      The profile page of the sample application is displayed. The script captures the user ID from the session, and the audit service includes it with the audit event.
   3. Search the standard output for a message like this, containing the user ID:

```
{
  "_id": "23a...-23",
  "timestamp": "...",
  "eventName": "OPENIG-HTTP-ACCESS",
  "transactionId": "23a...-22",
  "userId": "id=demo,ou=user,dc=openam,dc=forgerock,dc=org",
  "client": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 57843
  },
  "server": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 8080
  },
  "http": {
    "request": {
      "secure": false,
      "method": "GET",
      "path": "http://ig.example.com/home/audit-sso",
      "headers": {
        "accept": ["text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,/;q=0.8"],
        "host": ["ig.example.com:8080"],
        "user-agent": [...]
      }
    }
  },
  "response": {
    "status": "SUCCESSFUL",
    "statusCode": "200",
    "elapsedTime": 276,
    "elapsedTimeUnits": "MILLISECONDS"
  },
  "ig": {
    "exchangeId": "1dc...-26",
    "routeId": "audit-sso",
    "routeName": "audit-sso"
  },
  "source": "audit",
  "topic": "access",
  "level": "INFO"
}
```


Record user ID in audit logs after OpenID connect authentication

Before you start, set up and test the example in AM as a single OpenID Connect provider.

1. Set up the script:
    1. Add the following example script to IG:

Linux

```
$HOME/.openig/scripts/groovy/InjectUserIdOpenId.groovy
```

Windows

```
%appdata%\OpenIG\scripts\groovy\InjectUserIdOpenId.groovy
```

```
package scripts.groovy

import org.forgerock.services.context.AttributesContext
import org.forgerock.services.context.RequestAuditContext

/**
 * Sample script implementation supporting user id injection in an OpenId scenario.
 * This sample captures the user id and injects it into the RequestAuditContext for
 * later use when the audit event is written.
 *
 * This ScriptableFilter should be added in the filter chain at whatever point the
 * desired user id is available - e.g. after OpenId client authentication (in the
 * OAuth2 authentication filter chain) - as follows:
 *
 * "handler" : {
 *   "type" : "Chain",
 *   "config" : {
 *     "filters" : [ {
 *       "type" : "OAuth2ClientFilter",
 *       "config" : {
 *         ...
 *         "target" : "${attributes.target}",
 *         "registrations" : [ "ClientRegistrationWithOpenIdScope" ],
 *       }
 *     }, {
 *       "type" : "ScriptableFilter",
 *       "config" : {
 *         "file" : "InjectUserIdOpenId.groovy",
 *         "type": "application/x-groovy"
 *       }
 *     } ],
 *     "handler" : "display-user-info-groovy-handler"
 *   }
 * }
 *
 * The ClientRegistration associated with the above OAuth2ClientFilter config will
 * require the 'openid' scope. The OAuth2SessionContext is guaranteed to exist and
 * be populated on successful authentication. The userinfo will then be populated
 * according to the OAuth2ClientFilter OpenId 'target' configuration (e.g. in this
 * sample, on the AttributesContext). The 'target' referenced will be populated
 * with a 'user_info' JSON value containing the userinfo. It should be noted that
 * the OAuth2ClientFilter 'target' config is a config-time expression, and cannot
 * be used in a ScriptableFilter to read runtime data. The RequestAuditContext is
 * also guaranteed to be available.
 *
 * Implementors may decide which 'user_info' field to capture in the audit event:
 * - The userinfo 'sub' field is the user's "complex" ID marked with a type - e.g.
 *   "(usr!bonnie)".
 * - The userinfo 'subName' field is the user's username (or resource name) - e.g.
 *   "bonnie".
 * - To capture the universalId (consistent with the session info universalId),
 *   it is necessary to configure AM to provide it as a claim in the id-token. To
 *   do this, edit the OIDC Claims Script to include the following line just prior
 *   to the UserInfoClaims creation:
 *       computedClaims["universalId"] = identity.universalId
 * - This will include 'universalId' in the userinfo which we can use with audit
 *   e.g. "id=bonnie,ou=user,o=myrealm,ou=services,dc=openam,dc=forgerock,dc=org"
 *
 * Additional error handling may be required.
 *
```

```
 * @see RequestAuditContext
 * @see AttributesContext
 */

def requestAuditContext = context.asContext(RequestAuditContext.class)
def attributesContext = context.asContext(AttributesContext.class)

// The OAuth2ClientFilter captures userinfo based on its 'target' configuration.
// In this sample 'target' is configured as the AttributesContext with key "target".
// We can query this for 'user_info' values: 'sub', 'subName' or anything else
// made available via the OIDC Claims Script (see above).
def oauth2UserInfo = attributesContext.getAttributes().get("target")
requestAuditContext.setUserId(oauth2UserInfo.get("user_info").get("sub"))

// Propagate the request to the next filter/ handler in the chain
next.handle(context, request)
```

The script captures the user ID from the AuthorizationCodeOAuth2ClientFilter `target` object, by default at `${attributes.openid}`, and injects it into the RequestAuditContext so that it is available when the audit event is written.

2. Edit the script to get the attributes from the `openid` target:

   Replace `attributesContext.getAttributes().get("target")`

   with `attributesContext.getAttributes().get("openid")`.

2. Replace `07-openid.json` with the following route:

Linux

```
$HOME/.openig/config/routes/audit-oidc.json
```

Windows

```
%appdata%\OpenIG\config\routes\audit-oidc.json
```

```
{
  "name": "audit-oidc",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/id_token')}",
  "heap": [
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.json.stdout.JsonStdoutAuditEventHandler",
            "config": {
              "name": "jsonstdout",
              "elasticsearchCompatible": false,
              "topics": [
                "access"
              ]
            }
          }
        ],
        "config": {}
      }
    },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    }
  ],
  "auditService": "AuditService",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "AuthorizationCodeOAuth2ClientFilter-1",
          "type": "AuthorizationCodeOAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/home/id_token",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "status": 500,
                "headers": {
                  "Content-Type": [
                    "text/plain"
                  ]
                },
                "entity": "Error in OAuth 2.0 setup."
              }
            },
            "registrations": [
              {
                "name": "oidc-user-info-client",
                "type": "ClientRegistration",
                "config": {
                  "clientId": "oidc_client",
                  "clientSecretId": "oidc.secret.id",
                  "issuer": {
                    "name": "Issuer",
```

```
                    "type": "Issuer",
                    "config": {
                      "wellKnownEndpoint": "http://am.example.com:8088/openam/oauth2/.well-known/openid-
configuration"
                    }
                  },
                  "scopes": [
                    "openid",
                    "profile",
                    "email"
                  ],
                  "secretsProvider": "SystemAndEnvSecretStore-1",
                  "tokenEndpointAuthMethod": "client_secret_basic"
                }
              }
            ],
            "requireHttps": false,
            "cacheExpiration": "disabled"
          }
        },
        {
          "type" : "ScriptableFilter",
          "config" : {
            "file" : "InjectUserIdOpenId.groovy",
            "type": "application/x-groovy"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Notice the following features of the route compared to `07-openid.json`:

- An audit service is included to publish access log messages to standard output.
- The chain includes a scriptable filter that refers to `InjectUserIdOpenId.groovy`.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to https://ig.example.com:8443/home/id_token. The AM login page is displayed.
   2. Log in to AM as user `demo`, password `Ch4ng31t`, and then allow the application to access user information.

      The home page of the sample application is displayed. The script captures the user ID from the `openid` target, and the audit service includes it with the audit event.
   3. Search the standard output for a message like this, containing the user ID:

```
{
  "_id": "b64...-25",
  "timestamp": "2021...",
  "eventName": "OPENIG-HTTP-ACCESS",
  "transactionId": "b64...-24",
  "userId": "(usr!demo)",
  "client": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 64443
  },
  "server": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 8080
  },
  "http": {
    "request": {
      "secure": false,
      "method": "GET",
      "path": "http://ig.example.com:8080/home/id_token",
      "headers": {
        "accept": ["text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,/;q=0.8"],
        "host": ["ig.example.com:8080"],
        "user-agent": [...]
      }
    }
  },
  "response": {
    "status": "SUCCESSFUL",
    "statusCode": "200",
    "elapsedTime": 199,
    "elapsedTimeUnits": "MILLISECONDS"
  },
  "ig": {
    "exchangeId": "1dc...-26",
    "routeId": "audit-oidc",
    "routeName": "audit-oidc"
  },
  "source": "audit",
  "topic": "access",
  "level": "INFO"
}
```


Record user ID in audit logs after SAML authentication

> **Important**
>
> This example uses the deprecated SamlFederationHandler. The SamlFederationHandler is replaced by the SamlFederationFilter and will be removed in a future release.

Before you start, set up and test the example in SAML.

1. Set up the script:
    1. Add the following example script to IG:

Linux

```
$HOME/.openig/scripts/groovy/InjectUserIdSaml.groovy
```

Windows

```
%appdata%\OpenIG\scripts\groovy\InjectUserIdSaml.groovy
```

```
package scripts.groovy

import org.forgerock.http.session.SessionContext
import org.forgerock.services.context.RequestAuditContext

/**
 * Sample ScriptableFilter implementation to capture the user id obtained from a
 * SAML assertion. The IG SamlFederationHandler captures this and locates it on
 * the SessionContext with the key as the configured SAML 2 user id key. We then
 * take this and inject it into the RequestAuditContext for later use when the
 * audit event is written.
 *
 * This ScriptableFilter should be added in the filter chain together with the
 * SamlFederationHandler, as follows. Note that the InjectUserIdSaml.groovy script
 * operates on the response, injecting the userId as captured by the handler.
 *
 * {
 *     "condition" : "${matches(request.uri.path,'^/api/saml')}",
 *     "handler" : {
 *         "type" : "Chain",
 *         "config" : {
 *             "filters" : [ {
 *                 "type" : "ScriptableFilter",
 *                 "config" : {
 *                     "file" : "InjectUserIdSaml.groovy",
 *                     "type": "application/x-groovy"
 *                 }
 *             } ],
 *             "handler" : {
 *                 "name" : "saml_handler_SPOne",
 *                 "type" : "SamlFederationHandler",
 *                 "config" : {
 *                      "assertionMapping" : {
 *                          "SPOne_userName" : "uid",
 *                          "SPOne_password" : "mail"
 *                      },
 *                      "redirectURI" : "/api/home",
 *                      "logoutURI" : "http://openig.example.com:8082/api/after_logout",
 *                      "subjectMapping" : "SubjectName_SPOne",
 *                      "authnContext" : "AuthnContext_SPOne",
 *                      "sessionIndexMapping" : "SessionIndex_SPOne"
 *                 }
 *             }
 *         }
 *     }
 * }
 *
 * The SessionContext and RequestAuditContext are guaranteed to be available and the
 * SessionContext will have been populated with userinfo on successful authentication.
 *
 * Implementors may decide which user id field to capture in the audit event:
 * - This should be based on SAML attribute mappings and/ or the subject mapping (if
 *   transient names are not used).
 * - Other attributes are available, such as 'uid' and 'userName', though  it must be
 *   noted that there is an expectation that the IDP makes available the user id.
 * - In this sample, 'SPOne_userName' maps to the 'uid'.
 *
 * Additional error handling may be required.
 *
 * @see RequestAuditContext
```

```
 * @see SessionContext
 */

// Propagate the request to the next filter/ handler in the chain
next.handle(context, request)
    .then({ response ->
        def requestAuditContext = context.asContext(RequestAuditContext.class)
        def sessionContext = context.asContext(SessionContext.class)

        // Inject the user id as captured by the SamlFederationHandler
        requestAuditContext.setUserId(sessionContext.getSession().get("SPOne_userName"))
        return response
    })
```

The script captures the user ID from the SessionContext subject or attribute mappings, provided by the SamlFederationHandler from the inbound assertions. It injects the user ID into the RequestAuditContext so that it is available when the audit event is written.

2. Replace `get("SPOne_userName"))` with `get("username"))`.

   The script captures the user ID from the assertionMapping `username`, which is mapped in the route to `cn`.

2. Replace `saml.json` with the following route:

Linux

```
$HOME/.openig/config/routes/audit-saml.json
```

Windows

```
%appdata%\OpenIG\config\routes\audit-saml.json
```

```
{
  "name": "audit-saml",
  "condition": "${find(request.uri.path, '^/saml')}",
  "session": "JwtSession",
  "heap": [
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.json.stdout.JsonStdoutAuditEventHandler",
            "config": {
              "name": "jsonstdout",
              "elasticsearchCompatible": false,
              "topics": [
                "access"
              ]
            }
          }
        ],
        "config": {}
      }
    }
  ],
  "auditService": "AuditService",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type" : "ScriptableFilter",
          "config" : {
            "file" : "InjectUserIdSaml.groovy",
            "type": "application/x-groovy"
          }
        }
      ],
      "handler": {
        "type": "SamlFederationHandler",
        "config": {
          "useOriginalUri": true,
          "assertionMapping": {
            "username": "cn",
            "password": "sn"
          },
          "subjectMapping": "sp-subject-name",
          "redirectURI": "/home/federate"
        }
      }
    }
  }
}
```

Notice the following features of the route compared to `saml.json`:

- An audit service is included to publish access log messages to standard output.
- The main Handler is a Chain, that includes a scriptable filter to refer to `InjectUserIdSaml.groovy`.

- The script uses the assertionMapping `username` to capture the user ID.

3. Test the setup:

   1. In your browser's privacy or incognito mode, go to IDP-initiated SSO.

   2. Log in to AM with username `demo` and password `Ch4ng31t`.

      IG returns the response page showing that the the demo user has logged in. The script captures the user ID from the session, and the audit service includes it with the audit event.

   3. Search the standard output for a message like this, containing the user ID:


```
{
  "_id": "82f...-14",
  "timestamp": "2021-...",
  "eventName": "OPENIG-HTTP-ACCESS",
  "transactionId": "82f...-13",
  "userId": "demo",
  "client": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 60655
  },
  "server": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 8080
  },
  "http": {
    "request": {
      "secure": false,
      "method": "POST",
      "path": "http://sp.example.com:8080/saml/fedletapplication/metaAlias/sp",
      "headers": {
        "accept": ["text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,/;q=0.8"],
        "content-type": ["application/x-www-form-urlencoded"],
        "host": ["sp.example.com:8080"],
        "user-agent": [...]
      }
    }
  },
  "response": {
    "status": "SUCCESSFUL",
    "statusCode": "302",
    "elapsedTime": 2112,
    "elapsedTimeUnits": "MILLISECONDS"
  },
  "ig": {
    "exchangeId": "1dc...-26",
    "routeId": "audit-saml",
    "routeName": "audit-saml"
  },
  "source": "audit",
  "topic": "access",
  "level": "INFO"
}
```

Monitor services

The following sections describe how to set up and maintain monitoring in your deployment, to ensure appropriate performance and service availability:

Access the monitoring endpoints

All ForgeRock products automatically expose a monitoring endpoint to expose metrics in a standard Prometheus format, and as a JSON format monitoring resource.

In IG, metrics are available for each router, subrouter, and route in the configuration. When a TimerDecorator is configured, timer metrics are also available.

Learn more about IG monitoring endpoints and available metrics in Monitoring.

Monitor at the Prometheus Scrape Endpoint

> **Note**
>
> Prometheus metric names are deprecated and expected to be replaced with names ending in _total. The information provided by the metric is not deprecated. Other Prometheus metrics are not affected.

All ForgeRock products automatically expose a monitoring endpoint where Prometheus can scrape metrics, in a standard Prometheus format.

When IG is set up as described in the Quick install, the Prometheus Scrape Endpoint is available at http://ig.example.com:8080/openig/metrics/prometheus.

By default, no special setup or configuration is required to access metrics at this endpoint. The following example queries the Prometheus Scrape Endpoint for a route.

Tools such as Grafana are available to create customized charts and graphs based on the information collected by Prometheus. For more information on installing and running Grafana, refer to the Grafana website.

1. Set up IG:
    1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).
    2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/myroute1.json
```

Windows

```
%appdata%\OpenIG\config\routes\myroute1.json
```

```
{
  "name": "myroute1",
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/plain; charset=UTF-8" ]
      },
      "entity": "Hello world, from myroute1!"
    }
  },
  "condition": "${find(request.uri.path, '^/myroute1')}"
}
```

The route contains a StaticResponseHandler to display a simple message.

2. Test the setup:

    1. Access the route a few times, on https://ig.example.com:8443/myroute1.

    2. Query the Prometheus Scrape Endpoint:

    ```
    $ curl "https://ig.example.com:8443/openig/metrics/prometheus"
    ```

    Metrics for `myroute1` and `_router` are displayed:

```
# HELP ig_router_deployed_routes Generated from Dropwizard metric import
(metric=gateway._router.deployed-routes, type=gauge)
# TYPE ig_router_deployed_routes gauge
ig_router_deployed_routes{fully_qualified_name="gateway._router",heap="gateway",name="_router",} 1.0
# HELP ig_route_request_active Generated from Dropwizard metric import
(metric=gateway._router.route.default.request.active, type=gauge)
# TYPE ig_route_request_active gauge
ig_route_request_active{name="default",route="default",router="gateway._router",} 0.0
# HELP ig_route_request_active Generated from Dropwizard metric import
(metric=gateway._router.route.myroute1.request.active, type=gauge)
# TYPE ig_route_request_active gauge
ig_route_request_active{name="myroute1",route="myroute1",router="gateway._router",} 0.0
# HELP ig_route_request_total Generated from Dropwizard metric import
(metric=gateway._router.route.default.request, type=counter)
# TYPE ig_route_request_total counter
ig_route_request_total{name="default",route="default",router="gateway._router",} 0.0
# HELP ig_route_response_error Generated from Dropwizard metric import
(metric=gateway._router.route.default.response.error, type=counter)
# TYPE ig_route_response_error counter
ig_route_response_error{name="default",route="default",router="gateway._router",} 0.0
# HELP ig_route_response_null Generated from Dropwizard metric import
(metric=gateway._router.route.default.response.null, type=counter)
# TYPE ig_route_response_null counter
ig_route_response_null{name="default",route="default",router="gateway._router",} 0.0
# HELP ig_route_response_status_total Generated from Dropwizard metric import
(metric=gateway._router.route.default.response.status.client_error, type=counter)
# TYPE ig_route_response_status_total counter
ig_route_response_status_total{family="client_error",name="default",route="default",router="gateway._ro
uter",} 0.0
...
```

Vert.x monitoring is enabled by default to provide additional metrics for HTTP, TCP, and the internal component pool. The metrics provide low-level information about requests and responses, such as the number of bytes, duration, the number of concurrent requests, and so on.

Monitor the Common REST Monitoring Endpoint (deprecated)

All ForgeRock products expose a monitoring endpoint where metrics are exposed as a JSON format monitoring resource.

When IG is set up as described in Quick install, the Common REST Monitoring Endpoint is available at https://ig.example.com:8443/openig/metrics/api?_prettyPrint=true&_sortKeys=_id&_queryFilter=true

By default, no special setup or configuration is required to access metrics at this endpoint. The following example queries the Common REST Monitoring Endpoint for a route, and restricts the query to specific metrics only.

Before you start, prepare IG as described in the Quick install.

1. Set up IG and some example routes, as described in the first few steps of Monitor the Prometheus Scrape Endpoint.
2. Query the Common REST Monitoring Endpoint:

```
$ curl "https://ig.example.com:8443/openig/metrics/api?_prettyPrint=true&_sortKeys=_id&_queryFilter=true"
```

Metrics for `myroute1` and `_router` are displayed:

```
{
  "result" : [ {
  "_id" : "gateway._router.deployed-routes",
  "value" : 1.0,
  "_type" : "gauge"
}, {
  "_id" : "gateway._router.route.default.request",
  "count" : 204,
  "_type" : "counter"
}, {
  "_id" : "gateway._router.route.default.request.active",
  "value" : 0.0,
  "_type" : "gauge"
}, {

        . . .

       _id" : "gateway._router.route.myroute1.response.status.unknown",
  "count" : 0,
  "_type" : "counter"
}, {
  "_id" : "gateway._router.route.myroute1.response.time",
  "count" : 204,
  "max" : 0.420135,
  "mean" : 0.08624678327176545,
  "min" : 0.045079999999999995,
  "p50" : 0.070241,
  "p75" : 0.096049,
  "p95" : 0.178534,
  "p98" : 0.227217,
  "p99" : 0.242554,
  "p999" : 0.420135,
  "stddev" : 0.046611762381930474,
  "m15_rate" : 0.2004491450567003,
  "m1_rate" : 2.8726563452698075,
  "m5_rate" : 0.5974045160056258,
  "mean_rate" : 0.010877725092634833,
  "duration_units" : "milliseconds",
  "rate_units" : "calls/second",
  "total" : 17.721825,
  "_type" : "timer"
} ],
  "resultCount" : 11,
  "pagedResultsCookie" : null,
  "totalPagedResultsPolicy" : "EXACT",
  "totalPagedResults" : 11,
  "remainingPagedResults" : -1
}
```


Vert.x monitoring is enabled by default to provide additional metrics for HTTP, TCP, and the internal component pool. The metrics provide low-level information about requests and responses, such as the number of bytes, duration, the number of concurrent requests, and so on.

3. Change the query to access metrics only for `myroute1`: https://ig.example.com:8443/openig/metrics/api?_prettyPrint=true&_sortKeys=_id&_queryFilter=_id+sw+"gateway._router.route.myroute1".

   Note that metric for the router, `"_id" : "gateway._router.deployed-routes"`, is no longer displayed.

Monitor Vert.x metrics

> **Note**
>
> Vert.x metric names are deprecated and expected to be replaced with names ending in _total. The information provided by the metric is not deprecated. Other Prometheus metrics are not affected.

Vert.x monitoring is enabled by default to provide metrics for HTTP, TCP, and the internal component pool. The metrics provide low-level information about requests and responses, such as the number of bytes, duration, the number of concurrent requests, and so on.

To disable Vert.x monitoring, add the following lines to `admin.json`, and restart IG:

```
{
  "vertx": {
    "metricsEnabled": false
  }
}
```

For more information, refer to AdminHttpApplication (admin.json).

Protect monitoring endpoints

By default, no special credentials or privileges are required for read-access to the Prometheus Scrape Endpoint and Common REST Monitoring Endpoint.

To protect the monitoring endpoints, add an `admin.json` file to your configuration, with a filter declared in the heap and named `MetricsProtectionFilter`. The following procedure gives an example of how to manage access to the monitoring endpoints.

1. Set up the procedure in Monitor at the Prometheus Scrape Endpoint, query the Prometheus Scrape Endpoint, and note that metrics for `myroute1` and `_router` are displayed:

   ```
   $ curl -v "https://ig.example.com:8443/openig/metrics/prometheus"
   ```

2. Add the following script to the IG configuration:

   Linux

   ```
   $HOME/.openig/scripts/groovy/BasicAuthResourceServerFilter.groovy
   ```

Windows

```
%appdata%\OpenIG\scripts\groovy\BasicAuthResourceServerFilter.groovy
```

```
/*
 * This script is a simple implementation of HTTP basic access authentication on
 * server side.
 * It expects the following arguments:
 *  - realm: the realm to display when the user agent prompts for
 *    username and password if none were provided.
 *  - username: the expected username
 *  - passwordSecretId: the secretId to find the password
 *  - secretsProvider: the SecretsProvider to query for the password
*/
import static org.forgerock.util.promise.Promises.newResultPromise;

import java.nio.charset.Charset;
import org.forgerock.util.encode.Base64;
import org.forgerock.secrets.Purpose;
import org.forgerock.secrets.GenericSecret;

String authorizationHeader = request.getHeaders().getFirst("Authorization");
if (authorizationHeader == null) {
    // No credentials provided, return 401 Unauthorized
    Response response = new Response(Status.UNAUTHORIZED);
    response.getHeaders().put("WWW-Authenticate", "Basic realm=\"" + realm + "\"");
    return newResultPromise(response);
}

return secretsProvider.getNamed(Purpose.PASSWORD, passwordSecretId)
        .thenAsync(password -> {
            // Build basic authentication string -> username:password
            StringBuilder basicAuthString = new StringBuilder(username).append(":");
            password.revealAsUtf8{ p -> basicAuthString.append(new String(p).trim()) };
            String expectedAuthorization = "Basic " +
Base64.encode(basicAuthString.toString().getBytes(Charset.defaultCharset()));
            // Incorrect credentials provided, return 403 forbidden
            if (!expectedAuthorization.equals(authorizationHeader)) {
                return newResultPromise(new Response(Status.FORBIDDEN));
            }
            // Correct credentials provided, continue.
            return next.handle(context, request);
        },
                noSuchSecretException -> { throw new RuntimeException(noSuchSecretException); });
```

The script is a simple implementation of the HTTP basic access authentication scheme. For information about scripting filters and handlers, refer to Extend.

3. Add the following heap configuration for MetricsProtectionFilter to `admin.json`:

```
{
  ...,
  "heap": [
    {
      "name": "ClientHandler",
      "type": "ClientHandler"
    },
    {
      "name": "mySecretsProvider",
      "type": "Base64EncodedSecretStore",
      "config": {
        "secrets": {
          "password.secret.id": "cGFzc3dvcmQ="
        }
      }
    },
    {
      "name": "MetricsProtectionFilter",
      "type": "ScriptableFilter",
      "config": {
        "type": "application/x-groovy",
        "file": "BasicAuthResourceServerFilter.groovy",
        "args": {
          "realm": "/",
          "username": "myUsername",
          "passwordSecretId": "password.secret.id",
          "secretsProvider": "${heap['mySecretsProvider']}"
        }
      }
    }
  ],
  ...
}
```

Notice the following features of the configuration:

- The MetricsProtectionFilter uses the script to protect the monitoring endpoint.
- The MetricsProtectionFilter requires the username `myUsername`, and a password provided by the SecretsProvider in the heap.

4. Restart IG.
5. Query the Prometheus Scrape Endpoint without providing credentials, and note that an HTTP 401 Unauthorized is returned:

```
$ curl -v "https://ig.example.com:8443/openig/metrics/prometheus"
```

6. Query the Prometheus Scrape Endpoint by providing correct credentials, and note that metrics are displayed:

```
$ curl -v "https://ig.example.com:8443/openig/metrics/prometheus" -u myUsername:password
```

7. Query the Prometheus Scrape Endpoint by providing incorrect credentials`, and note that an HTTP 403 Forbidden is returned:

```
$ curl -v "https://ig.example.com:8443/openig/metrics/prometheus" -u myUsername:wrong-password
```

Manage sessions

For information about IG sessions, refer to Sessions. Change IG session properties in the following ways:

Mode	To change the session properties
Stateless sessions	Configure the JwtSession object in the route that processes a request, or in its ascending configuration. For example, define the `cookie` property to configure the IG session name. `{ "name": "JwtSession", "type": "JwtSession", "config": { "cookie": { "name": "MY_SESSIONID" } } }`
Stateful sessions	Change the `session` property in `admin.json`, and restart IG. For example, add the following lines to `admin.json` to configure the IG session name: `"session": { "cookie": { "name": "MY_SESSIONID" } }`

Manage logs

Log messages in IG and third-party dependencies are recorded using the Logback implementation of the Simple Logging Facade for Java (SLF4J) API. The following log levels are supported: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `ALL`, and `OFF`. For a full description of the options for logging, refer to the Logback website.

Default logging behavior

By default, log messages are recorded with the following configuration:

- When IG starts, log messages for IG and third-party dependencies, such as the ForgeRock Common Audit framework, are displayed on the console and written to `$HOME/.openig/logs/route-system.log`, where `$HOME/.openig` is the instance directory.
- When a capture point for the default CaptureDecorator is defined in a route, for example, when `"capture: "all"` is set as a top-level attribute of the JSON, log messages for requests and responses passing through the route are written to a log file in `$HOME/.openig/logs`.

  When no capture point is defined in a route, only exceptions thrown during request or response processing are logged.

  For more information, refer to Capturing log messages for routes and CaptureDecorator.
- By default, log messages with the level `INFO` or higher are recorded, with the titles and the top line of the stack trace. Messages on the console are highlighted with a color related to their log level.

The content and format of logs in IG is defined by the reference `logback.xml` delivered with IG. This file defines the following configuration items for logs:

- A root logger to set the overall log level, and to write all log messages to the `SIFT` and `STDOUT` appenders.
- A `STDOUT` appender to define the format of log messages on the console.
- A `SIFT` appender to separate log messages according to the key `routeId`, to define when log files are rolled, and to define the format of log messages in the file.
- An exception logger, called `LogAttachedExceptionFilter`, to write log messages for exceptions attached to responses.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

  <!--
    Prevent log flow attacks, by limiting repeated log messages.

    Configuration properties:
     * AllowedRepetitions (int): Threshold above which repeated messages are no longer logged.
     * CacheSize (int): When CacheSize is reached, remove the oldest entry.
  -->
  <!--<turboFilter class="ch.qos.logback.classic.turbo.DuplicateMessageFilter" />-->

  <!-- Allow configuration of JUL loggers within this file, without performance impact -->
  <contextListener class="ch.qos.logback.classic.jul.LevelChangePropagator" />

  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <withJansi>true</withJansi>
    <encoder>
      <pattern>%nopex[%thread] %highlight(%-5level) %boldWhite(%logger{35}) @%mdc{routeId:-system} -
%replace(%message){'([\r\n])(.)', '$1[CONTINUED]$2'}%n%highlight(%replace(%rootException{short}){'(^|[\r\n])(.)',
'$1[CONTINUED]$2'})</pattern>
    </encoder>
  </appender>

  <appender name="SIFT" class="ch.qos.logback.classic.sift.SiftingAppender">
    <discriminator>
      <key>routeId</key>
      <defaultValue>system</defaultValue>
    </discriminator>
    <sift>
      <!-- Create a separate log file for each <key> -->
      <appender name="FILE-${routeId}" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${instance.dir}/logs/route-${routeId}.log</file>

        <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
          <!-- Rotate files daily -->
          <fileNamePattern>${instance.dir}/logs/route-${routeId}-%d{yyyy-MM-dd}.%i.log</fileNamePattern>

          <!-- each file should be at most 100MB, keep 30 days worth of history, but at most 3GB -->
          <maxFileSize>100MB</maxFileSize>
          <maxHistory>30</maxHistory>
          <totalSizeCap>3GB</totalSizeCap>
        </rollingPolicy>

        <encoder>
          <pattern>%nopex%date{"yyyy-MM-dd'T'HH:mm:ss,SSSXXX", UTC} | %-5level | %thread | %logger{20} |
@%mdc{routeId:-system} | %replace(%message%n%xException){'([\r\n])(.)', '$1[CONTINUED]$2'}</pattern>
        </encoder>
      </appender>
    </sift>
  </appender>

  <!-- Disable logs of exceptions attached to responses by defining 'level' to OFF -->
  <logger name="org.forgerock.openig.filter.LogAttachedExceptionFilter" level="INHERITED" />

  <root level="${ROOT_LOG_LEVEL:-INFO}">
    <appender-ref ref="SIFT" />
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

Using a custom Logback file

To change the logging behavior, create a new logback file at `$HOME/.openig/config/logback.xml`, and restart IG. The custom Logback file overrides the default configuration.

To take into account edits to `logback.xml`, stop and restart IG, or edit the `configuration` parameter to add a scan and an interval:

```
<configuration scan="true" scanPeriod="5 seconds">
```

The `logback.xml` file is scanned after both of the following criteria are met:

- The specified number of logging operations have occurred, where the default is 16.
- The `scanPeriod` has elapsed.

If the custom `logback.xml` contains errors, messages like these are displayed on the console but log messages are not recorded:

```
14:38:59,667 |-ERROR in ch.qos.logback.core.joran.spi.Interpreter@20:72 …
14:38:59,690 |-ERROR in ch.qos.logback.core.joran.action.AppenderRefAction …
```

Change the global log level

The global log level is set by default to `INFO` by the following line of the default `logback.xml`:

```
<root level="${ROOT_LOG_LEVEL:-INFO}">
```

The log level set in `logback.xml` supercedes the log level set by environment variables. When the global log level is not set in `logback.xml`, set the global log level.

- To persist the log level for all future IG instances:
  - Add an environment variable in `$HOME/.openig/bin/env.sh`, where `$HOME/.openig` is the instance directory:

    ```
    export ROOT_LOG_LEVEL=DEBUG
    ```

  - Alternatively, add a system property in `$HOME/.openig/bin/env.sh`, where `$HOME/.openig` is the instance directory:

    ```
    export JAVA_OPTS="-DROOT_LOG_LEVEL=DEBUG"
    ```

    If both an environment variable and system property is set, the system property takes precedence.
- To persist the log level for IG instances launched from the same shell, add an environment variable in the shell before you start IG:

Linux

```
$ export ROOT_LOG_LEVEL=DEBUG
$ /path/to/identity-gateway-2024.3.0/bin/start.sh $HOME/.openig
```

Windows

```
C:\set ROOT_LOG_LEVEL=DEBUG
C:\path\to\identity-gateway-2024.3.0\bin\start.bat %appdata%\OpenIG
```

- To persist the log level for a single IG instance:

Linux

```
$ export ROOT_LOG_LEVEL=DEBUG /path/to/identity-gateway-2024.3.0/bin/start.sh $HOME/.openig
```

Windows

```
C:\set ROOT_LOG_LEVEL=DEBUG
C:\path\to\identity-gateway-2024.3.0\bin\start.bat %appdata%\OpenIG
```

Change the log level for different object types

To change the log level for a single object type without changing it for the rest of the configuration, edit `logback.xml` to add a logger defined by the fully qualified class name or package name of the object, and set its log level.

The following line in `logback.xml` sets the ClientHandler log level to `ERROR`, but does not change the log level of other classes or packages:

```
<logger name="org.forgerock.openig.handler.ClientHandler" level="ERROR" />
```

To facilitate debugging, in `logback.xml` add loggers defined by the fully qualified package name or class name of the object. For example, add loggers for the following feature:

Feature	Logger
OAuth 2.0 client authentication: • AuthorizationCodeOAuth2ClientFilter • ClientCredentialsOAuth2ClientFilter • ResourceOwnerOAuth2ClientFilter	`org.forgerock.secrets.oauth2`
Expression resolution	`org.forgerock.openig.el` `org.forgerock.openig.resolver`
WebSocket notifications	`org.forgerock.openig.tools.notifications.ws`
Session management with JwtSession	`org.forgerock.openig.jwt`
OAuth 2.0 and OpenID Connect and token resolution and validation	`org.forgerock.openig.filter.oauth2`
AM policies, SSO, CDSSO, and user profiles	`org.forgerock.openig.openam` `org.forgerock.openig.tools`
SAML	`org.forgerock.openig.handler.saml`
UMA	`org.forgerock.openig.uma`
WebSocket tunnelling	`org.forgerock.openig.websocket`

Secret resolution	`org.forgerock.secrets.propertyresolver`
	`org.forgerock.secrets.jwkset`
	`org.forgerock.secrets.keystore`
	`org.forgerock.secrets.oauth2`
	`org.forgerock.openig.secrets.Base64EncodedSecretStore`

AllowOnlyFilter	`org.forgerock.openig.filter.allow.AllowOnlyFilter.<filter_name>`
Condition of a route	`org.forgerock.openig.handler.router.RouterHandler`
Header field size	`io.vertx.core.http.impl.HttpServerImpl`

Change the character set and format of log messages

By default, logs use the system default character set where IG is running.

Tip

If your logs might contain characters that are not in your system character set, edit `logback.xml` to change the `encoder` part of the SIFT appender.

The following lines add the date to log messages, and change the character set:

```
<encoder>
   <pattern>%d{yyyyMMdd-HH:mm:ss} | %-5level | %thread | %logger{20} | %message%n%xException</pattern>
   <charset>UTF-8</charset>
</encoder>
```

For more information about what information you can include in the logs, and its format, refer to PatternLayoutEncoder and Layouts in the Logback documentation.

Log in scripts

The logger object provides access to a unique SLF4J logger instance for scripts. Events are logged as defined in by a dedicated logger in `logback.xml`, and are included in the logs with the name of with the scriptable object.

To log events for scripts:

- Add logger objects to the script to enable logging at different levels. For example, add some of the following logger objects:

  ```
  logger.error("ERROR")
  logger.warn("WARN")
  logger.info("INFO")
  logger.debug("DEBUG")
  logger.trace("TRACE")
  ```

- Add a logger to `logback.xml` to reference the scriptable object and set the log level. The logger is defined by the type and name of the scriptable object that references the script, as follows:

  - ScriptableFilter: `org.forgerock.openig.filter.ScriptableFilter.filter_name`
  - ScriptableHandler: `org.forgerock.openig.handler.ScriptableHandler.handler_name`
  - ScriptableThrottlingPolicy: `org.forgerock.openig.filter.throttling.ScriptableThrottlingPolicy.throttling_policy_name`
  - ScriptableAccessTokenResolver: `org.forgerock.openig.filter.oauth2.ScriptableAccessTokenResolver.access_token_resolver_name`

For example, the following logger logs trace-level messages for a ScriptableFilter named `cors_filter`:

```
<logger name="org.forgerock.openig.filter.ScriptableFilter.cors_filter" level="TRACE" />
```

The resulting messages in the logs contain the name of the scriptable object:

```
14:54:38:307 | TRACE | http-nio-8080-exec-6 | o.f.o.f.S.cors_filter | TRACE
```

Log the BaseUriDecorator

During setup and configuration, it can be helpful to display log messages from the BaseUriDecorator. To record a log message each time a request URI is rebased , edit `logback.xml` to add a logger defined by the fully qualified class name of the BaseUriDecorator appended by the name of the baseURI decorator:

```
<logger name="org.forgerock.openig.decoration.baseuri.BaseUriDecorator.baseURI" level="TRACE" />
```

Each time a request URI is rebased, a log message similar to this is created:

```
12:27:40| TRACE | http-nio-8080-exec-3 | o.f.o.d.b.B.b.{Router}/handler| Rebasing request to http://app.example.com:
8081
```

Stop exception logging

To stop recording log messages for exceptions, edit `logback.xml` to set the level to `OFF`:

```
<logger name="org.forgerock.openig.filter.LogAttachedExceptionFilter" level="OFF" />
```

Capture the context or entity of messages for routes

To capture the context or entity of inbound and outbound messages for a route, or for an individual handler or filter in the route, configure a CaptureDecorator. Captured information is written to SLF4J logs.

> **Important**
>
> During debugging, consider using a CaptureDecorator to capture the entity and context of requests and responses. However, increased logging consumes resources, such as disk space, and can cause performance issues. In production, reduce logging by disabling the CaptureDecorator properties `captureEntity` and `captureContext`, or setting `maxEntityLength`.

For more information about the decorator configuration, refer to CaptureDecorator.

Studio provides an easy way to capture messages while developing your configuration. The following image illustrates the capture points where you can log messages on a route:

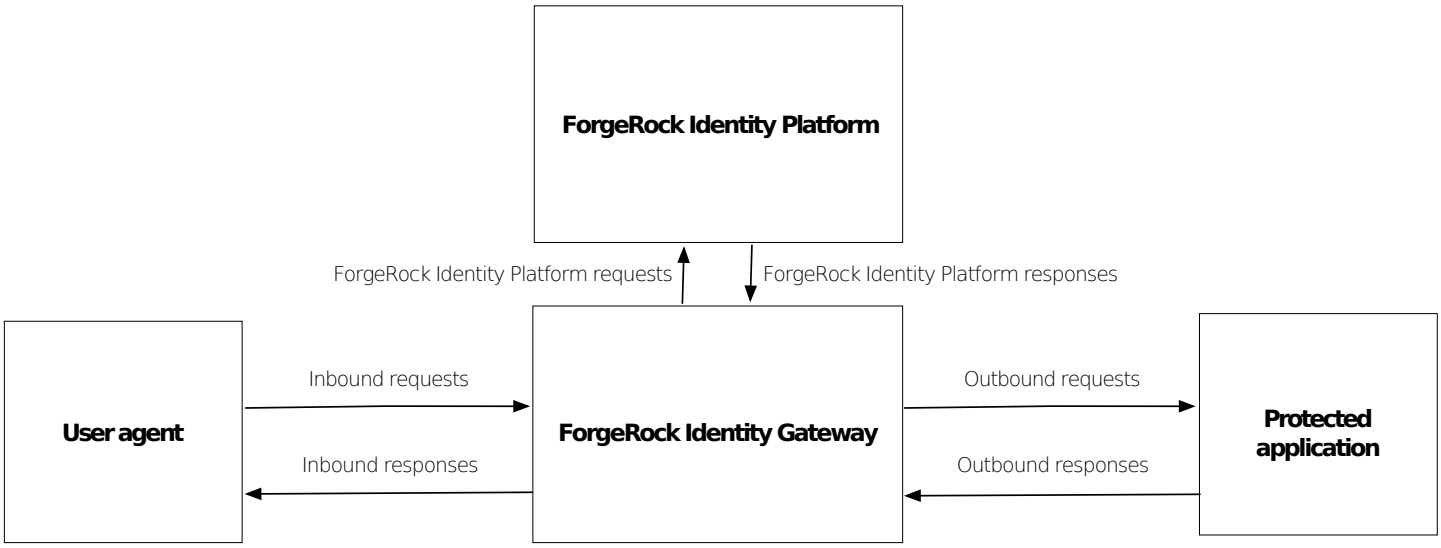


Figure 1. Capturing log messages for routes

Capture messages on a route in Studio

1. In Studio, select **ROUTES**, and then select a route with the icon.
2. On the left side of the screen, select **Capture**, and then select capture options. You can capture the body and context of messages passing to and from the user agent, the protected application, and the ForgeRock Identity Platform.
3. Select **Deploy** to push the route to the IG configuration.

   You can check the `$HOME/.openig/config/routes` folder to see that the route is there.
4. Access the route, and then check `$HOME/.openig/logs` for a log file named by the route, where `$HOME/.openig` is the instance directory. The log file should contain the messages defined by your capture configuration.

Limit repetitive log messages

To keep log files clean and readable, and to prevent log flow attacks, limit the number of repeat log messages. Add a custom `logback.xml` with a `DuplicateMessageFilter`. This filter detects duplicate messages, and after the specified number of repetitions, drops repeated messages.

The following example allows 5 repetitions of a log message, and holds the last 10 repeated messages in the cache:

```
<turboFilter class="ch.qos.logback.classic.turbo.DuplicateMessageFilter" allowedRepetitions="5" CacheSize="10" />
```

The DuplicateMessageFilter has the following limitations:

- Filters out **all** duplicate messages. It does not filter per logger, or logger instance, or logger name.
- Detects repetition of raw messages, meaning that the following example messages are considered as repetition:

  ```
  logger.debug("Hello {}.", name0);
  logger.debug("Hello {}.", name1);
  ```

- Doesn't limit the lifespan of the cache. After the specified number of repetitions is reached, the repeated log messages never appear again, even if they're frequently hit.

Tune performance

Tune deployments in the following steps:

1. Consider the issues that impact the performance of a deployment. See Define requirements and constraints.
2. Tune and test the downstream servers and applications:
    1. Tune the downstream web container and JVM to achieve performance targets.
    2. Test downstream servers and applications in a pre-production environment, under the expected load, and with common use cases.
3. Increase hardware resources as required, and then re-tune the deployment.

Define requirements and constraints

When you consider performance requirements, bear in mind the following points:

- The capabilities and limitations of downstream services or applications on your performance goals.
- The increase in response time due to the extra network hop and processing, when IG is inserted as a proxy in front of a service or application.
- The constraint that downstream limitations and response times place on IG.

Service level objectives

A service level objective (SLO) is a target that you can measure quantitatively. Where possible, define SLOs to set out what performance your users expect. Even if your first version of an SLO consists of guesses, it is a first step towards creating a clear set of measurable goals for your performance tuning.

When you define SLOs, bear in mind that IG can depend on external resources that can impact performance, such as AM's response time for token validation, policy evaluation, and so on. Consider measuring remote interactions to take dependencies into account.

Consider defining SLOs for the following metrics of a route:

- Average response time for a route.

  The response time is the time to process and forward a request, and then receive, process, and forward the response from the protected application.

  The average response time can range from less than a millisecond, for a low latency connection on the same network, to however long it takes your network to deliver the response.

- Distribution of response times for a route.

  Because applications set timeouts based on worst case scenarios, the distribution of response times can be more important than the average response time.

- Peak throughput.

  The maximum rate at which requests can be processed at peak times. Because applications are limited by their peak throughput, this SLO is arguably more important than an SLO for average throughput.

- Average throughput.

  The average rate at which requests are processed.

Metrics are returned at the monitoring endpoints. For information about monitoring endpoints, refer to Monitoring. For examples of how to set up monitoring in IG, refer to Monitor services.

Available resources

With your defined SLOs, inventory the server, networks, storage, people, and other resources. Estimate whether it is possible to meet the requirements, with the resources at hand.

Benchmarks

Before you can improve the performance of your deployment, establish an accurate benchmark of its current performance. Consider creating a deployment scenario that you can control, measure, and reproduce.

For information about running ForgeRock Identity Platform benchmark tests, refer to the ForgeOps documentation on benchmarks. Adapt the scenarios as necessary for your IG deployment.

Tune IG

Consider the following recommendations for improving performance, throughput, and response times. Adjust the tuning to your system workload and available resources, and then test suggestions before rolling them out into production.

Logs

Log messages in IG and third-party dependencies are recorded using the Logback implementation of the Simple Logging Facade for Java (SLF4J) API. By default, logging level is INFO.

To reduce the number of log messages, consider setting the logging level to `error`. For information, refer to Manage logs.

Buffering message content

IG creates a TemporaryStorage object to buffer content during processing. For information about this object and its default values, refer to TemporaryStorage.

Messages bigger than the buffer size are written to disk, consuming I/O resources and reducing throughput.

The default size of the buffer is 64 KB. If the number of concurrent messages in your application is generally bigger than the default, consider allocating more heap memory or changing the initial or maximum size of the buffer.

To change the values, add a TemporaryStorage object named `TemporaryStorage`, and use non-default values.

Caches

When caches are enabled, IG can reuse cached information without making additional or repeated queries for the information. This gives the advantage of higher system performance, but the disadvantage of lower trust in results.

When caches are disabled, IG must query a data store each time it needs data. This gives the disadvantage of lower system performance, and the advantage of higher trust in results.

All caches provide similar configuration properties for timeout, defining the duration to cache entries. When the timeout is lower, the cache is evicted more frequently, and consequently, the performance is lower but the trust in results is higher.

When you configure caches in IG, make choices to balance your required performance with your security needs.

Learn more about IG caches in Caches.

WebSocket notifications

By default, IG receives WebSocket notifications from AM for the following events:

- When a user logs out of AM, or when the AM session is modified, closed, or times out. IG can use WebSocket notifications to evict entries from the session cache. For an example of setting up session cache eviction, refer to Session cache eviction.
- When AM creates, deletes, or changes a policy decision. IG can use WebSocket notifications to evict entries from the policy cache. For an example of setting up policy cache eviction, refer to Using WebSocket notifications to evict the policy cache.
- When IG automatically renews a WebSocket connection to AM. To configure WebSocket renewal, refer to the `notifications.renewalDelay` property of AmService.

If the WebSocket connection is lost, during that time the WebSocket is not connected, IG behaves as follows:

- Responds to session service calls with an empty SessionInfo result.

  When the SingleSignOn filter recieves an empty SessionInfo call, it concludes that the user is not logged in, and triggers a login redirect.

- Responds to policy evaluations with a deny policy result.

By default, IG waits for five seconds before trying to re-establish the WebSocket connection. If it can't re-establish the connection, it keeps trying every five seconds.

To disable WebSocket notifications, or change any of the parameters, configure the `notifications` property in AmService. For information, refer to AmService.

Tune the ClientHandler/ReverseProxyHandler

The ClientHandler/ReverseProxyHandler communicates as a client to a downstream third-party service or protected application. The performance of the communication is determined by the following parameters:

- The number of available connections to the downstream service or application.
- The connection timeout, which is the maximum time to connect to a server-side socket before timing out and abandoning the connection attempt.
- The socket timeout, which is the maximum time a request can take before a response is received after which the request is deemed to have failed.

Configure IG in conjunction with IG's first-class Vert.x configuration, and the `vertx` property of `admin.json`. For more information, refer to AdminHttpApplication (admin.json).

Vert.x options for tuning

Object	Vert.x Option	Description
IG (first-class)	`gatewayUnits`	The number of deployed Vert.x Verticles. This setting is effectively the number of cores that IG operates across, or in other words, the number of available threads. Each instance operates on the same port on its own event-loop thread. Default: Number of available cores. (This is the optimal value.)
root.vertx	`eventLoopPoolSize`	The size of the pool available to service Verticles for event-loop threads. To guarantee that a single thread handles all I/O events for a single request or response, IG deploys a Verticle onto each event loop. Configure `eventLoopPoolSize` to be greater than or equal to `gatewayUnits`. Default: 2 * number of available cores. For more information, refer to Reactor and Multi-Reactor.
root.connectors.<connector>.vertx	`acceptBacklog`	The maximum number of connections to queue before refusing requests.
	`sendBufferSize`	The TCP connection send buffer size. Set this property according to the available RAM and required number of concurrent connections.
	`receiveBufferSize`	The TCP receive buffer size. Set this property according to the available RAM and required number of concurrent connections.
	`"maxHeaderSize"`	Set this property when HTTP headers manage large values (such as JWT). Default: 8 KB (8,192 bytes)

Vert.x options for troubleshooting performance

Object	Vert.x Option	Description
root.vertx	`blockedThreadCheckInterval` and `blockedThreadCheckIntervalUnit`	The interval at which Vert.x checks for blocked threads and logs a warning. Default: 1 second
	`maxEventLoopExecuteTime` and `maxEventLoopExecuteTimeUnit`	The maximum execution time before Vert.x logs a warning. Default: 2 seconds

Object	Vert.x Option	Description
	`warningExceptionTime` and `warningExceptionTimeUnit`	The threshold at which warning logs are accompanied by a stack trace to identify cause. Default: 5 seconds
	`logActivity`	Log network activity.

Set the maximum number of file descriptors and processes per user

Each IG instance in your environment should have access to at least 65,536 file descriptors to handle multiple client connections.

Ensure that every IG instance is allocated enough file descriptors. For example, use the `ulimit -n` command to check the limits for a particular user:

```
$ su - iguser
$ ulimit -n
```

It may also be necessary to increase the number of processes available to the user running the IG processes.

For example, use the `ulimit -u` command to check the process limits for a user:

```
$ su - iguser
$ ulimit -u
```

Important

Before increasing the file descriptors for the IG instance, ensure that the total amount of file descriptors configured for the operating system is higher than 65,536.
If the IG instance uses all of the file descriptors, the operating system will run out of file descriptors. This may prevent other services from working, including those required for logging in the system.

Refer to your operating system's documentation for instructions on how to display and increase the file descriptors or process limits for the operating system and for a given user.

Tune IG's JVM

Start tuning the JVM with default values, and monitor the execution, paying particular attention to memory consumption, and GC collection time and frequency. Incrementally adjust the configuration, and retest to find the best settings for memory and garbage collection.

Make sure there is enough memory to accommodate the peak number of required connections, and make sure timeouts in IG and its container support latency in downstream servers and applications.

IG makes low memory demands, and consumes mostly YoungGen memory. However, using caches, or proxying large resources, increases the consumption of OldGen memory. For information about how to optimize JVM memory, refer to the Oracle documentation.

Consider these points when choosing a JVM:

- Find out which version of the JVM is available. More recent JVMs usually contain performance improvements, especially for garbage collection.
- Choose a 64-bit JVM if you need to maximize available memory.

Consider these points when choosing a GC:

- Test GCs in realistic scenarios, and load them into a pre-production environment.
- Choose a GC that is adapted to your requirements and limitations. Consider comparing the *Garbage-First Collector (G1)* and *Parallel GC* in typical business use cases.

  The G1 is targeted for multi-processor environments with large memories. It provides good overall performance without the need for additional options. The G1 is designed to reduce garbage collection, through low-GC latency. It is largely self-tuning, with an adaptive optimization algorithm.

  The Parallel GC aims to improve garbage collection by following a high-throughput strategy, but it requires more full garbage collections.

Learn more in Best practice for JVM Tuning with G1 GC.

Rotate keys

The following sections give an overview of how to manage rotation of encryption keys and signing keys, and include examples for key rotation based on use cases from the Gateway guide.

About key rotation

Key rotation is the process of generating a new version of a key, assigning that version as the *active* key to encrypt or sign new messages, or as a *valid* key to decrypt or validate messages, and then deprovisioning the old key.

Why and when to rotate keys

Regular key rotation is a security consideration that is sometimes required for internal business compliance. Regularly rotate keys to:

- Limit the amount of data protected by a single key.
- Reduce dependence on specific keys, making it easier to migrate to stronger algorithms.
- Prepare for when a key is compromised. The first time you try key rotation shouldn't be during a real-time recovery.

Key revocation is a type of key rotation, done exceptionally if you suspect that a key has been compromised. To decide when to revoke a key, consider the following points:

- If limited use of the old keys can be tolerated, provision the new keys and then deprovision the old keys. Messages produced before the new keys are provisioned are impacted.
- If use of the old keys cannot be tolerated, deprovision the old keys before you provision the new keys. The system is unusable until new keys are provisioned.

Steps for rotating symmetric keys

The following steps outline key rotation and revocation for symmetric keys managed by a KeyStoreSecretStore. For an example, refer to Rotate keys in a shared JWT session.

1. Using OpenSSL, Keytool, or another key creation mechanism, create the new symmetric key. The keystore should contain the old key and the new key.
2. Provision the new key.
    1. In the `mappings` property of KeyStoreSecretStore, add the alias for the new key after the alias for the old key. The new key is now valid. Because the old key is the first key in the list, it is the active key.
    2. Move the new key to be the first key in the list. The new key is now the active key.
3. Deprovision the old key.

    To ensure that no messages or users are impacted, wait until messages encrypted or signed with the old key are out of the system before you deprovision the old key.

    1. In the `mappings` property of KeyStoreSecretStore, delete the alias for the old key. The old key can no longer be used.
    2. Using OpenSSL, Keytool, or another key creation mechanism, delete the old symmetric key.

Steps for rotating asymmetric keys

The following steps outline the process for key rotation and revocation for asymmetric keys managed by a KeyStoreSecretStore or HsmSecretStore. For an example, refer to Rotate keys for stateless access tokens signed with a KeyStoreSecretStore.

1. Create new asymmetric keys for signing and encryption, using OpenSSL, Keytool, or another key creation mechanism.
2. Provision the message consumer with the private portion of the new encryption key, and the public portion of the new signing key.

    The message consumer can now decrypt and verify messages with the old key and the new key.
3. Provision the message producer, with the public portion of the new encryption key, and the private portion of the signing key. The message producer starts encrypting and signing messages with the new key, and stops using the old key.
4. Deprovision the message consumer with the private portion of the old encryption key, and the public portion of the old signing key. The message consumer can no longer decrypt and verify messages with the old key.

    To ensure that no messages or users are impacted, wait until messages encrypted or signed with the corresponding old key are out of the system before you deprovision the old key.
5. Deprovision the message producer, with the public portion of the old encryption key, and the private portion of old signing key.

Key rotation for keys in a JWK set

When keys are provided by a JWK Set from AM, the key rotation is transparent to IG. AM generates a key ID (`kid`) for each key it exposes at the `jwk_uri` . For more information, refer to Mapping and rotating secrets in AM's *Security guide*.

When IG processes a request with a JWT containing a `kid`, IG uses the `kid` to identify the key in the JWK Set. If the `kid` is available at the `jwk_uri` on AM, IG processes the request. Otherwise, IG tries all compatible secrets from the JWK Set. If none of the secrets work, the JWT is rejected.

Rotate keys for stateless access tokens signed with a KeyStoreSecretStore

This example extends the example in Validate signed access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore to rotate the keys that sign an access token and verify the signature.

Rotate Keys For Stateless Access Tokens Signed With a KeyStoreSecretStore

Before you start, set up and test the example in Validate signed access tokens with the StatelessAccessTokenResolver and KeyStoreSecretStore.

1. Set up the new keys:

    1. Generate a new private key called `signature-key-new`, and a corresponding public certificate called `x509certificate-new.pem`:

    ```
    $ openssl req -x509 \
    -newkey rsa:2048 \
    -nodes \
    -subj "/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \
    -keyout keystore_directory/signature-key-new.key \
    -out keystore_directory/x509certificate-new.pem \
    -days 365

    ... writing new private key to 'keystore_directory/signature-key-new.key'
    ```

    2. Convert the private key and certificate files into a new PKCS#12 keystore file:

    ```
    $ openssl pkcs12 \
    -export \
    -in keystore_directory/x509certificate-new.pem \
    -inkey keystore_directory/signature-key-new.key \
    -out keystore_directory/keystore-new.p12 \
    -passout pass:password \
    -name signature-key-new
    ```

    3. List the keys in the new keystore:

    ```
    $ keytool -list \
    -keystore "keystore_directory/keystore-new.p12" \
    -storepass "password" \
    -storetype PKCS12

    ...
    Your keystore contains 1 entry
    Alias name: signature-key-new
    ```

    4. Import the new keystore into `keystore.p12`, so that `keystore.p12` contains both keys:

```
$ keytool -importkeystore
-srckeystore keystore_directory/keystore-new.p12
-srcstoretype pkcs12
-srcstorepass password
-destkeystore keystore_directory/keystore.p12
-deststoretype pkcs12
-deststorepass password

Entry for alias signature-key-new successfully imported ...
```

5. List the keys in `keystore.p12`, to make sure it contains the new and old keys:

```
$ keytool -list \
-keystore "keystore_directory/keystore.p12" \
-storepass "password" \
-storetype PKCS12

...
Your keystore contains 2 entries
Alias name: signature-key
Alias name: signature-key-new
```

2. Set up AM:

    1. Copy the updated keystore to AM:

        1. Copy `keystore.p12` to AM:

```
$ cp keystore_directory/keystore.p12 am_keystore_directory/AM_keystore.p12
```

        2. List the keys in the updated AM keystore:

```
$ keytool -list \
-keystore "am_keystore_directory/AM_keystore.p12" \
-storepass "password" \
-storetype PKCS12

...
Your keystore contains 2 entries
Alias name: signature-key
Alias name: signature-key-new
```

        3. Restart AM to update the keystore cache.

    2. Update the KeyStoreSecretStore on AM:

        1. In AM, select **Secret Stores** > keystoresecretstore.

        2. Select the **Mappings** tab, and in `am.services.oauth2.stateless .signing.RSA` add the alias `signature-key-new`.

The mapping now contains two aliases, but the alias `signature-key` is still the active alias. AM still uses `signature-key` to sign tokens.

3. Drag `signature-key-new` above `signature-key`.

   AM now uses `signature-key-new` to sign tokens.

3. Set up IG:

   1. Set up IG for HTTPS, as described in Configure IG for HTTPS (server-side).

   2. Import the public certificate to the IG keystore, with the alias `verification-key-new`:

```
$ keytool -import \
-trustcacerts \
-rfc \
-alias verification-key-new \
-file "keystore_directory/x509certificate-new.pem" \
-keystore "ig_keystore_directory/IG_keystore.p12" \
-storetype PKCS12 \
-storepass "password"

...
Trust this certificate? [no]:  yes
Certificate was added to keystore
```

   3. List the keys in the IG keystore:

```
$ keytool -list \
-keystore "ig_keystore_directory/IG_keystore.p12" \
-storepass "password" \
-storetype PKCS12

...
Your keystore contains 2 entries
Alias name: verification-key
Alias name: verification-key-new
```

   4. In `rs-stateless-signed-ksss.json`, edit the KeyStoreSecretStore mapping with the new verification key:

```
"mappings": [
  {
    "secretId": "stateless.access.token.verification.key",
    "aliases": [ "verification-key", "verification-key-new" ]
  }
]
```

      If the Router `scanInterval` is disabled, restart IG to reload the route.

      IG can now check the authenticity of access tokens signed with `verification-key`, the old key, and `verification-key-new`, the new key. However, AM signs with the old key.

4. Test the setup:

    1. Get an access token for the demo user, using the scope `myscope`:

    ```
    $ mytoken=$(curl -s \
    --user "client-application:password" \
    --data "grant_type=password&username=demo&password=Ch4ng31t&scope=myscope" \
    http://am.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
    ```

    2. Display the token:

    ```
    $ echo ${mytoken}
    ```

    3. Access the route by providing the token returned in the previous step:

    ```
    $ curl -v \
    --cacert /path/to/secrets/ig.example.com-certificate.pem \
    --header "Authorization: Bearer ${mytoken}" \
    https://ig.example.com:8443/rs-stateless-signed-ksss


    ...
    Decoded access_token: {
    sub=demo,
    cts=OAUTH2_STATELESS_GRANT,
    ...
    ```

Deprovision Old Keys

1. Remove `signature-key` from the AM keystore:

    1. Delete the key from the keystore:

    ```
    $ keytool -delete \
    -keystore "am_keystore_directory/AM_keystore.p12" \
    -storepass "password" \
    -alias signature-key
    ```

    2. List the keys in the AM keystore to make sure `signature-key` is removed:

    ```
    $ keytool -list \
    -keystore "am_keystore_directory/AM_keystore-new.p12" \
    -storepass "password" \
    -storetype PKCS12
    ```

    3. Restart AM.

2. Remove `verification-key` from the IG keystore:

    1. Delete the key from the keystore:

    ```
    $ keytool -delete \
    -keystore "ig_keystore_directory/IG_keystore.p12" \
    -storepass "password" \
    -alias verification-key
    ```

    2. List the keys in the IG keystore to make sure that `verification-key` is removed:

    ```
    $ keytool -list \
    -keystore "ig_keystore_directory/IG_keystore.p12" \
    -storepass "password" \
    -storetype PKCS12
    ```

3. In AM, delete the mapping for `signature-key` from `keystoresecretstore`.

4. In IG, delete the mapping for `verification-key` from the route `rs-stateless-signed-ksss.json`. If the Router `scanInterval` is disabled, restart IG to reload the route.

Rotate keys in a shared JWT session

This section builds on the example in Share JWT Session Between Multiple Instances of IG to rotate a key used in a shared JWT session.

When a JWT session is shared between multiple instances of IG, the instances are able to share the session information for load balancing and failover.

Before you start, set up the example in Set up shared secrets for multiple instances of IG, where three instances of IG share a JwtSession and use the same authenticated encryption key. Instance 1 acts as a load balancer, and generates a session. Instances 2 and 3 access the session information.

1. Test the setup with the existing key, `symmetric-key`:

    1. Access instance 1 to generate a session:

    ```
    $ curl -v http://ig.example.com:8001/log-in-and-generate-session

    GET /log-in-and-generate-session HTTP/1.1
    ...

    HTTP/1.1 200 OK
    Content-Length: 84
    Set-Cookie: IG=eyJ...HyI; Path=/; Domain=.example.com; HttpOnly
    ...
    Sam Carter logged IN. (JWT session generated)
    ```

    2. Using the JWT cookie returned in the previous step, access instance 2:

```
$ curl -v http://ig.example.com:8001/webapp/browsing?one --header "cookie:IG=<JWT cookie>"

GET /webapp/browsing?one HTTP/1.1
...
cookie: IG=eyJ...QHyI
...
HTTP/1.1 200 OK
...
Hello, Sam Carter !! (instance2)
```

Note that instance 2 can access the session info.

3. Using the JWT cookie again, access instance 3:

```
$ curl -v http://ig.example.com:8001/webapp/browsing?two --header "Cookie:IG=<JWT cookie>"

GET /webapp/browsing?two HTTP/1.1
...
cookie: IG=eyJ...QHyI
...
HTTP/1.1 200 OK
...
Hello, Sam Carter !! (instance3)
```

Note that instance 3 can access the session info.

2. Commission a new key:

1. Generate a new encryption key, called `symmetric-key-new`, in the existing keystore:

```
$ keytool \
-genseckey \
-alias symmetric-key-new
-keystore /path/to/secrets/jwtsessionkeystore.pkcs12 \
-storepass password \
-storetype PKCS12 \
-keyalg HmacSHA512 \
-keysize 512
```

2. Make sure the keystore contains the old key and the new key:

```
$ keytool \
-list \
-keystore /path/to/secrets/jwtsessionkeystore.pkcs12 \
-storepass password \
-storetype PKCS12

...
Your keystore contains 2 entries
symmetric-key, ...
symmetric-key-new ...
```

3. Add the key alias to `instance1-loadbalancer.json`, `instance2-retrieve-session-username.json`, and `instance3-retrieve-session-username.json`, for each IG instance, as follows:

```
"mappings": [{
  "secretId": "jwtsession.encryption.secret.id",
  "aliases": ["symmetric-key", "symmetric-key-new"]
}]
```

If the Router `scanInterval` is disabled, restart IG to reload the route.

The active key is `symmetric-key`, and the valid key is `symmetric-key-new`.

4. Test the setup again, as described in step 1, and make sure instances 2 and 3 can still access the session information.

3. Make the new key the active key for generating sessions:

1. In `instance1-loadbalancer.json`, change the order of the keys to make `symmetric-key-new` the active key, and `symmetric-key` the valid key:

```
"mappings": [{
  "secretId": "jwtsession.encryption.secret.id",
  "aliases": ["symmetric-key-new", "symmetric-key"]
}]
```

Don't change `instance2-retrieve-session-username.json` or `instance3-retrieve-session-username.json`.

2. Test the setup again, as described in step 1, and make sure instances 2 and 3 can still access the session information.

Instance 1 creates the session using the new active key, `symmetric-key-new`.

Because `symmetric-key-new` is declared as a valid key in instances 2 and 3, the instances can still access the session. It isn't necessary to make `symmetric-key-new` the active key.

4. Decommission the old key:

1. Remove the old key from all of the routes, as follows:

```
"mappings": [{
  "secretId": "jwtsession.encryption.secret.id",
  "aliases": ["symmetric-key-new"]
}]
```

Key `symmetric-key-new` is the only key in the routes.

2. Remove the old key, `symmetric-key`, from the keystore:

    1. Delete `symmetric-key`:

```
$ keytool \
-delete \
-alias symmetric-key \
-keystore /path/to/secrets/jwtsessionkeystore.pkcs12 \
-storepass password \
-storetype PKCS12 \
-keypass password
```

2. Make sure the keystore contains only `symmetric-key-new`:

```
$ keytool \
-list \
-keystore /path/to/secrets/jwtsessionkeystore.pkcs12 \
-storepass password \
-storetype PKCS12

...
Your keystore contains 1 entry
symmetric-key-new ...
```

3. Test the setup again, as described in step 1, and make sure instances 2 and 3 can still access the session information.

Troubleshoot

ForgeRock provides support services, professional services, training through ForgeRock University, and partner services to help you set up and maintain your deployments.

Getting support

Ping Identity provides support services, professional services, training, and partner services to assist you in setting up and maintaining your deployments. For a general overview of these services, refer to https://www.pingidentity.com.

Ping Identity has staff members around the globe who support our international customers and partners. For details on Ping Identity's support offering, visit https://www.pingidentity.com/support.

Ping Identity publishes comprehensive documentation online:

- The Ping Identity support site offers a large and increasing number of up-to-date, practical articles that help you deploy and manage Ping Identity software.

  While many articles are visible to everyone, Ping Identity customers have access to much more, including advanced information for customers using Ping Identity software in a mission-critical capacity.

- Ping Identity product documentation, such as this document, aims to be technically accurate and complete with respect to the software documented. It is visible to everyone and covers all product features and examples of how to use them.

Getting info about the problem

When trying to solve a problem, save time by asking the following questions:

- How do you reproduce the problem?
- What behavior do you expect, and what behavior do you have?
- When did the problem start occurring?
- Are their circumstances in which the problem does not occur?
- Is the problem permanent, intermittent, getting better, getting worse, or staying the same?

If you contact ForgeRock for help, include the following information with your request:

- The product version and build information. This information is included in the logs when IG starts up. If IG is running in development mode, and set up as described in the Quick install, access the information at http://ig.example.com:8080/openig/api/info or https://ig.example.com:8443/openig/api/info.
- Description of the problem, including when the problem occurs and its impact on your operation.
- Steps you took to reproduce the problem.
- Relevant access and error logs, stack traces, and core dumps.
- Description of the environment, including the following information:
  - Machine type
  - Operating system and version
  - Web server or container and version
  - Java version
  - Patches or other software that might affect the problem

Displaying resources

By default, ForgeRock Access Management 5 and later writes cookies to the fully qualified domain name of the server; for example, `am.example.com`. Therefore, a host-based cookie, rather than a domain-based cookie, is set.

Consequently, after authentication through Access Management, requests can be redirected to Access Management instead of to the resource.

To resolve this issue, add a cookie domain to the Access Management configuration. For example, in the AM admin UI, go to **Configure** > **Global Services** > **Platform**, and add the domain `example.com`.

When the sample application is used with IG in the documentation examples, the sample application must serve static resources, such as the .css. Add the following route to the IG configuration:

```
{
  "name" : "00-static-resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${find(request.uri.path,'^/css') or matchesWithRegex(request.uri.path, '^/.*\\\\.ico$') or
matchesWithRegex(request.uri.path, '^/.*\\\\.gif$')}",
  "handler": "ReverseProxyHandler"
}
```

Define an entity for the response, as in the following example:

```
{
  "name": "AccessDeniedHandler",
  "type": "StaticResponseHandler",
  "config": {
    "status": 403,
    "headers": {
      "Content-Type": [ "text/html; charset=UTF-8" ]
    },
    "entity": "<html><body><p>User does not have permission</p></body></html>"
  }
}
```

Using routes

Symptom

The following errors are in `route-system.log`:

```
... | ERROR | main | o.f.o.h.r.RouterHandler | no handler to dispatch to
```

```
08:22:54:974 | ERROR | http-... | o.f.o.h.DispatchHandler | no handler to dispatch to for URI 'http://
ig.example.com/demo'
```

Cause

IG is not configured to handle the incoming request or the request to the specified URI:

- "no handler to dispatch to": the router cannot find a route that accepts the incoming request. This error happens when none of the route conditions match the incoming request and there is no default route.
- "no handler to dispatch to for URI": the router cannot find a route that can handle the request to the specified URI because none of the route conditions match the request path (URI).

Solution

If the errors occur during the startup, they are safe to ignore. If the errors occur after the startup, do the following:

- Identify why the request matched none of the Route conditions, and adapt the conditions. For examples, refer to Example conditions and requests.

- Add a default handler to the Router.
- Add a default route for when no condition is met.

If you have the following error, you have specified `"handler": "Router2"` in `config.json` or in the route, but no handler configuration object named Router2 exists:

```
org.forgerock.json.fluent.JsonValueException: /handler:
     object Router2 not found in heap
     at org.forgerock.openig.heap.HeapImpl.resolve(HeapImpl.java:351)
     at org.forgerock.openig.heap.HeapImpl.resolve(HeapImpl.java:334)
     at org.forgerock.openig.heap.HeapImpl.getHandler(HeapImpl.java:538)
```

Make sure you have added an entry for the handler, and that you have correctly spelled its name.

When the JSON for a route is not valid, IG does not load the route. Instead, a description of the error appears in the log.

Use a JSON editor or JSON validation tool such as JSONLint to make sure your JSON is valid.

IG loads all configurations at startup, and, by default, periodically reloads changed route configurations.

If you make changes to a route that result in an invalid configuration, IG logs errors, but it keeps the previous, correct configuration, and continues to use the old route.

IG only uses the new configuration after you save a valid version or when you restart IG.

Of course, if you restart IG with an invalid route configuration, then IG tries to load the invalid route at startup and logs an error. In that case, if there is no default handler to accept any incoming request for the invalid route, then you have an error, `No handler to dispatch to`.

IG returns an exception if it loads a route for which it can't resolve a requirement. For example, when you load a route that uses an AmService object, the object must be available in the AM configuration.

If you add routes to a configuration when the environment is not ready, rename the route to prevent IG from loading it. For example, rename a route as follows:

```
$ mv $HOME/.openig/config/routes/03-sql.json $HOME/.openig/config/routes/03-sql.inactive
```

If necessary, restart IG to reload the configuration. When you have configured the environment, change the file extension back to `.json`.

Using Studio

Studio deploys and undeploys routes through a main router named `_router`, which is the name of the main router in the default configuration. If you use a custom `config.json`, make sure it contains a main router named `_router`.

For information about creating routes in Studio, refer to the Studio guide.

Timeout errors

Problem: After a request is sent to IG, IG seems to hang. An HTTP 502 Bad Gateway error is produced, and the IG log is flushed with SocketTimeoutException warnings.

Possible cause: The `baseURI` configuration is missing or causes the request to return to IG, so IG can't produce a response to the request.

Possible solution: Configure the `baseURI` to use a different host and port to IG.

Other problems

Make sure the user running IG can read the flat file. Remember that values include spaces and tabs between the separator, so make sure the values are not padded with spaces.

The following error can be encountered when using an `AssignmentFilter` as described in AssignmentFilter and setting a string value for one of the headers.

```
HTTP ERROR 500
      Problem accessing /myURL . Reason:
      java.lang.String cannot be cast to java.util.List
      Caused by:
      java.lang.ClassCastException: java.lang.String cannot be cast to java.util.List
```

All headers are stored in lists so the header must be addressed with a subscript. For example, rather than trying to set `request.headers['Location']` for a redirect in the response object, you should instead set `request.headers['Location'][0]`. A header without a subscript leads to the error above.

When a request is longer than 4096 bytes, it can cause an HTTP 414 URI Too Long response.

The default limit for request length is set by the Vert.x configuration `DEFAULT_MAX_INITIAL_LINE_LENGTH`. This default acts on the `connectors` property of admin.json.

When working with requests constructed with parameters and query strings, such as for SAML or token transformation, where the request can become long consider setting the Vert.x property `getMaxInitialLineLength` to increase the limit.

The following example configuration in `admin.json` increases the request length limit to 9999 bytes:

```
"connectors": [
  {
    "vertx": {
      "maxInitialLineLength": 9999,
      ...
    }
  }
]
```

Security

Use this guide to reduce risk and mitigate threats to IG security.

Threats

Understand and address security threats.

Operating systems

Secure your operating systems.

Connections

Secure network connections.

Access

Remove non-essential access and features, update patches, and manage cookies.

Keys and Secrets

Manage keys and secrets.

Audit Trails

Audit events in your deployment.

Access

The following sections describe how to prevent unwanted access to your deployment, and reduce the amount of non-essential information that it provides.

Use an IG service account

Install and run IG from a dedicated service account. This is optional when evaluating IG, but essential in production installations. For more information, refer to Create an IG service account.

Remove non-essential access

Make sure only authorized people can access your servers and applications through the appropriate network, using the appropriate ports, and by presenting strong-enough credentials.

Apply the principle of least privilege to IG logs and configuration directories. For more information, refer to Configuration location.

Make sure that users connect to systems through the latest versions of TLS, and audit system access periodically.

Restrict access to your monitoring data by protecting the Prometheus Scrape Endpoint and Common REST Monitoring Endpoint (deprecated). Learn more from Protecting the monitoring endpoints.

Prevent IG from scanning for changes to routes. For information, see `scanInterval` in Router.

Disable administration endpoints and Studio by setting the IG run mode to `production`. For information, refer to Operating modes.

Remove non-essential features

The more features you have turned on, the greater the attack surface. If something isn't used, uninstall it, disable it, or protect access to it.

Update patches

Prevent the exploitation of security vulnerabilities by using up-to-date versions of IG and third-party software.

Review and follow the Ping Identity security advisories.

Follow similar lists from all of your vendors.

Manage sessions

Expire Identity Cloud and AM sessions

To minimize the time an attacker can attack an active session, set expiration timeouts for every Identity Cloud and AM session. Set timeouts according to context of the deployment, balancing security and usability so that the user can complete operations without the session frequently expiring.

For more information, refer to OWASP's Session Management Cheat Sheet.

Set a maximum session lifetime and idle time in Identity Cloud:

- In the Identity Cloud admin UI, select **Native Consoles** > **Access Management**.
- In the AM admin UI, select **Services** > **Add a Service** and add a **Session** service.

- Specify the following properties in minutes:
  - **Maximum Session Time**
  - **Maximum Idle Time**

Set a maximum session lifetime and idle time in Access Management:

- In the AM admin UI, select **Services** > **Add a Service** and add a **Session** service.
- Specify the following properties in minutes:
  - **Maximum Session Time**
  - **Maximum Idle Time**

Validate the signature of Identity Cloud and AM session cookies

Always configure `verificationSecretId` in the CrossDomainSingleSignOnFilter.

When `verificationSecretId` is not configured, IG does not verify the signature of AM session tokens, increasing the risk of CDSSO token tampering.

Manage cookies

Increase the security of cookies genrated by IG or the protected application in the following ways:

- Change the default name of cookies to prevent them from being easily associated with an application.
- Create cookies with the `secure` flag to ensure that browsers cannot transmit the cookie over non-SSL.

  When cookies have the `secure` flag, the first hop of the connection between the user agent and protected application must be secure (over HTTPs); subsequent hops do not have to be secure. In this example, the first hop from the user agent to NGINX is secure, the subsequent hop to IG is not secure:

  ```
  User agent -> NGINX (https://acme.com) -> IG (http://gateway:8080)-> protected application (https://
  internal.app:8081)
  ```

- Create cookies with the `httpOnly` flag, to ensure that the cookie cannot be accessed through client-side scripts, and to mitigate any cross-site scripting attacks.

  Cookies are `httpOnly` by default in `admin.json`, JwtSession, CrossDomainSingleSignOnFilter, and FragmentFilter.

- Set the `samesite` attribute of cookies to `STRICT` or `LAX`. For more information, refer to SameSite cookies.
- Set a timeout for cookies, to strike a good compromise between security and usability.

Harden an IG configuration by configuring the following objects:

- For stateful sessions, configure the `session.cookie` property in admin.json.
- For stateless sessions, configure the `cookie` property of JwtSession.
- For authentication results, configure the `authCookie` property of CrossDomainSingleSignOnFilter.

- For the fragment part of a URI when a request triggers a login redirect, configure the `cookie` property of FragmentFilter.

Threats

The following sections describe some of the possible threats to IG, which you can mitigate by following the instructions in this guide.

Out-of-date software

Prevent the exploitation of security vulnerabilities by using up-to-date versions of IG and third-party software.

Review and follow the Ping Identity security advisories.

Follow similar lists from all of your vendors.

Reconnaissance

The initial phase of an attack sequence is often reconnaissance. Limit the amount of information available to attackers during reconnaissance, as follows:

- Avoid using words that help to identify IG in error messages, such as those produced by the entity in a StaticResponseHandler. For information, see StaticResponseHandler.
- Use the lowest level of logging necessary. For example, consider logging at the `ERROR` or `WARNING` level, instead of `TRACE` or `MESSAGE`. For information, refer to Changing the global log level.

Cross-site scripting

When using a StaticResponseHandler, secure responses from cross-site scripting attacks, as follows:

- Sanitize any external input, such as the request, before incorporating it in the response.
- Specify `Content-Type` in the `headers` property of StaticResponseHandler when an entity is used. (Required by default, from IG 7.)
- Set the response header `X-Content-Type-Options: nosniff` to prevent the user agent from interpreting the response entity as a different content type. (Set by default, from IG 7.)
- Set a restrictive value in the `Cache-Control` response header. For example, setting `Cache-Control: private` indicates that all or part of the response message is intended for a single user and MUST NOT be cached by a shared cache.

Compromised passwords

Despite efforts to improve how people manage passwords, users have more passwords than ever before, and many use weak passwords. You are strongly encouraged to use a password manager to generate secure passwords. You can use identity and access management services to avoid password proliferation, and you can ensure the safety of passwords that you cannot eliminate.

Manage passwords for server administration securely. Passwords supplied to IG can be provided in files, through environment variables, or as system property values. Choose the approach that is most appropriate and secure for your deployment.

Misconfiguration

Misconfiguration can arise from bad or mistaken configuration decisions, and from poor change management. Depending on the configuration error, features can stop working in obvious or subtle ways, and potentially introduce security vulnerabilities.

The following behaviour can be caused by misconfiguration:

- Routes fail to load, or succeed in loading but cause unexpected behaviour.

  For example, if a configuration change prevents the server from making HTTPS connections, many applications can no longer connect, and the problem is detected immediately. However, if a configuration change allows insecure TLS protocol versions or cipher suites for HTTPS connections, some applications negotiate insecure TLS, but appear to continue to work properly.

- Access policy is not correctly enforced.

  Incorrect parameters for secure connections and incorrect Access Control Instructions (ACI) can lead to overly permissive access to data, and potentially to a security breach.

- The server fails to restart.

  Although failure to start a server is not directly a threat to security, it can affect service availability.

To guard against bad configuration decisions, implement good change management:

- For all enabled features, document why they are enabled and what your configuration choices mean. This implies a review of configuration settings, including default settings that you accept.
- Validate configuration decisions with thorough testing.
- Maintain a record of your configurations and the changes applied.

  For example, use a filtered audit log. Use version control software for any configuration scripts and to record changes to configuration files.

- Maintain a record of external changes to the system, such as changes to operating system configuration, and updates to software, such as the JVM that introduces security changes.

Unauthorized access

Data theft can occur when access policies are too permissive, and when the credentials to gain access are too easily cracked. It can also occur when the data is not protected, when administrative roles are too permissive, and when administrative credentials are poorly managed.

Poor risk management

Threats can arise when plans fail to account for outside risks. To mitigate risk, develop appropriate answers to at least the following questions:

- What happens when a server or an entire data center becomes unavailable?
- How do you remedy a serious security issue in the service, either in the IG software or the connected systems?
- How do you validate mitigation plans and remedial actions?

- How do client applications work when the IG offline?

  If client applications require always-on services, how do your operations ensure high availability, even when a server goes offline?

For critical services, test expected operation and disaster recovery operation.

Operating systems

When you deploy IG, familiarize yourself with the recommendations for the host operating systems that you use. For comprehensive information about securing operating systems, refer to the CIS Benchmark documentation.

System updates

Over the lifetime of a deployment, the operating system might be subject to vulnerabilities. Some vulnerabilities require system upgrades, whereas others require only configuration changes. All updates require proactive planning and careful testing.

For the operating systems used in production, put a plan in place for avoiding and resolving security issues. The plan should answer the following questions:

- How does your organization become aware of system security issues early?

  This could involve following bug reports, mailing lists, forums, and other sources of information.

- How do you test security fixes, including configuration changes, patches, service packs, and system updates?

  Validate the changes first in development, then in one or more test environments, then in production in the same way you would validate other changes to the deployment.

- How do you roll out solutions for security issues?

  In some cases, fixes might involve both changes to the service, and specific actions by those who use the service.

- What must you communicate about security issues?

- How must you respond to security issues?

Software providers often do not communicate what they know about a vulnerability until they have a way to mitigate or fix the problem. Once they do communicate about security issues, the information is likely to become public knowledge quickly. Make sure you can expedite resolution of security issues.

To resolve security issues quickly, make sure you are ready to validate any changes that must be made. When you validate a change, check that the fix resolves the security issue. Validate that the system and IG software continue to function as expected in all the ways they are used.

System audits

System audit logs make it possible to uncover system-level security policy violations that are not recorded in IG, such as unauthorized access to IG files. Such violations are not recorded in IG logs or monitoring information.

Also consider how to prevent or at least detect tampering. A malicious user violating security policy is likely to try to remove evidence of how security was compromised.

Unused features

By default, operating systems include many features, accounts, and services that IG software does not require. Each optional feature, account, and service on the system brings a risk of additional vulnerabilities. To reduce the surface of attack, enable only required features, system accounts, and services. Disable or remove those that are not needed for the deployment.

The features needed to run and manage IG software securely include the following:

- A Java runtime environment, required to run IG software.
- Software to secure access to service management tools; in particular, when administrators access the system remotely.
- Software to secure access for remote transfer of software updates, backup files, and log files.
- Software to manage system-level authentication, authorization, and accounts.
- Firewall software, intrusion-detection/intrusion-prevention software.
- Software to allow auditing access to the system.
- System update software to allow updates that you have validated previously.
- If required for the deployment, system access management software such as SELinux.
- Any other software that is clearly indispensable to the deployment.

Consider the minimal installation options for your operating system, and the options to turn off features.

Consider configuration options for system hardening to further limit access even to required services.

For each account used to run a necessary service, limit the access granted to the account to what is required. This reduces the risk that a vulnerability in access to one account affects multiple services across the system.

Make sure you validate the operating system behavior every time you deploy new or changed software. When preparing the deployment and when testing changes, maintain a full operating system with IG software that is not used for any publicly available services, but only for troubleshooting problems that might stem from the system being *too* minimally configured.

Network connections

Protect network traffic by using HTTPS where possible, and secure communications during stateless sessions by signing and/or encrypting JWTs. For information about configuring IG for HTTPS client-side and HTTPS server-side, refer to the Installation guide.

Recommendations for incoming connections (from clients to IG).

Protocol	Recommendations
HTTP	HTTP connections that are not protected by SSL/TLS use cleartext messages. When you permit insecure connections, you cannot prevent client applications from sending sensitive data. For example, a client could send unprotected credentials in an HTTP Authorization header. Even if the server were to reject the request, the credentials would already be leaked to any eavesdroppers. Always use HTTPS for connections up to a load-balancer or proxy in front of the web application or server.
HTTPS	Follow industry-standard TLS recommendations for Security/Server Side TLS. Use a secure version of TLS/SSL to connect to TLS-protected endpoints with HTTP connection handlers, such as ClientHandler and ReverseProxyHandler. TLS protocols below 1.2 aren't considered secure. Some client applications require a higher level of trust, such as clients with additional privileges or access. Client application deployers might find it easier to manage public keys as credentials than to manage user name/password credentials. Client applications can use SSL client authentication. When using IG REST to LDAP gateway, use HTTPS to protect client connections.
JMX	Secure JMX access with the SSL/TLS-related properties, such as `use-ssl` and others.
SSH	IG administration tools can connect securely. Administrators should use SSH when changing the IG configuration or binaries. The user account for running IG should not be the same user account for connecting remotely. Secure Copy (SCP) uses SSH to transfer files securely. SCP is an appropriate protocol for copying backup data, for example.

Recommendations for outgoing connections (from IG to another service.)

Client	Recommendations
Common Audit event handlers	Configure ForgeRock Common Audit event handlers to use HTTPS when connecting to external log services.
OAuth 2.0-based HTTP authorization mechanisms	HTTP authorization can be based on OAuth 2.0, where IG servers act as resource servers, and make requests to resolve OAuth 2.0 tokens. Use HTTPS to protect the connections to OAuth2ResourceServerFilter and AuthorizationCodeOAuth2ClientFilter. For information, refer to OAuth2ResourceServerFilter and AuthorizationCodeOAuth2ClientFilter.

Message-level security

Server protocols such as HTTP and JMX rely on TLS to protect connections. To enforce secure communication, configure TLS as follows:

- HTTPS server-side: Configure admin.json, Configure IG for HTTPS (server-side).
- HTTPS client-side: Configure trust managers and key managers, as described in Configure IG for HTTPS (client-side).

When negotiating connection security, the server and client must use a common security protocol and cipher suite. In ClientTlsOptions and ServerTlsOptions, define lists of security protocols and cipher suites. For security, use the most recent protocols and ciphers that the client supports. Clients with older TLS implementations might not support the most recent protocols and ciphers.

Keys and secrets

IG uses cryptographic keys for encryption, signing, and securing network connections, and passwords. The following sections describe how to secure keys and secrets in your deployment.

About secrets

IG uses the Commons Secrets API to manage secrets, such as passwords and cryptographic keys.

Repositories of secrets are managed through secret stores, provided to the configuration by the `SecretsProvider` object or `secrets` object. For more information about these objects and the types of secret stores provided in IG, refer to SecretsProvider and Secrets.

Secret types

IG uses two secret types:

- `GenericSecret`: An opaque blob of bytes, such as a password or API key, without any metadata. A `GenericSecret` cannot be used to perform cryptographic operations.
- `CryptoKey`: A secret that contains either a private or shared key, and/or a public certificate. A `CryptoKey` contains the secret material itself and its metadata; for example, the associated algorithm or key type. This secret type can be used for cryptographic operations.

For example:

- A `Base64EncodedSecretStore` can only serve secrets of the `GenericSecret` type.
- An `HsmSecretStore` can only server secrets of the `CryptoKey` type.
- A `FileSystemSecretStore` can serve secrets of both types.

To learn more about secret store specificities, refer to Secret Stores.

Secret terminology

The following terms describe secrets:

- **Secret ID**: A label to indicate the purpose of a secret. A secret ID is generally associated with one or more aliases of a key in a keystore or HSM.
- **Stable ID**: A label to identify a secret. The stable ID corresponds to the following values in each type of secret store:
  - Base64EncodedSecretStore: The value of `secret-id` in the `"secret-id": "string"` pair.
  - FileSystemSecretStore: The filename of a file in the specified directory, without the prefix/suffix defined in the store configuration.
  - HsmSecretStore: The value of an `alias` in a `secret-id` / `aliases` mapping.
  - JwkSetSecretStore: The value of the `kid` of a JWK stored in a JwkSetSecretStore.
  - KeyStoreSecretStore: The value of an `alias` in a `secret-id` / `aliases` mapping.
  - SystemAndEnvSecretStore: The name of a system property or environment. variable
- **Valid secret**: A secret whose purpose matches the secret ID **and** any purpose constraints. Constraints can include requirements for the following:
  - Secret type, such as signing key or encryption key
  - Cryptographic algorithm, such as Diffie-Hellman and RSA
  - Signature algorithm, such as ES256 and ES384

  Constraints are defined when the secret is generated, and cannot be added after.
- **Named secret**: A valid secret that a secret store can find by using a secret ID and stable ID.
- **Active secret**: One of the valid secrets that is considered eligible at the time of use. The way that the active secret is chosen is determined by the type of secret store. For more information, refer to Secrets,

About keys and certificates

The examples in this doc set use self-signed certificates, but your deployment is likely to use certificates issued by a certificate authority (CA certificates).

The way to obtain CA certificates depends on the certificate authority that you are using, and is not described in this document. As an example, refer to Let's Encrypt.

Integrate CA certificates by using secret stores:

- For PEM files, use a FileSystemSecretStore and PemPropertyFormat
- For PKCS12 keystores, use a KeyStoreSecretStore

For examples, refer to Serve the same certificate for TLS connections to all server names.

Note the following points about using secrets:

- When IG starts up, it listens for HTTPS connections, using the ServerTlsOptions configuration in `admin.json`. The keys and certificates are fetched at startup.
- Keys and certificates must be present at startup.
- If keys or certificates change, you must to restart IG.
- When the `autoRefresh` property of FileSystemSecretStore or KeyStoreSecretStore is enabled, the secret store is automatically reloaded when the filesystem or keystore is changed.

For information about secret stores provided in IG, refer to Secrets.

Validate signatures of signed tokens

IG validates the signature of signed tokens as follows:

- Named secret resolution:
  - If the JWT contains a `kid`, IG queries the secret stores declared in `secretsProvider` or `secrets` to find a named secret, identified by a secret ID and stable ID.
  - If a named secret is found, IG then uses the named secret to try to validate the signature. If the named secret can't validate the signature, the token is considered as invalid.
  - If a named secret isn't found, IG tries valid secret resolution.
- Valid secret resolution:
  - IG uses the value of `verificationSecretId` as the secret ID, and queries the declared secret stores to find all secrets that match the provided secret ID.
  - All matching secrets are returned as valid secrets, in the order that the secret stores are declared, and for KeyStoreSecretStore and HsmSecretStore, in the order defined by the mappings.
  - IG tries to verify the signature with each valid secret, starting with the first valid secret, and stopping when it succeeds.
  - If no valid secrets are returned, or if none of the valid secrets can verify the signature, the token is considered as invalid.

For examples where a StatelessAccessTokenResolver uses a secret store to validate the signature of signed tokens, refer to the example sections of JwkSetSecretStore and KeyStoreSecretStore.

Using multiple secret stores in a configuration

When multiple secrets stores are provided in a configuration, the secrets stores are queried in the following order:

- Locally in the route, starting with the first secret store in the list, up to the last.
- In ascending parent routes, starting with the first secret store in each list, up to the last.
- In `config.json`, starting with the first secret store in the list, up to the last.

- If a secrets store is not configured in `config.json`, the secret is queried in a default SystemAndEnvSecretStore, and a base64-encoded value is expected.
- If a secret is not resolved, an error is produced.

Secrets stores defined in `admin.json` can be accessed only by heap objects in `admin.json`.

Algorithms for elliptic curve digital signatures

When the Elliptic Curve Digital Signature Algorithm (ECDSA) is used for signing, and both of the following conditions are met, JWTs are signed with a deterministic ECDSA:

- Bouncy Castle is installed.
- The system property `org.forgerock.secrets.preferDeterministicEcdsa` is `true`, which is its default value.

Otherwise, when ECDSA is used for signing, JWTs are signed with a non-deterministic ECDSA.

A non-deterministic ECDSA signature can be verified by the equivalent deterministic algorithm.

For information about deterministic ECDSA, refer to RFC 6979. For information about Bouncy Castle, refer to The Legion of the Bouncy Castle.

Update cryptography

Different algorithms and methods are discovered and tested over time, and communities of experts decide which are the most secure for different uses. Use up-to-date cryptographic methods and algorithms to generate keys.

> **Warning**
>
> Legacy keystore types such as JKS and JCEKS are supported but are not secure. Consider using the PKCS#12 keystore type.

Use strong keys

Small keys are easily compromised. Use at least the recommended key size.

In JVM, the default ephemeral Diffie-Hellman (DH) key size is 1024 bits. To support stronger ephemeral DH keys, and protect against weak keys, consider setting the following system property to increase the DH key size: `jdk.tls.ephemeralDHKeySize=2048`.

For more information, refer to Customizing size of ephemeral Diffie-Hellman keys

Rotate keys

Rotate keys regularly to:

- Limit the amount of data protected by a single key.
- Reduce dependence on specific keys, making it easier to migrate to stronger algorithms.
- Prepare for when a key is compromised. The first time you try key rotation shouldn't be during a real-time recovery.

- Conform to internal business compliance requirements.

For more information, refer to Rotate keys.

Audits and logs

Audit trails

Audits in IG record access to a route. Audit logs in operating systems detect system login attempts and changes to the software.

The IG audit logging service adheres to the log structure common across the ForgeRock Identity Platform. For information, refer to Audit the deployment.

Prevent logging of sensitive data for audit events by excluding fields from the audit logs. For information, refer to Including or excluding audit event fields in logs.

Log files

Logs in IG contain informational, error, and warning events, to troubleshoot and debug transactions and events that take place within the IG instance.

Protect logs from unauthorised access, and make sure they contain a minimum of sensitive or personally identifiable information that could be used in attacks.

When using a CaptureDecorator, mask captured header and attribute values to avoid disclosing information, such as token values or passwords. For information, refer to CaptureDecorator.

Limit the number of repeat log messages to prevent log flow attacks, by adding a custom `logback.xml` with a `DuplicateMessageFilter`. For information, refer to Limit repetitive log messages.

Reference

This guide describes configuration options for IG. It is for IG designers, developers, and administrators.

For API specifications, refer to the appropriate Javadoc.

The examples in this guide use some of the following third-party tools:

- `curl`: https://curl.haxx.se
- `HTTPie`: https://httpie.org
- `jq`: https://stedolan.github.io/jq/
- `keytool`: https://docs.oracle.com/en/java/javase/11/tools/keytool.html

Reserved routes

By default, IG reserves all paths starting with `/openig` for administrative use, and only local client applications can access resources exposed under `/openig`.

IG uses an ApiProtectionFilter to protect reserved routes. By default, the ApiProtectionFilter allows access to reserved routes only from the loopback address. To override this behavior, declare a custom ApiProtectionFilter in the top-level heap. For an example, refer to the CORS filter described in Set up the UMA example.

For information about how to change the base for administrative routes, refer to Change the base location.

Reserved field names

IG reserves all configuration field names that contain only alphanumeric characters.

If you must define your own field names, for example, in custom decorators, use names with dots, `.`, or dashes, `-`. Examples include `my-decorator` and `com.example.myDecorator`.

Field value conventions

IG configuration uses JSON notation.

This reference uses the following terms when referring to values of configuration object fields:

array

JSON array.

boolean

Either `true` or `false`.

certificate

`java.security.cert.Certificate` instance.

configuration token

Configuration tokens introduce variables into the server configuration. They can take values from Java system properties, environment variables, JSON and Java properties files held in specified directories, and from properties configured in routes. For more information, refer to JSON Evaluation.

duration

A duration is a lapse of time expressed in English, such as `23 hours 59 minutes and 59 seconds`. Durations are not case sensitive, and negative durations are not supported. The following units can be used in durations:

- `indefinite`, `infinity`, `undefined`, `unlimited`: unlimited duration
- `zero`, `disabled`: zero-length duration
- `days`, `day`, `d`: days
- `hours`, `hour`, `h`: hours
- `minutes`, `minute`, `min`, `m`: minutes
- `seconds`, `second`, `sec`, `s`: seconds
- `milliseconds`, `millisecond`, `millisec`, `millis`, `milli`, `ms`: milliseconds
- `microseconds`, `microsecond`, `microsec`, `micros`, `micro`, `us`, `µs`: microseconds
- `nanoseconds`, `nanosecond`, `nanosec`, `nanos`, `nano`, `ns`: nanoseconds

enumeration

A collections of constants.

expression

See Expressions.

configuration expression

Expression evaluated at configuration time, when routes are loaded. See Configuration Expressions.

runtime expression

Expression evaluated at runtime, for each request and response. See Runtime Expressions.

instant

An instantaneous point on the timeline, as a Java type. For more information, see Class Instant.

JsonValue

An object (JsonObject), an array (JsonArray), a number (JsonNumber), a string (JsonString), true (JsonValue.TRUE), false (JsonValue.FALSE), or null (JsonValue.NULL).

lvalue-expression

Expression yielding an object whose value is to be set.

Properties whose format is `lvalue-expression` cannot consume streamed content. They must be written with `$` instead of `#`.

map

An object that maps keys to values. Keys must be unique, and can map to at most one value.

number

JSON number.

object

JSON object where the content depends on the object's type.

pattern

A regular expression according to the rules for the Java Pattern class.

pattern-template

Template for referencing capturing groups in a pattern by using `$n`, where `n` is the index number of the capturing group starting from zero.

For example, if the pattern is `\w+\s*=\s*(\w)+`, the pattern-template is `$1`, and the text to match is `key = value`, the pattern-template yields `value`.

reference

References an object in the following ways:

- An inline configuration object, where the name is optional.
- A configuration expression that is a string or contains variable elements that evaluate to a string, where the string is the name of an object declared in the heap.

  For example, the following `temporaryStorage` object takes the value of the system property `storage.ref`, which must a be string equivalent to the name of an object defined in the heap:

  ```
  {
    "temporaryStorage": "${system['storage.ref']}"
  }
  ```

secret-id

String that references a secret managed by the Commons Secrets API, as described in Secrets.

The secret ID must conform to the following regex pattern: `Pattern.compile("(\\.[a-zA-Z0-9])*");`

string

JSON string.

url

String representation for a resource available via the Internet. For more information, refer to Uniform Resource Locators (URL).

Required configuration

AdminHttpApplication (`admin.json`)

The AdminHttpApplication serves requests on the administrative route, such as the creation of routes and the collection of monitoring information. The administrative route and its subroutes are reserved for administration endpoints.

The configuration is loaded from a JSON-encoded file, expected at `$HOME/.openig/config/admin.json`. Objects configured in `admin.json` cannot be used by `config.json` or any IG routes.

Default objects

IG provides default objects in `admin.json`. To override a default object, configure an object with the same name in `admin.json`.

Configure default objects in `admin.json` and `config.json` separately. An object configured in `admin.json` with the same name as an object configured in `config.json` isn't the same object.

`AuditService`

Records no audit events. The default AuditService is `NoOpAuditService`. Learn more from NoOpAuditService.

`CaptureDecorator`

Captures requests and response messages. The default CaptureDecorator is named `capture`, and uses the default settings given in CaptureDecorator.

When a capture point for the default CaptureDecorator is defined in a route, for example, when `"capture: "all"` is set as a top-level attribute of the JSON, log messages for requests and responses passing through the route are written to a log file in `$HOME/.openig/logs`.

When no capture point is defined in a route, only exceptions thrown during request or response processing are logged.

By default, request and response contexts and entities are not captured. Do one of the following to capture information:

- Override the default capture decorator declaration, and set `captureEntity` to `true`.
- Declare another CaptureDecorator object with an appropriate configuration and use it at your capture points.

The capture decorator logs information about the HTTP request and response messages, along with their respective headers.

`ClientHandler`

Communicates with third-party services. Learn more from ClientHandler.

`ForgeRockClientHandler`

Sends ForgeRock Common Audit transaction IDs when communicating with protected applications. The default ForgeRockClientHandler is a Chain, composed of a TransactionIdOutboundFilter and a ClientHandler.

`IssuerRepository`

A repository of Issuers declared in the heap. To overwrite the default issuer, configure a local IssuerRepository with the name `IssuerRepository`. To create a new IssuerRepository containing a subset of Issuers, configure a local IssuerRepository with a different name.

`ProxyOptions`

A proxy to which a ClientHandler or ReverseProxyHandler can submit requests, and an AmService can submit Websocket notifications. For more information, refer to ProxyOptions.

`ReverseProxyHandler`

Communicates with third-party services. For more information, refer to ReverseProxyHandler.

`ScheduledExecutorService`

Specifies the number of threads in a pool.

`SecretsService` *(deprecated)*

Manages a store of secrets from files, system properties, and environment variables, by using Commons Secrets API. The default SecretsService is a SystemAndEnvSecretStore with the default configuration. For more information, refer to Secrets.

`TemporaryStorage`

Manages temporary buffers. For more information, refer to TemporaryStorage.

`TimerDecorator`

Records time spent within filters and handlers. The default TimerDecorator is named `timer`. For more information, refer to TimerDecorator.

`TransactionIdOutboundFilter`

Inserts the ID of a transaction into the header of a request.

Provided objects

IG creates the following objects when a filter with the name of the object is declared in `admin.json`:

`"ApiProtectionFilter"`

A filter to protect administrative APIs on reserved routes. By default, only the loopback address can access reserved routes.

For an example that uses an ApiProtectionFilter, refer to Set up the UMA example. For information about reserved routes, refer to Reserved routes.

`"MetricsProtectionFilter"`

A filter to protect the monitoring endpoints.

By default, the Prometheus Scrape Endpoint and Common REST Monitoring Endpoint (deprecated) are open and accessible; no special credentials or privileges are required to access the monitoring endpoints.

For an example that uses a MetricsProtectionFilter, refer to Protect monitoring endpoints.

`"StudioProtectionFilter"`

A filter to protect the Studio endpoint when IG is running in development mode.

When IG is running in development mode, by default the Studio endpoint is open and accessible.

For an example that uses a StudioProtectionFilter, refer to Restrict access to Studio.

Usage

```
{
  "heap": [ object, ... ],
  "connectors": [ object, ... ],
  "vertx": object,
  "gatewayUnits": configuration expression<number>,
  "mode": configuration expression<enumeration>,
  "prefix": configuration expression<string>,
  "properties": object,
  "temporaryDirectory": configuration expression<string>,
  "temporaryStorage": TemporaryStorage reference,
  "pidFileMode": configuration expression<enumeration>,
  "preserveOriginalQueryString": configuration expression<boolean>,
  "session": object,
  "streamingEnabled": configuration expression<boolean>
}
```

Properties

`"heap"`: *array of objects, optional*

The heap object configuration, described in Heap objects.

`"connectors"`: *array of objects, required*

Server port configuration, when IG is acting *server-side*.

> **Note**
>
> When an application sends requests to IG or requests services from IG, IG is server-side. IG is acting as a server of the application, and the application is acting as a client.

```
{
  "connectors" : [
    {
      "port": [ configuration expression<number>, ... ],
      "tls": ServerTlsOptions reference,
      "vertx": object,
      "maxTotalHeadersSize": configuration expression<integer>
    },
    {
      ...
    }
  ]
}
```

`port`: *array of configuration expression<numbers>, required*

One or more ports on which IG is connected. When more than one port is defined, IG is connected to each port.

`tls`: *ServerTlsOptions reference, optional*

Configure options for connections to TLS-protected endpoints, based on ServerTlsOptions. Define the object inline or in the heap.

Default: Connections to TLS-protected endpoints are not configured.

`vertx`: *object, optional*

Vert.x-specific configuration for this connector when IG is acting *server-side*. When IG is acting *client-side*, configure the `vertx` property of ClientHandler or ReverseProxyHandler.

Vert.x options are described in HttpServerOptions.

For properties where IG provides its own first-class configuration, Vert.x configuration options are disallowed, and the IG configuration option takes precedence over Vert.x options configured in `vertx`. Vert.x values are evaluated as configuration expressions.

The following Vert.x configuration options are disallowed server-side:

- `port`
- `useAlpn`
- `ssl`
- `enabledCipherSuites`
- `enabledSecureTransportProtocols`
- `jdkSslEngineOptions`
- `keyStoreOptions`
- `openSslEngineOptions`
- `pemKeyCertOptions`
- `pemTrustOptions`

- `pfxKeyCertOptions`
- `pfxTrustOptions`
- `trustStoreOptions`
- `clientAuth`

The following Vert.x configuration options are deprecated server-side:

- `maxHeaderSize`
- `initialSettings:maxHeaderListSize`

Use `connectors:maxTotalHeadersSize` instead of `vertx.maxHeaderSize` or `vertx.initialSettings.maxHeaderListSize`.

The following example configures connectors on ports 8080 and 8443 when IG is acting server-side. When IG is acting client-side, configure the `vertx` property of ClientHandler or ReverseProxyHandler:

```
{
  "connectors": [{
    "port": 8080,
    "vertx": {
      "maxWebSocketFrameSize": 128000,
      "maxWebSocketMessageSize": 256000,
      "compressionLevel": 4
    }
  },
  {
    "port": 8443,
    "tls": "ServerTlsOptions-1"
  }]
}
```

`maxTotalHeadersSize`: *integer*, *optional*

The maximum size in bytes of the sum of all request headers. When the request headers exceed this limit, IG returns an HTTP 431 error.

Default: 8 192 bytes

The following example configures HTTP/2 connections on port 7070 when IG is acting server-side. The configuration allows IG to accept HTTP/2 requests with large headers. When IG is acting client-side, configure the `vertx` property of ClientHandler or ReverseProxyHandler:

```
{
  "connectors": [
    {
      "port": 7070,
      "maxTotalHeadersSize": 16384
    }
  ]
}
```

`vertx`: *object*, *optional*

Vert.x-specific configuration used to more finely-tune Vert.x instances. Vert.x values are evaluated as configuration expressions.

Use the Vert.x options described in VertxOptions, with the following exceptions:

- `metricsOptions`: Not used
- `metricsEnabled`: Enable Vertx metrics. Default: `true`.

For an example, refer to Monitoring Services.

IG proxies all WebSocket subprotocols by default. To proxy specific WebSocket subprotocols only, list them as follows:

```
"vertx": {
  "webSocketSubProtocols": ["v1.notifications.forgerock.org", ... ]
}
```

`"gatewayUnits"`: *configuration expression*<*number*>, *optional*

The number of parallel instances of IG to bind to an event loop. All instances listen on the same ports.

Default: The number of cores available to the JVM.

`mode`: *configuration expression*<*enumeration*>, *optional*

Set the IG mode to `development` or `production`. The value is not case-sensitive.

If `mode` is not set, the provided configuration token `ig.run.mode` can be resolved at startup to define the run mode.

For more information, refer to Operating modes.

Default: `production`

`"prefix"`: *configuration expression*<*string*>, *optional*

The base of the route for administration requests. This route and its subroutes are reserved for administration endpoints.

Default: `openig`

`"properties"`: *object*, *optional*

Configuration parameters declared as property variables for use in the configuration. See also Route properties.

Default: Null

`"temporaryDirectory"`: *configuration expression*<*string*>, *optional*

Directory containing temporary storage files.

Set this property to store temporary files in a different directory, for example:

```
{
   "temporaryDirectory": "/path/to/my-temporary-directory"
}
```

Default: `$HOME/.openig/tmp` (on Windows, `%appdata%\OpenIG\tmp`)

`"temporaryStorage"`: *TemporaryStorage reference, optional*

The TemporaryStorage object to buffer content during processing.

Provide the name of a TemporaryStorage object defined in the heap or an inline TemporaryStorage configuration object.

Incoming requests use the temporary storage buffer as follows:

- Used only when `streamingEnabled` is `false`.
- The request is loaded into the IG storage defined in `temporaryStorage`, before it enters the chain.
- If the content length of the request is more than the buffer limit, IG returns an `HTTP 413 Payload Too Large`.

Default: Use the heap object named TemporaryStorage. Otherwise, use an internally-created TemporaryStorage object named `TemporaryStorage` that uses default settings for a TemporaryStorage object.

`"pidFileMode"`: *configuration expression<enumeration>, optional*

Mode to allow or disallow startup if there is an existing PID file. Use one of the following values:

- `fail`: Startup fails if there is an existing PID file.
- `override`: Startup is allowed if there is an existing PID file. IG removes the existing PID file and creates a new one during startup.

Default: `fail`

`"preserveOriginalQueryString"`: *configuration expression<boolean>, optional*

Process query strings in URLs, by applying or not applying a decode/encode process to the whole query string.

The following characters are disallowed in query string URL components: `"`, `{`, `}`, `<`, `>`, (space), and `|`. For more information about which query strings characters require encoding, refer to Uniform Resource Identifier (URI): Generic Syntax.

- `true`: Preserve query strings as they are presented.

  Select this option if the query string must not change during processing, for example, in signature verification.

  If a query string contains a disallowed character, the request produces a `400 Bad Request`.

- `false`: Tolerate disallowed characters in query string URL components, by applying a decode/encode process to the whole query string.

  Select this option when a user agent or client produces query searches with disallowed characters. IG transparently encodes the disallowed characters before forwarding requests to the protected application.

Characters in query strings are transformed as follows:

- Allowed characters are not changed. For example, `sep=a` is not changed.
- Percent-encoded values are re-encoded when the decoded value is an allowed character. For example, `sep=%27` is changed to `sep='`, because `'` is an allowed character.
- Percent-encoded values are not changed when the decoded value is a disallowed character. For example, `sep=%22` is not changed, because `"` is a disallowed character.
- Disallowed characters are encoded. For example, `sep="`, is changed to `sep=%22`, because `"` is a disallowed character.

Default: `false`

`"session"`: *object*, *optional*

Configures stateful sessions for IG. For information about IG sessions, refer to Sessions.

```
{
  "session": {
    "cookie": {
      "name": configuration expression<string>,
      "httpOnly": configuration expression<boolean>,
      "path": configuration expression<string>,
      "sameSite": configuration expression<enumeration>,
      "secure": configuration expression<boolean>,
    },
    "timeout": configuration expression<duration>
  }
}
```

`"cookie"`: *object*, *optional*

The configuration of the cookie used to store the stateful session.

Default: The session cookie is treated as a host-based cookie.

`"name"`: *configuration expression<string>*, *optional*

The session cookie name.

Default: `IG_SESSIONID`

`"httpOnly"`: *configuration expression<boolean>*, *optional*

Flag to mitigate the risk of client-side scripts accessing protected session cookies.

Default: `true`

`"path"`: *configuration expression<string>*, *optional*

The path protected by the session.

Set a path only if the user agent is able to re-emit session cookies on the path. For example, to re-emit a session cookie on the path `/home/cdsso`, the user agent must be able to access that path on its next hop.

Default: `/`.

`"sameSite"`: *configuration expression<enumeration>, optional*

Options to manage the circumstance in which the session cookie is sent to the server. The following values are listed in order of strictness, with most strict first:

- `STRICT`: Send the session cookie only if the request was initiated from the session cookie domain. Not case-sensitive. Use this value to reduce the risk of cross-site request forgery (CSRF) attacks.
- `LAX`: Send the session cookie only with GET requests in a first-party context, where the URL in the address bar matches the session cookie domain. Not case-sensitive. Use this value to reduce the risk of cross-site request forgery (CSRF) attacks.
- `NONE`: Send the session cookie whenever a request is made to the session cookie domain. With this setting, consider setting `secure` to `true` to prevent browsers from rejecting the session cookie. For more information, refer to SameSite cookies

Default: `LAX`

> **Note**
>
> For CDSSO, set `"sameSite":"none"` and `"secure":"true"`. For security reasons, many browsers require the connection used by the browser to be secure (HTTPS) for `"sameSite":"none"`. Therefore, if the connection used by the browser is not secure (HTTP), the browser might not supply cookies with `"sameSite":"none"`. For more information, refer to Authenticate with CDSSO.

`"secure"`: *configuration expression<boolean>, optional*

Flag to limit the scope of the session cookie to secure channels.

Set this flag only if the user agent is able to re-emit session cookies over HTTPS on its next hop. For example, to re-emit a session cookie with the `secure` flag, the user agent must be connected to its next hop by HTTPS.

Default: `false`

> **Note**
>
> For CDSSO, set `"sameSite":"none"` and `"secure":"true"`. For security reasons, many browsers require the connection used by the browser to be secure (HTTPS) for `"sameSite":"none"`. Therefore, if the connection used by the browser is not secure (HTTP), the browser might not supply cookies with `"sameSite":"none"`. For more information, refer to Authenticate with CDSSO.

`"timeout"`: *configuration expression<duration>, optional*

The duration after which idle sessions are automatically timed out.

The value must be above zero, and no greater than 3650 days (approximately 10 years). IG truncates the duration of longer values to 3650 days.

Default: 30 minutes

`"streamingEnabled"`: *configuration expression<boolean>, optional*

A flag to manage content:

- `true`: IG streams the content of HTTP requests and responses. The content is available for processing bit-by-bit, as soon as it is received.
- `false`: IG buffers the content of HTTP requests and responses into the storage defined in `temporaryStorage`. The content is available for processing only after it has all been received.

When this property is `true`, consider the following requirements to prevent IG from blocking an executing thread to wait for streamed content:

- Write runtime expressions that consume streamed content with `#` instead of `$`. For more information, refer to runtime expression.
- In scripts and Java extensions, never use a `Promise` blocking method, such as `get()`, `getOrThrow()`, or `getOrThrowUninterruptibly()` to obtain the response. For more information, refer to Scripts.

> **Note**
>
> When `streamingEnabled=true` and a CaptureDecorator with `captureEntity=true` decorates a component, the decorator interrupts streaming for the captured request or response until the whole entity is captured.

Default: `false`

Example configuration files

Default configuration

When your configuration does not include an `admin.json` file, the following `admin.json` is provided by default:

```
{
  "prefix": "openig",
  "connectors": [
    { "port" : 8080 }
  ]
}
```

Overriding the default ApiProtectionFilter

The following example shows an `admin.json` file configured to override the default `ApiProtectionFilter` that protects the reserved administrative route. This example is used in Set up the UMA example.

```
{
  "prefix": "openig",
  "connectors": [
    { "port" : 8080 }
  ],
  "heap": [
    {
      "name": "ClientHandler",
      "type": "ClientHandler"
    },
    {
      "name": "ApiProtectionFilter",
      "type": "CorsFilter",
      "config": {
        "policies": [
          {
            "acceptedOrigins": [ "http://app.example.com:8081" ],
            "acceptedMethods": [ "GET", "POST", "DELETE" ],
            "acceptedHeaders": [ "Content-Type" ]
          }
        ]
      }
    }
  ]
}
```

More information

org.forgerock.openig.http.AdminHttpApplication

GatewayHttpApplication (`config.json`)

The GatewayHttpApplication is the entry point for all incoming gateway requests. It is responsible for initializing a heap of objects, described in Heap objects, and providing the main Handler that receives all the incoming requests.

The configuration is loaded from a JSON-encoded file, expected by default at `$HOME/.openig/config/config.json`. Objects configured in `config.json` can be used by `config.json` and any IG route. They cannot be used by `admin.json`.

If you provide a `config.json`, the IG configuration is loaded from that file. If there is no file, the default configuration is loaded. For the default configuration, and the example `config.json` used in many of the examples in the documentation, refer to the Examples section of this page.

Routes endpoint

The endpoint is defined by the presence and content of `config.json`, as follows:

- When `config.json` is not provided, the routes endpoint includes the name of the main router in the default configuration, `_router`.
- When `config.json` is provided with an unnamed main router, the routes endpoint includes the main router name `router-handler`.
- When `config.json` is provided with a named main router, the routes endpoint includes the provided name or the transformed, URL-friendly name.

Studio deploys and undeploys routes through a main router named `_router`, which is the name of the main router in the default configuration. If you use a custom `config.json`, make sure it contains a main router named `_router`.

Default objects

IG creates objects by default in `config.json`. To override a default object, configure an object with the same name in `config.json`.

Configure default objects in `config.json` and `admin.json` separately. An object configured in `config.json` with the same name as an object configured in `admin.json` is not the same object.

`BaseUriDecorator`

A decorator to override the scheme, host, and port of the existing request URI. The default BaseUriDecorator is named `baseURI`. For more information, refer to BaseUriDecorator.

`AuditService`

Records no audit events. The default AuditService is `NoOpAuditService`. Learn more from NoOpAuditService.

`CaptureDecorator`

Captures requests and response messages. The default CaptureDecorator is named `capture`, and uses the default settings given in CaptureDecorator.

When a capture point for the default CaptureDecorator is defined in a route, for example, when `"capture: "all"` is set as a top-level attribute of the JSON, log messages for requests and responses passing through the route are written to a log file in `$HOME/.openig/logs`.

When no capture point is defined in a route, only exceptions thrown during request or response processing are logged.

By default, request and response contexts and entities are not captured. Do one of the following to capture information:

- Override the default capture decorator declaration, and set `captureEntity` to `true`.
- Declare another CaptureDecorator object with an appropriate configuration and use it at your capture points.

The capture decorator logs information about the HTTP request and response messages, along with their respective headers.

`ClientHandler`

Communicates with third-party services. Learn more from ClientHandler.

`ForgeRockClientHandler`

Sends ForgeRock Common Audit transaction IDs when communicating with protected applications. The default ForgeRockClientHandler is a Chain, composed of a TransactionIdOutboundFilter and a ClientHandler.

`IssuerRepository`

A repository of Issuers declared in the heap. To overwrite the default issuer, configure a local IssuerRepository with the name `IssuerRepository`. To create a new IssuerRepository containing a subset of Issuers, configure a local IssuerRepository with a different name.

`ProxyOptions`

A proxy to which a ClientHandler or ReverseProxyHandler can submit requests, and an AmService can submit Websocket notifications. For more information, refer to ProxyOptions.

`ReverseProxyHandler`

Communicates with third-party services. For more information, refer to ReverseProxyHandler.

`ScheduledExecutorService`

Specifies the number of threads in a pool.

`SecretsService` *(deprecated)*

Manages a store of secrets from files, system properties, and environment variables, by using Commons Secrets API. The default SecretsService is a SystemAndEnvSecretStore with the default configuration. For more information, refer to Secrets.

`TemporaryStorage`

Manages temporary buffers. For more information, refer to TemporaryStorage.

`TimerDecorator`

Records time spent within filters and handlers. The default TimerDecorator is named `timer`. For more information, refer to TimerDecorator.

`TransactionIdOutboundFilter`

Inserts the ID of a transaction into the header of a request.

Sessions

When the heap is configured with a JwtSession object named `Session`, the object is used as the default session producer. Stateless sessions are created for all requests.

When a JwtSession is not configured for a request, session information is stored in the IG cookie, called by default `IG_SESSIONID`.

For more information, refer to Sessions and JwtSession.

Usage

```
{
  "handler": Handler reference,
  "heap": [ object, ... ],
  "properties": object,
  "temporaryStorage": TemporaryStorage reference
}
```

Properties

`"handler"`: *Handler reference, required*

The Handler to which IG dispaches requests.

Provide the name of a Handler object defined in the heap or an inline Handler configuration object.

`"heap"`: *array of objects, optional*

The heap object configuration, described in Heap objects.

`"properties"`: *object, optional*

Configuration parameters declared as property variables for use in the configuration. See also Route properties.

Default: Null

`"temporaryStorage"`: *TemporaryStorage reference, optional*

The TemporaryStorage object to buffer content during processing.

Provide the name of a TemporaryStorage object defined in the heap or an inline TemporaryStorage configuration object.

Incoming requests use the temporary storage buffer as follows:

- Used only when `streamingEnabled` is `false`.
- The request is loaded into the IG storage defined in `temporaryStorage`, before it enters the chain.
- If the content length of the request is more than the buffer limit, IG returns an `HTTP 413 Payload Too Large`.

Default: Use the heap object named TemporaryStorage. Otherwise, use an internally-created TemporaryStorage object named `TemporaryStorage` that uses default settings for a TemporaryStorage object.

Example configuration files

Default configuration

When your configuration does not include a `config.json` file, the following configuration is provided by default.

```
{
    "heap": [
        {
            "name": "_router",
            "type": "Router",
            "config": {
                "scanInterval": "&{ig.router.scan.interval|10 seconds}",
                "defaultHandler": {
                    "type": "DispatchHandler",
                    "config": {
                        "bindings": [
                            {
                                "condition": "${request.method == 'GET' and request.uri.path == '/'}",
                                "handler": {
                                    "type": "WelcomeHandler"
                                }
                            },
                            {
                                "condition": "${request.uri.path == '/'}",
                                "handler": {
                                    "type": "StaticResponseHandler",
                                    "config": {
                                        "status": 405,
                                        "reason": "Method Not Allowed"
                                    }
                                }
                            },
                            {
                                "handler": {
                                    "type": "StaticResponseHandler",
                                    "config": {
                                        "status": 404,
                                        "reason": "Not Found"
                                    }
                                }
                            }
                        ]
                    }
                }
            }
        }
    ],
    "handler": "_router"
}
```

Notice the following features of the default configuration:

- The handler contains a main router named `_router`. When IG receives an incoming request, `_router` routes the request to the first route in the configuration whose condition is satisfied.
- If the request doesn't satisfy the condition of any route, it is routed to the defaultHandler. If the request is to access the IG welcome page, IG dispatches the request. Otherwise, IG returns an HTTP status 404 (Resource not found), because the requested resource does not exist.

Example config.json used in the doc

The following example of `config.json` is used in many of the examples in the documentation:

```
{
  "handler": {
    "type": "Router",
    "name": "_router",
    "baseURI": "http://app.example.com:8081",
    "capture": "all"
  },
  "heap": [
    {
      "name": "JwtSession",
      "type": "JwtSession"
    },
    {
      "name": "capture",
      "type": "CaptureDecorator",
      "config": {
        "captureEntity": true,
        "_captureContext": true
      }
    }
  ]
}
```

Notice the following features of the file:

- The handler contains a main router named `_router`. When IG receives an incoming request, `_router` routes the request to the first route in the configuration whose condition is satisfied.
- The `baseURI` changes the request URI to point the request to the sample application.
- The `capture` captures the body of the HTTP request and response.
- The JwtSession object in the heap can be used in routes to store the session information as JSON Web Tokens (JWT) in a cookie. For more information, refer to JwtSession.

More information

org.forgerock.openig.http.GatewayHttpApplication

Heap objects

An array of objects created and initialized by heaplet objects. Learn more about the configuration in Inline and heap objects.

Usage

```
{
  "heap": [
    {
      "name": string,
      "type": string,
      "config": {
        type-specific configuration
      }
    }
  ]
}
```

Properties

`"name"`: *string*, *required*

A unique name for an object in the heap.

Routes and other configurations must refer to heap objects by their `name` property.

`"type"`: *string*, *required*

The class name of the heap object.

`"config"`: *object*, *required unless all the fields are optional and the configuration uses only default settings*

The configuration of the heap object. The object configuration must conform to the object class.

More information

org.forgerock.openig.heap.Heap

Configuration settings

Filters, handlers, and other objects whose configuration settings are defined by strings, integers, or booleans, can alternatively be defined by expressions that match the expected type.

For information about expressions, refer to Expressions.

System properties

The following properties are supported in IG. Their names have a special meaning in IG, and they should be used only for their stated purpose:

`ig.instance.dir`, `IG_INSTANCE_DIR`

The full path to the directory containing configuration and data for the IG instance.

Default: Linux, `$HOME/.openig`; Windows, %appdata%\OpenIG

For information about how to use a different location, refer to Configuration location.

`org.forgerock.json.jose.jwe.compression.max.decompressed.size.bytes`

The maximum size in bytes to which a compressed JWT can be decompressed.

Default: 32 KBytes

`org.forgerock.secrets.preferDeterministicEcdsa`

When this property is `true`, and the following conditions are met, JWTs are signed with a deterministic Elliptic Curve Digital Signature Algorithm (ECDSA):

- ECDSA is used for signing
- Bouncy Castle is installed

Default: `true`

`org.forgerock.http.TrustTransactionHeader`

When this property is `true`, all incoming `X-ForgeRock-TransactionId` headers are trusted. Monitoring or reporting systems that consume the logs can allow requests to be correlated as they traverse multiple servers.

Default: `false`

`org.forgerock.http.util.ignoreFormParamDecodingError`

When this property is `true`, form encoding errors caused by invalid characters are ignored, and encoded values are used instead.

Default: `false`

`org.forgerock.json.jose.jwe.compression.max.decompressed.size.bytes`

The maximum size in bytes to which a compressed JWT can be decompressed.

Default: 32 KBytes

Handlers

Handler objects process a request and context, and return a response. The way the response is created depends on the type of handler.

Chain

Dispatches a request and context to an ordered list of filters, and then finally to a handler.

Filters process the incoming request and context, pass it on to the next filter, and then to the handler. After the handler produces a response, the filters process the outgoing response and context as it makes its way to the client. Note that the same filter can process both the incoming request and the outgoing response but most filters do one or the other.

A Chain can be placed in a configuration anywhere that a handler can be placed.

Unlike ChainOfFilters, Chain finishes by dispatching the request to a handler. For more information, refer to ChainOfFilters.

Usage

```
{
     "name": string,
     "type": "Chain",
     "config": {
         "filters": [ Filter reference, ... ],
         "handler": Handler reference
     }
}
```

Properties

`"filters"`: *array of Filter references, required*

An array of names of filter objects defined in the heap, and inline filter configuration objects.

The chain dispatches the request to these filters in the order they appear in the array.

See also Filters.

`"handler"`: *Handler reference, required*

The Handler to which the Chain dispatches the request after it has traversed the specified filters.

Provide the name of a Handler object defined in the heap or an inline Handler configuration object.

Example

```
{
     "name": "LoginChain",
     "type": "Chain",
     "config": {
         "filters": [ "LoginFilter" ],
         "handler": "ReverseProxyHandler"
     }
}
```

More information

org.forgerock.openig.filter.ChainHandlerHeaplet

ClientHandler

Sends requests to third-party services that are accessible through HTTP, and reconstructs the response from the received bytes. A third-party service is one that IG calls for data, such as an HTTP API or AM, or one to which IG submits data. When IG relays a request to a third-party service, IG is acting as a client of the service. IG is *client-side*.

Consider the following comparison of the ClientHandler and ReverseProxyHandler:

	ClientHandler	ReverseProxyHandler
Use this handler to ...	Send requests to third-party services accessed within a route. The service can be AM or an HTTP API. The service can be an HTTP endpoint, such as AM, IDM, Identity Cloud, or any custom HTTP API.	Send requests to the final service accessed by a route. The service can be the final downstream application.
If the service does not respond in time, this handler ...	Propagates the error through the Promise flow. If the error is not handled within the route, for example, by a FailureHandler, the handler returns a `500 Internal Server Error` response.	Stops processing the request, and returns a `502 Bad Gateway` response.

When uploading or downloading large files, prevent timeout issues by increasing the value of `soTimeout`, and using a streaming mode, as follows:

Configure the `streamingEnabled` property of AdminHttpApplication.

Usage

```
{
  "name": string,
  "type": "ClientHandler",
  "config": {
    "vertx": object,
    "connections": configuration expression<number>,
    "waitQueueSize": configuration expression<number>,
    "soTimeout": configuration expression<duration>,
    "connectionTimeout": configuration expression<duration>,
    "protocolVersion": configuration expression<enumeration>,
    "http2PriorKnowledge": configuration expression<boolean>,
    "proxyOptions": ProxyOptions reference,
    "temporaryStorage": TemporaryStorage reference,
    "tls": ClientTlsOptions reference,
    "retries": object,
    "circuitBreaker": object,
    "hostnameVerifier": configuration expression<enumeration>, //deprecated
    "proxy": Server reference, //deprecated
    "systemProxy": boolean //deprecated
  }
}
```

* Legacy; no longer supported

Properties

`"vertx"`: *object*, *optional*

Vert.x-specific configuration for the handler when IG is *client-side*. When IG is acting server-side, configure the `connectors:vertx` property of admin.json.

> **Note**
>
> When IG sends requests to a proxied application or requests services from a third-party application, IG is client-side. IG is acting as a client of the application, and the application is acting as a server.

Vert.x options are described in HttpClientOptions.

The `vertx` object is read as a map, and values are evaluated as configuration expressions.

For properties where IG provides its own first-class configuration, Vert.x configuration options are disallowed, and the IG configuration option takes precedence over Vert.x options configured in `vertx`. The following Vert.x configuration options are disallowed client-side:

- `alpnVersions`
- `connectTimeout`
- `enabledCipherSuites`
- `enabledSecureTransportProtocols`
- `http2ClearTextUpgrade`
- `idleTimeout`
- `idleTimeoutUnit`
- `keyCertOptions`
- `keyStoreOptions`
- `maxWaitQueueSize`
- `pemKeyCertOptions`
- `pemTrustOptions`
- `pfxKeyCertOptions`
- `pfxTrustOptions`
- `port`
- `protocolVersion`
- `proxyOptions`
- `ssl`
- `trustOptions`
- `trustStoreOptions`
- `useAlpn`
- `verifyHost`

The following example configures the Vert.x configuration when IG is acting client-side. When IG is acting server-side, configure the `connectors:vertx` property of `admin.json`:

```
{
  "vertx": {
    "maxWebSocketFrameSize": 128000,
    "maxWebSocketMessageSize": 256000,
    "compressionLevel": 4,
    "maxHeaderSize": 16384
  }
}
```

The following example configures HTTP/2 connections when IG is acting client-side. The configuration allows IG to make HTTP/2 requests with large headers. When IG is acting server-side, configure the `connectors:vertx` property of `admin.json`:

```
{
  "vertx": {
    "initialSettings": {
      "maxHeaderListSize": 16384
    }
  }
}
```

`"connections"`: *configuration expression<number>, optional*

The maximum number of concurrent HTTP connections in the client connection pool.

For information about the interaction between this property and `waitQueueSize`, see the description of `waitQueueSize`.

Default: 64

"waitQueueSize": configuration expression<number>, optional

The maximum number of outbound requests allowed to queue when no downstream connections are available. Outbound requests received when the queue is full are rejected.

Use this property to limit memory use when there is a backlog of outbound requests, for example, when the protected application or third-party service is slow.

Configure `waitQueueSize` as follows:

- Not set (default): The wait queue is calculated as the square of `connections`.
  - If `connections` is not configured, then its default of `64` is used, giving the `waitQueueSize` of `4096`.
  - If the square of `connections` exceeds the maximum integer value for the Java JVM, the maximum integer value for the Java JVM is used.
- `-1`: The wait queue is unlimited. Requests received when there are no available connections are queued without limit.
- `0`: There is no wait queue. Requests received when there are no available connections are rejected.

- A value that is less than the square of `connections`:

  When the configuration is loaded, the configured value is used. IG generates a warning that the `waitQueueSize` is too small for the `connections` size, and recommends a different value.

- A value where `waitQueueSize` plus `connections` exceeds the maximum integer value for the Java JVM:

  When the configuration is loaded, the `waitQueueSize` is reduced to the maximum integer value for the Java JVM minus the value of `connections`. IG generates a warning.

Consider the following example configuration of `connections` and `waitQueueSize`:

```
{
  "handler" : {
    "name" : "proxy-handler",
    "type" : "ReverseProxyHandler",
    "MyCapture" : "all",
    "config": {
      "soTimeout": "10 seconds",
      "connectionTimeout": "10 seconds",
      "connections": 64,
      "waitQueueSize": 100
    }
  },
  "baseURI" : "http://app.example.com:8080",
  "condition" : "${find(request.uri.path, '/')}"
}
```

IG can propagate the request to the sample application using 64 connections. When the connections are consumed, up to 100 subsequent requests are queued until a connection is freed. Effectively IG can accommodate 164 requests, although user concurrency delay means more may be handled. Requests received when the waitQueue is full are rejected.

Default: Not set

`"connectionTimeout"`: *configuration expression<duration>, optional*

Time to wait to establish a connection, expressed as a duration

Default: 10 seconds

`"protocolVersion"`: *configuration expression<enumeration>, optional*

The version of HTTP protocol to use when processing requests:

- `HTTP/2`:
  - For HTTP, process requests using HTTP/1.1.
  - For HTTPS, process requests using HTTP/2.
- `HTTP/1.1`:
  - For HTTP and HTTPS, process requests using HTTP/1.1.

- Not set:
  - For HTTP, process requests using HTTP/1.1.
  - For HTTPS with `alpn` enabled in ClientTlsOptions, process requests using HTTP/1.1, with an HTTP/2 upgrade request. If the targeted server can use HTTP/2, the client uses HTTP/2.

    For HTTPS with `alpn` disabled in ClientTlsOptions, process requests using HTTP/1.1, without an HTTP/2 upgrade request.

    Note that `alpn` is enabled by default in ClientTlsOptions.

Default: Not set

> **Note**
>
> In HTTP/1.1 request messages, a `Host` header is required to specify the host and port number of the requested resource. In HTTP/2 request messages, the `Host` header is not available.
> In scripts or custom extensions that use HTTP/2, use `UriRouterContext.originalUri.host` or `UriRouterContext.originalUri.port` in requests.

`"http2PriorKnowledge"`: *configuration expression<boolean>, optional*

A flag for whether the client should have prior knowledge that the server supports HTTP/2. This property is for cleartext (non-TLS requests) only, and is used only when `protocolVersion` is HTTP/2.

- `false`: The client checks whether the server supports HTTP/2 by sending an HTTP/1.1 request to upgrade the connection to HTTP/2:
  - If the server supports HTTP/2, the server upgrades the connection to HTTP/2, and subsequent requests are processed over HTTP/2.
  - If the server does not support HTTP/2, the connection is not upgraded, and subsequent requests are processed over HTTP/1.
- `true`: The client does not check that the server supports HTTP/2. The client sends HTTP/2 requests to the server, assuming that the server supports HTTP/2.

Default: `false`

`"proxyOptions"`: *ProxyOptions reference, optional*

A proxy server to which requests can be submitted. Use this property to relay requests to other parts of the network. For example, use it to submit requests from an internal network to the internet.

Provide the name of a ProxyOptions object defined in the heap or an inline configuration.

Default: A heap object named `ProxyOptions`.

`"soTimeout"`: *configuration expression<duration>, optional*

Socket timeout, after which stalled connections are destroyed, expressed as a duration.

> **Tip**
>
> If `SocketTimeoutException` errors occur in the logs when you try to upload or download large files, consider increasing `soTimeout`.

Default: 10 seconds

`"temporaryStorage"`: *TemporaryStorage reference, optional*

The TemporaryStorage object to buffer the request and response, when the `streamingEnabled` property of admin.json is `false`.

Default: A heap object named `TemporaryStorage`.

`tls`: *ClientTlsOptions reference, optional*

> **Important**
>
> Use of a TlsOptions reference is deprecated; use ClientTlsOptions instead. For more information, refer to the Deprecated section of the *Release Notes*.

Configure options for connections to TLS-protected endpoints, based on ClientTlsOptions. Define the object inline or in the heap.

Default: Connections to TLS-protected endpoints are not configured.

`"retries"`: *object, optional*

Enable and configure retry for requests.

During the execution of a request to a remote server, if a runtime exception occurs, or a condition is met, IG waits for a delay, and then schedules a new execution of the request. IG tries until the allowed number of retries is reached or the execution succeeds.

A warning-level entry is logged if all retry attempts fail; a debug-level entry is logged if a retry succeeds.

```
"retries": {
  "enabled": configuration expression<boolean>,
  "condition": runtime expression<boolean>,
  "executor": ScheduledExecutorService reference,
  "count": configuration expression<number>,
  "delay": configuration expression<duration>,
  }
}
```

`"enabled"`: *configuration expression<boolean>, optional*

Enable retries.

Default: `true`

`"condition"`: *runtine expression<boolean>, optional*

An inline IG expression to define a condition based on the response, such as an error code.

The condition is evaluated as follows:

- If `true`, IG retries the request until the value in `count` is reached.
- If `false`, IG retries the request only if a runtime exception occurs, until the value in `count` is reached.

Default: `${false}`

`"executor"`: *ScheduledExecutorService reference, optional*

The ScheduledExecutorService to use for scheduling delayed execution of the request.

Default: `ScheduledExecutorService`

See also ScheduledExecutorService.

`"count"`: *configuration expression<number>, optional*

The maximum number of retries to perform. After this threshold is passed and if the request is still not successful, then the ClientHandler propagates the failure.

Retries caused by any runtime exception or triggered condition are included in the count.

Default: `5`

`"delay"`: *_configuration expression<duration>, optional*

The time to wait before retrying the request.

After a failure to send the request, if the number of retries is below the threshold, a new attempt is scheduled with the executor service after this delay.

Default: `10 seconds`

The following example configures a retry when a downstream component returns a `502 Bad Gateway` response code:

```
"retries": {
    "enabled": true,
    "condition": "${response.status.code == 502}"
}
```

The following example configures the handler to retry the request only once, after a 1-minute delay:

```
{
  "retries": {
    "count": 1,
    "delay": "1 minute"
  }
}
```

The following example configures the handler to retry the request at most 20 times, every second:

```
{
  "retries": {
    "count": 20,
    "delay": "1 second"
  }
}
```

The following example configures the handler to retry the request 5 times, every 10 seconds (default values), with a dedicated executor:

```
{
  "retries": {
    "executor": {
      "type": "ScheduledExecutorService",
      "config": {
        "corePoolSize": 20
      }
    }
  }
}
```

`"circuitBreaker"`: *object*, *optional*

Enable and configure a circuit breaker to trip when the number of failures exceeds a configured threshold. Calls to downstream services are stopped, and a runtime exception is returned. The circuit breaker is reset after the configured delay.

```
{
  "circuitBreaker": {
    "enabled": configuration expression<boolean>,
    "maxFailures":  configuration expression<integer>,
    "openDuration": configuration expression<duration>,
    "openHandler": Handler reference,
    "slidingCounter":  object,
    "executor":  ScheduledExecutorService reference
  }
}
```

`"enabled"`: *configuration expression<boolean>, optional*

A flag to enable the circuit breaker.

Default: `true`

`"maxFailures"`: *configuration expression<number>, required*

The maximum number of failed requests allowed in the window given by `size`, before the circuit breaker trips. The value must be greater than zero.

Important

When `retries` is set, the circuit breaker does not count retried requests as failures. Bear this in mind when you set `maxFailures`.
In the following example, a request can fail and then be retried three times. If it fails the third retry, the request has failed four times, but the circuit breaker counts only one failure.

```
{
  "retries": {
    "count": 3,
    "delay": "1 second"
  }
}
```

`"openDuration"`: *configuration expression<duration>, required*

The duration for which the circuit stays open after the circuit breaker trips. The `executor` schedules the circuit to be closed after this duration.

`"openHandler"`: *Handler reference, optional*

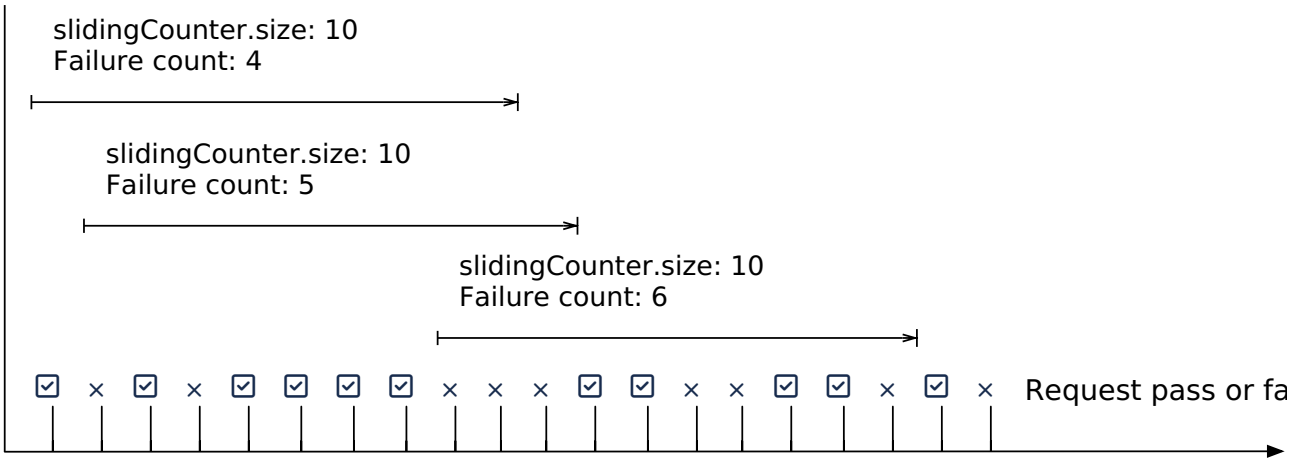
The Handler to call when the circuit is open.

Default: A handler that throws a RuntimeException with a "circuit-breaker open" message.

`"slidingCounter"`: *object, optional*

A sliding window error counter. The circuit breaker trips when the number of failed requests in the number of requests given by `size` reaches `maxFailures`.

The following image illustrates how the sliding window counts failed requests:




```
{
  "slidingCounter":  {
    "size": configuration expression<number>
  }
}
```

`"size"`: *configuration expression<number>, required*

The size of the sliding window in which to count errors.

The value of `size` must be greater than zero, and greater than the value of `maxFailures`, otherwise an exception is thrown.

`"executor"`: *ScheduledExecutorService reference, optional*

A ScheduledExecutorService to schedule closure of the circuit after the duration given by `openDuration`.

Default: The default ScheduledExecutorService in the heap

`"hostnameVerifier"`: *configuration expression<enumeration>, optional*

> **Important**
>
> This property is deprecated; use the `tls` property instead to configure ClientTlsOptions. For more information, refer to the Deprecated section of the *Release Notes*.

The way to handle hostname verification for outgoing SSL connections. Use one of the following values:

- `ALLOW_ALL`: Allow a certificate issued by a trusted CA for any hostname or domain to be accepted for a connection to any domain.

  This setting allows a certificate issued for one company to be accepted as a valid certificate for another company. To prevent the compromise of TLS connections, use this setting in development mode only. In production, use `STRICT`.

- `STRICT`: Match the hostname either as the value of the the first CN, or any of the subject-alt names.

  A wildcard can occur in the CN, and in any of the subject-alt names. Wildcards match one domain level, so `*.example.com` matches `www.example.com` but not `some.host.example.com`.

Default: `STRICT`

`"proxy"`: *Server reference, optional*

> **Important**
>
> This property is deprecated; use `proxyOptions` instead. For more information, refer to the Deprecated section of the *Release Notes*.

A proxy server to which requests can be submitted. Use this property to relay requests to other parts of the network. For example, use it to submit requests from an internal network to the internet.

If both `proxy` and `systemProxy` are defined, `proxy` takes precedence.

```
"proxy" : {
  "uri": configuration expression<uri string>,
  "username": configuration expression<string>,
  "passwordSecretId": configuration expression<secret-id>,
  "secretsProvider": SecretsProvider reference
}
```

`"uri"`: *configuration expression<uri string>, required*

URI of a server to use as a proxy for outgoing requests.

The result of the expression must be a string that represents a valid URI, but is not a real java.net.URI object.

`"username"`: *configuration expression<string>, required if the proxy requires authentication*

Username to access the proxy server.

`"passwordSecretId"`: *configuration expression<secret-id>, required if the proxy requires authentication*

The secret ID of the password to access the proxy server.

This secret ID must point to a GenericSecret.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for the proxy's password.

`"systemProxy"`: *boolean, optional*

> **Important**
>
> This property is deprecated; use `proxyOptions` instead. For more information, refer to the Deprecated section of the *Release Notes*.

Submit outgoing requests to a system-defined proxy, set by the following system properties or their HTTPS equivalents:

- `http.proxyHost`, the host name of the proxy server.
- `http.proxyPort`, the port number of the proxy server. The default is `80`.
- `http.nonProxyHosts`, a list of hosts that should be reached directly, bypassing the proxy.

This property can't be used with a proxy that requires a username and password. Use the property `proxy` instead.

If both `proxy` and `systemProxy` are defined, `proxy` takes precedence.

For more information, refer to Java Networking and Proxies.

Default: False.

`"keyManager"`: *Key manager reference(s), optional*

> **Important**
>
> This property is deprecated; use the `tls` property instead to configure ClientTlsOptions. For more information, refer to the Deprecated section of the *Release Notes*.

The key manager(s) that handle(s) this client's keys and certificates.

The value of this field can be a single reference, or an array of references.

Provide either the name(s) of key manager object(s) defined in the heap, or specify the configuration object(s) inline.

You can specify either a single key manager, as in `"keyManager": "MyKeyManager"`, or an array of key managers, as in `"keyManager": [ "FirstKeyManager", "SecondKeyManager" ]`.

If you do not configure a key manager, then the client cannot present a certificate, and so cannot play the client role in mutual authentication.

`"sslCipherSuites"`: *array of strings, optional*

> **Important**
>
> This property is deprecated; use the `tls` property instead to configure ClientTlsOptions. For more information, refer to the Deprecated section of the *Release Notes*.

Array of cipher suite names, used to restrict the cipher suites allowed when negotiating transport layer security for an HTTPS connection.

For information about the available cipher suite names, refer to the documentation for the Java virtual machine (JVM) where you run IG. For Oracle Java, refer to the list of JSSE Cipher Suite Names.

Default: Allow any cipher suite supported by the JVM.

`"sslContextAlgorithm"`: *string, optional*

> **Important**
>
> This property is deprecated; use the `tls` property instead to configure ClientTlsOptions. For more information, refer to the Deprecated section of the *Release Notes*.

The SSLContext algorithm name, as listed in the table of SSLContext Algorithms for the Java Virtual Machine used by IG.

Default: TLS

`"sslEnabledProtocols"`: *array of strings, optional*

> **Important**
>
> This property is deprecated; use the `tls` property instead to configure ClientTlsOptions. For more information, refer to the Deprecated section of the *Release Notes*.

Array of protocol names, used to restrict the protocols allowed when negotiating transport layer security for an HTTPS connection.

Default: Allow any protocol supported by the JVM.

`"trustManager"`: *Trust manager reference(s), optional*

> **Important**
>
> This property is deprecated; use the `tls` property instead to configure ClientTlsOptions. For more information, refer to the Deprecated section of the *Release Notes*.

The trust managers that handle(s) peers' public key certificates.

The value of this field can be a single reference, or an array of references.

Provide either the name(s) of trust manager object(s) defined in the heap, or specify the configuration object(s) inline.

You can specify either a single trust manager, as in `"trustManager": "MyTrustManager"`, or an array of trust managers, as in `"trustManager": [ "FirstTrustManager", "SecondTrustManager" ]`.

If you do not configure a trust manager, then the client uses only the default Java truststore. The default Java truststore depends on the Java environment. For example, `$JAVA_HOME/lib/security/cacerts`.

More information

org.forgerock.openig.handler.ClientHandlerHeaplet

DispatchHandler

When a request is handled, the first condition in the list of conditions is evaluated. If the condition expression yields `true`, the request is dispatched to the associated handler with no further processing. Otherwise, the next condition in the list is evaluated.

Usage

```
{
  "name": string,
  "type": "DispatchHandler",
  "config": {
    "bindings": [
      {
        "condition": runtime expression<boolean>,
        "handler": Handler reference,
        "baseURI": runtime expression<url>,
      }, ...
    ]
  }
}
```

Properties

`"bindings"`: *array of objects, required*

One or more condition and handler bindings.

`"condition"`: *runtime expression<boolean>, optional*

A flag to indicate that a condition is met. The condition can be based on the request, context, or IG runtime environment, such as system properties or environment variables.

Conditions are defined using IG expressions, as described in Expressions, and are evaluated as follows:

- `true`: The request is dispatched to the associated handler.
- `false`: The next condition in the list is evaluated.

For examples, refer to Example conditions and requests.

Default: `${true}`

`"handler"`: *Handler reference, required*

The Handler to which IG dispaches the request if the associated condition yields `true`.

Provide the name of a Handler object defined in the heap or an inline Handler configuration object.

`"baseURI"`: *runtime expression<url>,optional*

A base URI that overrides the existing request URI. Only scheme, host, and port are used in the supplied URI.

The result of the expression must be a string that represents a valid URI, but is not a real `java.net.URI` object. For example, it would be incorrect to use `${request.uri}`, which is not a String but a MutableUri.

In the following example, the binding condition looks up the hostname of the request. If it finds a match, the value is used for the `baseURI`. Otherwise, the default value is used:

```
{
  "properties": {
    "uris": {
      "app1.example.com": {
        "baseURI": "http://backend1:8080/"
      },
      "app2.example.com": {
        "baseURI": "http://backend2:8080/"
      },
      "default": {
        "baseURI": "http://backend3:8080/"
      }
    }
  },
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${not empty uris[contexts.router.originalUri.host]}",
          "baseURI": "${uris[contexts.router.originalUri.host].baseURI}",
          "handler": "ReverseProxyHandler"
        },
        {
          "baseURI": "${uris['default'].baseURI}",
          "handler": "ReverseProxyHandler"
        }
      ]
    }
  }
}
```

Default: No change to the base URI

Example

For an example that uses a DispatchHandler, refer to Implement not-enforced URIs with a DispatchHandler

More information

org.forgerock.openig.handler.DispatchHandler

Expressions

ForgeRockClientHandler

The ForgeRockClientHandler is a Handler available by default on the heap that chains a default ClientHandler with a TransactionIdOutboundFilter.

This Handler supports ForgeRock audit by supporting the initiation or propagation of audit information from IG to the audit framework. For more information, see AuditService.

The following default ForgeRockClientHandler is available as a default object on the heap, and can be referenced by the name ForgeRockClientHandler.

```
{
     "name": "ForgeRockClientHandler",
     "type": "Chain",
     "config": {
         "filters": [ "TransactionIdOutboundFilter" ],
         "handler": "ClientHandler"
     }
}
```

Example

For an example that uses ForgeRockClientHandler to log interactions between IG and AM, see Decorating IG's interactions with AM.

More information

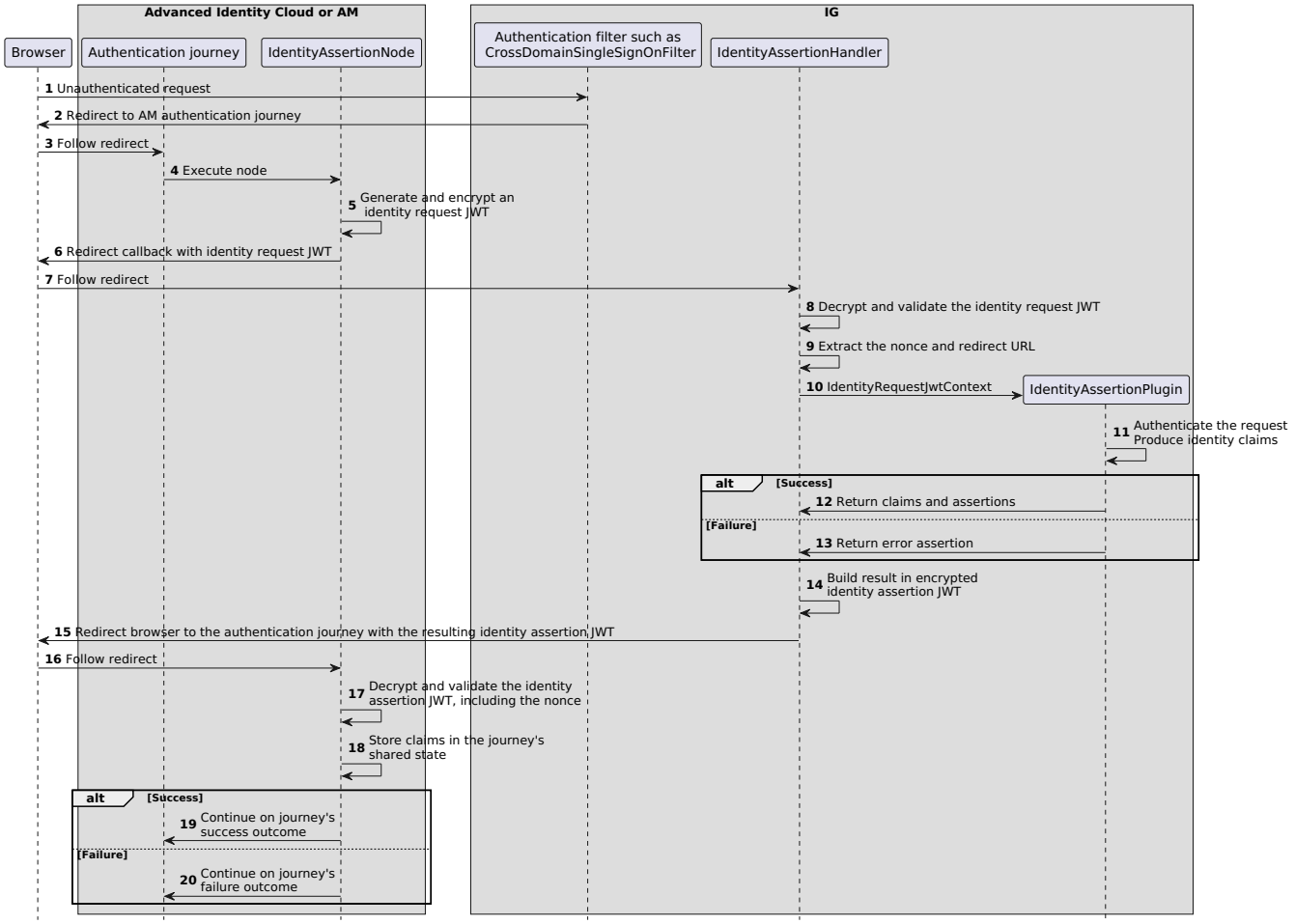
org.forgerock.openig.heap.Keys

IdentityAssertionHandler

Use in an Identity Cloud authentication journey with the IdentityAssertionNode node.

This handler replaces IdentityAssertionHandlerTechPreview designed for the Gateway Communication node described in Identity Cloud's Gateway Communication overview.

The following image shows the flow of information when an Identity Assertion node authenticates internal accesses:



As part of an Identity Cloud journey, the IdentityAssertionHandler uses an identityAssertionPlugin to manage local authentication as follows:

1. The Identity Cloud authentication journey redirects a user to IG for local authentication, providing an ***identity request JWT***.
2. IG validates the identity request JWT.
3. The identityAssertionPlugin accesses the IdentityRequestJwtContext generated from the identity request JWT. It then performs local processing and returns the principal and identity claims in an ***identity assertion JWT***.
4. IG redirects the user back to Identity Cloud authentication journey, providing the identity assertion JWT. If an exception prevents IG from returning a valid identity assertion JWT, IG returns an HTTP 500.

The following table lists the claims contained in identity request JWT and identity assertion JWT:

Claim	Description	Identity request JWT	Identity assertion JWT (succesful plugin processing)	Identity assertion JWT (plugin processing error)
`iss`	Issuer	✓	✓	✓
`aud`	Audience	✓	✓	✓

Claim	Description	Identity request JWT	Identity assertion JWT (succesful plugin processing)	Identity assertion JWT (plugin processing error)
`iat`	Issued at	✓	✓	✓
`exp`	Expiration time	✓	✓	✓
`nonce`	Unique ID generated by the IdentityGatewayAssertionNode and returned in the identity assertion JWT	✓	✓	✓
`redirect`	URL on which to send the identity assertion JWT	✓	✗	✗
`version`	JWT version; only v1 is supported	✓	✗	✗
`data`	Map of claims items that can be required by a plugin	Optional	✗	✗
`principal`	The user for whom the identity assertion JWT is issued	✗	✓	✗
`identity`	Map of additional identity claims returned by the plugin	✗	✓	✗
`error`	Error message of the plugin processing failure	✗	✗	✓

Usage

```
{
  "name": string,
  "type": "IdentityAssertionHandler",
  "config": {
    "identityAssertionPlugin": IdentityAssertionPlugin reference,
    "selfIdentifier": configuration expression<string>,
    "peerIdentifier": configuration expression<string>,
    "encryptionSecretId": configuration expression<secret-id>,
    "secretsProvider": Secrets Provider reference,
    "expiry": configuration expression<duration>,
    "skewAllowance": configuration expression<duration>
  }
}
```

`"identityAssertionPlugin"`: *configuration expression<string>, required*

An implementation of org.forgerock.openig.assertion.plugin.IdentityAssertionPlugin.

This plugin is called after the IdentityAssertionHandler validates the identity request JWT from Identity Cloud. The handler then passes the IdentityRequestJwtContext in the context chain to the plugin.

For an out-of-the-box plugin to support use-cases that aren't already provisioned by an IG plugin, refer to ScriptableIdentityAssertionPlugin.

`"selfIdentifier"`: *configuration expression<string>, required*

An identifier to validate that this IG instance is the correct audience for the identity request from Identity Cloud.

This identifier is the value of:

- `aud` claim in the identity request JWT
- `iss` claim in the identity assertion JWT

Can't be null.

`"peerIdentifier"`: *configuration expression<string>, required*

An identifier to validate that the expected Identity Cloud instance issued the identity request.

This identifier is the value of the:

- `iss` claim in the identity request JWT
- `aud` claim in the identity assertion JWT

Can't be null.

`"encryptionSecretId"`: *configuration expression<secret-id>, required*

The secret ID for the secret to decrypt the identity request JWT and encrypt the returned identity assertion JWT. The secret ID must point to a CryptoKey. Decryption and encryption is with AES GCM using a 256-bit key.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to resolve encrytion and decryption keys.

`"expiry"`: *_configuration expression<duration>, optional*

The expiry time of the identity assertion JWT.

Default: 30 seconds

`"skewAllowance"`: *configuration expression<duration>, optional*

The duration to add to the validity period of a JWT to allow for clock skew between different servers.

A `skewAllowance` of 2 minutes affects the validity period as follows:

- A JWT with an `iat` of 12:00 is valid from 11:58 on the IG clock.
- A JWT with an `exp` 13:00 is expired after 13:02 on the IG clock.

Default: To support a zero-trust policy, the skew allowance is by default `zero`.

Example

The following route is an Identity Assertion service route for use with the IdentityAssertionNode.

```
{
  "name": "IdentityAssertion",
  "condition": "${find(request.uri.path, '^/idassert')}",
  "properties": {
    "amIdcPeer": "myTenant.forgeblocks.com"
  },
  "handler": "IdentityAssertionHandler-1",
  "heap": [
    {
      "name": "IdentityAssertionHandler-1",
      "type": "IdentityAssertionHandler",
      "config": {
        "identityAssertionPlugin": "BasicAuthScriptablePlugin",
        "selfIdentifier": "https://ig.ext.com:8443",
        "peerIdentifier": "&{amIdcPeer}",
        "secretsProvider": [
          "secrets-pem"
        ],
        "encryptionSecretId": "idassert"
      }
    },
    {
      "name": "BasicAuthScriptablePlugin",
      "type": "ScriptableIdentityAssertionPlugin",
      "config": {
        "type": "application/x-groovy",
        "source": [
          "import org.forgerock.openig.assertion.IdentityAssertionClaims",
          "import org.forgerock.openig.assertion.plugin.IdentityAssertionPluginException",
          "logger.info('Running ScriptableIdentityAssertionPlugin')",
          "return new IdentityAssertionClaims('demo')"
        ]
      }
    },
    {
      "name": "pemPropertyFormat",
      "type": "PemPropertyFormat"
    },
    {
      "name": "secrets-pem",
      "type": "FileSystemSecretStore",
      "config": {
        "directory": "&{ig.instance.dir}/secrets/igfs",
        "suffix": ".pem",
        "format": "pemPropertyFormat",
        "mappings": [
          {
            "secretId": "idassert",
            "format": "pemPropertyFormat"
          }
        ]
      }
    }
  ]
}
```

More information

org.forgerock.openig.assertion.plugin.IdentityAssertionPlugin

IdentityAssertionHandlerTechPreview

> **Important**
>
> The IdentityAssertionHandlerTechPreview, ScriptableIdentityAssertionPluginTechPreview, and IdentityAssertionPluginTechPreview are available in Technology preview. They aren't yet supported, may be functionally incomplete, and are subject to change without notice.

Use in an Identity Cloud authentication journey with the Gateway Communication node. described in Identity Cloud's Gateway Communication overview.

The IdentityAssertionHandlerTechPreview sets up an IdentityAssertionPluginTechPreview to manage local processing, such as authentication. The Handler then calls the plugin at runtime for each request.

An Identity Cloud authentication journey does the following:

- Redirects users to IG for local authentication.
- After local authentication, provides an identity assertion and redirects users back to the Identity Cloud authentication journey.

The Identity Cloud authentication journey provides:

- A cryptographically-secure random value in a nonce to validate the identity assertion.
- A `returnUri` to redirect the user back to Identity Cloud to continue the authentication journey.

Exceptions during local processing cause a redirect with an assertion JWT containing an `assertionError` claim. Exceptions that prevent the return of a valid assertion, such as an invalid incoming JWT or key error, cause an HTTP 500.

Usage

```
{
  "name": string,
  "type": "IdentityAssertionHandlerTechPreview",
  "config": {
    "identityAssertionPlugin": IdentityAssertionPluginTechPreview reference,
    "selfIdentifier": configuration expression<string>,
    "peerIdentifier": configuration expression<string>,
    "expire": configuration expression<duration>,
    "secretsProvider": Secrets Provider reference,
    "verificationSecretId": configuration expression<secret-id>,
    "decryptionSecretId": configuration expression<secret-id>,
    "skewAllowance": configuration expression<duration>,
    "signature": object
  }
}
```

`"identityAssertionPlugin"`: *configuration expression<string>, required*

An implementation of org.forgerock.openig.handler.assertion.IdentityAssertionPluginTechPreview.

An out-of-the box implementation is available in ScriptableIdentityAssertionPluginTechPreview.

`"selfIdentifier"`: *configuration expression<string>, required*

An identifier to validate that this IG instance is the right audience for the incoming JWT from Identity Cloud. The same identifier is used for the `iss` claim of the outgoing JWT sent to Identity Cloud.

Can't be null.

`"peerIdentifier"`: *configuration expression<string>, required*

An identifier to validate that the incoming JWT is from the expected peer. The same identifier is used for the `aud` claim in the outgoing JWT sent Identity Cloud.

Can't be null.

`"expire"`: *duration, optional*

The expiry time of the outgoing JWT sent to Identity Cloud.

Default: 30 seconds

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for cryptographic keys.

`"verificationSecretId"`: *configuration expression<secret-id>, required*

The secret ID for the secret to validate the signature of the incoming JWT. The secret ID must point to a CryptoKey.

`"decryptionSecretId"`: *configuration expression<secret-id>, optional*

The secret ID for the secret to decrypt the incoming JWT. The secret ID must point to a CryptoKey.

When this property isn't set, IG treats the incoming JWT as signed but not encrypted.

Default: Not set.

`"skewAllowance"`: *configuration expression<duration>, optional*

The duration to add to the validity period of a JWT to allow for clock skew between different servers.

A `skewAllowance` of 2 minutes affects the validity period as follows:

- A JWT with an `iat` of 12:00 is valid from 11:58 on the IG clock.
- A JWT with an `exp` 13:00 is expired after 13:02 on the IG clock.

Default: To support a zero-trust policy, the skew allowance is by default `zero`.

`"signature"`: *object, required*

A JWT signature to validate the authenticity of claims or data for the outgoing JWT.

```
{
  "signature": {
    "secretId": configuration expression<secret-id>,
    "algorithm": configuration expression<string>,
    "encryption": object
  }
}
```

`"secretId"`: *secret-id*, ***required***

The secret ID of the signing key. The secret ID must point to a CryptoKey.

`"algorithm"`: ***configuration expression***<*string*>, ***optional***

The signing algorithm.

Default: `RS256`

`"encryption"`: *object*, ***required***

Configuration to encrypt the JWT.

```
{
  "encryption": {
    "secretId": configuration expression<secret-id>,
    "algorithm": configuration expression<string>,
    "method": configuration expression<string>
  }
}
```

`"secretId"`: *secret-id*, ***required***

The secret ID of the encryption key. The secret ID must point to a CryptoKey.

`"algorithm"`: ***configuration expression***<*string*>, ***required***

The encryption algorithm. Use an algorithm from the List of JWS Algorithms.

`"method"`: ***configuration expression***<*string*>, ***required***

The encryption method. Use a method from the List of JWE Algorithms.

Example

The following example route is for a Identity Cloud authentication journey that uses a Gateway Communication node.

For information about the `identityAssertionPlugin` object, refer to the example in ScriptableIdentityAssertionPluginTechPreview.

```
{
  "type": "IdentityAssertionHandlerTechPreview",
  "config": {
    "identityAssertionPlugin": "BasicAuthScriptablePlugin",
    "selfIdentifier": "identity-gateway",
    "peerIdentifier": "gateway-communication-node",
    "secretsProvider": [
      "IG-Decrypt",
      "Node-Verify",
      "IG-Sign",
      "Node-Encrypt"
    ],
    "verificationSecretId": "id.key.for.verifying.incoming.jwt",
    "decryptionSecretId": "id.key.for.decrypting.incoming.jwt",
    "signature": {
      "secretId": "id.key.for.signing.assertion.jwt",
      "algorithm": "RS256",
      "encryption": {
        "secretId": "id.key.for.encrypting.assertion.jwt",
        "algorithm": "RSA-OAEP-256",
        "method": "A256GCM"
      }
    }
  }
}
```

More information

org.forgerock.openig.handler.assertion.IdentityAssertionPluginTechPreview.

JwkSetHandler

Expose cryptographic keys as a JWK set. Use this handler to reuse exposed keys for their assigned purpose in a downstream application.

Consider the following limitations:

- When the public key isn't available, the corresponding private key can't be exposed.

> **Caution**
>
> You are not recommended to expose private keys as a JWK.

- Keys in secure storage, such as a Hardware Security Module (HSM) or remote server, can't be exposed.

For a description of how secrets are managed, refer to About secrets.

For information about JWKs and JWK Sets, refer to JSON Web Key (JWK).

Usage

```
{
  "name": string,
  "type": "JwkSetHandler",
  "config": {
    "secretsProvider": SecretsProvider reference,
    "purposes": [ object, ... ],
    "exposePrivateSecrets": configuration expression<boolean>
  }
}
```

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider containing secrets to expose in the JwkSet.

`"purposes"`: *array of objects, required*

One or more purposes for the JwkSet key.

```
{
  "purposes": [
    {
      "secretId": configuration expression<secret-id>,
      "keyUsage": configuration expression<enumeration>
    },
    ...
  ]
}
```

`"secretId"`: *configuration expression<secret-id>, required*

The secret ID of the key to be exposed in the JwkSet.

This secret ID must point to a CryptoKey.

`"keyUsage"`: *configuration expression<enumeration>, required*

The allowed use of the key:

- `AGREE_KEY`: Export the private key used in the key agreement protocol, for example, Diffie-Hellman.
- `ENCRYPT`: Export the public key used to encrypt data.
- `DECRYPT`: Export the private key used to decrypt data.
- `SIGN`: Export the private key used to sign data.
- `VERIFY`: Export the public key used to verify signature data.
- `WRAP_KEY`: Export the public key used to encrypt (wrap) other keys.
- `UNWRAP_KEY`: Export the private key used to decrypt (unwrap) other keys.

`exposePrivateSecrets`: *configuration expression<boolean>, optional*

A flag indicating whether to publish private keys in a JWK set. As a security safeguard, this property is `false` by default to prevent the accidental exposure of private keys.

`true`: Publish both public and private keys in the JWK set `false`: Publish only public keys in the JWK set

Default: `false`

Examples

This example uses a JwkSetHandler to expose a signing key used by the JwtBuilderFilter:

1. Set an environment variable for the base64-encoded secret to sign the JWT:

```
$ export SIGNING_KEY_SECRET_ID='cGFzc3dvcmQ='
```

2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/jwksethandler.json
```

Windows

```
%appdata%\OpenIG\config\routes\jwksethandler.json
```

```
{
  "name": "jwksethandler",
  "condition": "${find(request.uri.path, '/jwksethandler')}",
  "heap": [
    {
      "name": "SecretKeyPropertyFormat-1",
      "type": "SecretKeyPropertyFormat",
      "config": {
        "format": "BASE64",
        "algorithm": "AES"
      }
    },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore",
      "config": {
        "mappings": [{
          "secretId": "signing.key.secret.id",
          "format": "SecretKeyPropertyFormat-1"
        }]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "JwtBuilderFilter-1",
          "type": "JwtBuilderFilter",
          "config": {
            "template": {
              "name": "${contexts.userProfile.commonName}",
              "email": "${contexts.userProfile.rawInfo.mail[0]}"
            },
            "secretsProvider": "SystemAndEnvSecretStore-1",
            "signature": {
              "secretId": "signing.key.secret.id",
              "algorithm": "HS256"
            }
          }
        }
      ],
      "handler": {
        "type": "JwkSetHandler",
        "config": {
          "secretsProvider": "SystemAndEnvSecretStore-1",
          "purposes": [{
            "secretId": "signing.key.secret.id",
            "keyUsage": "SIGN"
          }]
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/jwksethandler`.
- The JWT signing key is managed by the SysEnvStoreSecretStore in the heap, which refers to the SecretKeyPropertyFormat for the secret's format.
- The JwtBuilderFilter `signature` property refers to the JWT signing key in the SysEnvStoreSecretStore.
- The JwkSetHandler refers to the JWT signing key.

3. Go to http://ig.example.com:8080/jwksethandler.

   The signing key is displayed as an array, as follows:

```
{
  "keys": [
    {
      "k": "cGFzc3dvcmQ",
      "kid": "signing.key.secret.id",
      "kty": "oct",
      "use": "sig"
    }
  ]
}
```

   The JWK set secret is ULR base64-encoded. Although the secret is set with the value `cGFzc3dvcmQ=`, the value `cGFzc3dvcmQ` is exposed.

More information

org.forgerock.openig.handler.JwkSetHandler

ResourceHandler

Serves static content from a directory.

Usage

```
{
  "name": string,
  "type": "ResourceHandler",
  "config": {
    "directories": [ configuration expression<string>, ... ],
    "basePath": configuration expression<string>,
    "welcomePages": [ configuration expression<string>, ... ],
    "temporaryStorage": TemporaryStorage reference
  }
}
```

Properties

`"directories"`: *array of configuration expression<strings>, required*

A list of one or more directories in which to search for static content.

When multiple directories are specified in an array, the directories are searched in the listed order.

`"basePath"`: *_configuration expression<string>, required if the route is not `/`*

The base path of the incoming request for static content.

To specify no base path, leave this property out of the configuration, or specify it as `"basePath": ""` or `"basePath": "/"`.

Default: `""`.

`"welcomePages"`: *array of configuration expression<strings>, optional*

A set of static content to serve from one of the specified directories when no specific resource is requested.

When multiple sets of static content are specified in an array, the sets are searched for in the listed order. The first set that is found is used.

Default: Empty

`"temporaryStorage"`: *TemporaryStorage reference, optional*

A TemporaryStorage object for the static content.

Default: `TemporaryStorage` heap object

Example

The following example serves requests to `http://ig.example.com:8080` with the static file `index.html` from `/path/to/static/pages/`:

```
{
  "name": "StaticWebsite",
  "type": "ResourceHandler",
  "config": {
    "directories": ["/path/to/static/pages"],
    "welcomePages": ["index.html"]
  }
}
```

When the `basePath` is `/website`, the example serves requests to `http://ig.example.com:8080/website`:

```
{
  "name": "StaticWebsite",
  "type": "ResourceHandler",
  "config": {
    "directories": ["/path/to/static/pages"],
    "basePath": "/website",
    "welcomePages": ["index.html"]
  }
}
```

More information

org.forgerock.openig.handler.resources.ResourceHandler

org.forgerock.http.protocol.Entity

ReverseProxyHandler

Proxy requests to protected applications. When IG relays the request to the protected application, IG is acting as a client of the application. IG is *client-side*.

Consider the following comparison of the ClientHandler and ReverseProxyHandler:

	ClientHandler	ReverseProxyHandler
Use this handler to …	Send requests to third-party services accessed within a route. The service can be AM or an HTTP API. The service can be an HTTP endpoint, such as AM, IDM, Identity Cloud, or any custom HTTP API.	Send requests to the final service accessed by a route. The service can be the final downstream application.
If the service does not respond in time, this handler …	Propagates the error through the Promise flow. If the error is not handled within the route, for example, by a FailureHandler, the handler returns a `500 Internal Server Error` response.	Stops processing the request, and returns a `502 Bad Gateway` response.

When uploading or downloading large files, prevent timeout issues by increasing the value of `soTimeout`, and using a streaming mode, as follows:

Configure the `streamingEnabled` property of AdminHttpApplication.

Usage

```
{
  "name": string,
  "type": "ReverseProxyHandler",
  "config": {
    "vertx": object,
    "connections": configuration expression<number>,
    "waitQueueSize": configuration expression<number>,
    "soTimeout": configuration expression<duration>,
    "connectionTimeout": configuration expression<duration>,
    "protocolVersion": configuration expression<enumeration>,
    "http2PriorKnowledge": configuration expression<boolean>,
    "proxyOptions": ProxyOptions reference,
    "temporaryStorage": TemporaryStorage reference,
    "tls": ClientTlsOptions reference,
    "retries": object,
    "circuitBreaker": object,
    "websocket": object,
    "hostnameVerifier": configuration expression<enumeration>, //deprecated
    "proxy": Server reference, //deprecated
    "systemProxy": boolean //deprecated
  }
}
```

* Legacy; no longer supported

Properties

`"vertx"`: *object*, *optional*

Vert.x-specific configuration for the handler, where IG does not provide its own first-class configuration. Vert.x options are described in HttpClientOptions.

The `vertx` object is read as a map, and values are evaluated as configuration expressions.

For properties where IG provides its own first-class configuration, Vert.x configuration options are disallowed, and the IG configuration option takes precedence over Vert.x options configured in `vertx`. The following Vert.x configuration options are disallowed client-side:

- `alpnVersions`
- `connectTimeout`
- `enabledCipherSuites`
- `enabledSecureTransportProtocols`
- `http2ClearTextUpgrade`
- `idleTimeout`
- `idleTimeoutUnit`
- `keyCertOptions`

- `keyStoreOptions`
- `maxWaitQueueSize`
- `pemKeyCertOptions`
- `pemTrustOptions`
- `pfxKeyCertOptions`
- `pfxTrustOptions`
- `port`
- `protocolVersion`
- `proxyOptions`
- `ssl`
- `trustOptions`
- `trustStoreOptions`
- `useAlpn`
- `verifyHost`

The following example configures the Vert.x configuration when IG is acting client-side. When IG is acting server-side, configure the `connectors:vertx` property of `admin.json`:

```
{
  "vertx": {
    "maxWebSocketFrameSize": 128000,
    "maxWebSocketMessageSize": 256000,
    "compressionLevel": 4,
    "maxHeaderSize": 16384
  }
}
```

The following example configures HTTP/2 connections when IG is acting client-side. The configuration allows IG to make HTTP/2 requests with large headers. When IG is acting server-side, configure the `connectors:vertx` property of `admin.json`:

```
{
  "vertx": {
    "initialSettings": {
      "maxHeaderListSize": 16384
    }
  }
}
```

`"connections"`: *configuration expression<number>, optional*

The maximum number of concurrent HTTP connections in the client connection pool.

For information about the interaction between this property and `waitQueueSize`, see the description of `waitQueueSize`.

Default: 64

"waitQueueSize": configuration expression<number>, optional

The maximum number of outbound requests allowed to queue when no downstream connections are available. Outbound requests received when the queue is full are rejected.

Use this property to limit memory use when there is a backlog of outbound requests, for example, when the protected application or third-party service is slow.

Configure `waitQueueSize` as follows:

- Not set (default): The wait queue is calculated as the square of `connections`.
  - If `connections` is not configured, then its default of `64` is used, giving the `waitQueueSize` of `4096`.
  - If the square of `connections` exceeds the maximum integer value for the Java JVM, the maximum integer value for the Java JVM is used.
- `-1`: The wait queue is unlimited. Requests received when there are no available connections are queued without limit.
- `0`: There is no wait queue. Requests received when there are no available connections are rejected.
- A value that is less than the square of `connections`:

  When the configuration is loaded, the configured value is used. IG generates a warning that the `waitQueueSize` is too small for the `connections` size, and recommends a different value.
- A value where `waitQueueSize` plus `connections` exceeds the maximum integer value for the Java JVM:

  When the configuration is loaded, the `waitQueueSize` is reduced to the maximum integer value for the Java JVM minus the value of `connections`. IG generates a warning.

Consider the following example configuration of `connections` and `waitQueueSize`:

```
{
  "handler" : {
    "name" : "proxy-handler",
    "type" : "ReverseProxyHandler",
    "MyCapture" : "all",
    "config": {
      "soTimeout": "10 seconds",
      "connectionTimeout": "10 seconds",
      "connections": 64,
      "waitQueueSize": 100
    }
  },
  "baseURI" : "http://app.example.com:8080",
  "condition" : "${find(request.uri.path, '/')}"
}
```

IG can propagate the request to the sample application using 64 connections. When the connections are consumed, up to 100 subsequent requests are queued until a connection is freed. Effectively IG can accommodate 164 requests, although user concurrency delay means more may be handled. Requests received when the waitQueue is full are rejected.

Default: Not set

`"connectionTimeout"`: *configuration expression<duration>, optional*

Time to wait to establish a connection, expressed as a duration

Default: 10 seconds

`"protocolVersion"`: *configuration expression<enumeration>, optional*

The version of HTTP protocol to use when processing requests:

- `HTTP/2`:
  - For HTTP, process requests using HTTP/1.1.
  - For HTTPS, process requests using HTTP/2.
- `HTTP/1.1`:
  - For HTTP and HTTPS, process requests using HTTP/1.1.
- Not set:
  - For HTTP, process requests using HTTP/1.1.
  - For HTTPS with `alpn` enabled in ClientTlsOptions, process requests using HTTP/1.1, with an HTTP/2 upgrade request. If the targeted server can use HTTP/2, the client uses HTTP/2.

    For HTTPS with `alpn` disabled in ClientTlsOptions, process requests using HTTP/1.1, without an HTTP/2 upgrade request.

    Note that `alpn` is enabled by default in ClientTlsOptions.

Default: Not set

> **Note**
>
> In HTTP/1.1 request messages, a `Host` header is required to specify the host and port number of the requested resource. In HTTP/2 request messages, the `Host` header is not available.
> In scripts or custom extensions that use HTTP/2, use `UriRouterContext.originalUri.host` or `UriRouterContext.originalUri.port` in requests.

`"http2PriorKnowledge"`: *configuration expression<boolean>, optional*

A flag for whether the client should have prior knowledge that the server supports HTTP/2. This property is for cleartext (non-TLS requests) only, and is used only when `protocolVersion` is HTTP/2.

- `false`: The client checks whether the server supports HTTP/2 by sending an HTTP/1.1 request to upgrade the connection to HTTP/2:
  - If the server supports HTTP/2, the server upgrades the connection to HTTP/2, and subsequent requests are processed over HTTP/2.
  - If the server does not support HTTP/2, the connection is not upgraded, and subsequent requests are processed over HTTP/1.
- `true`: The client does not check that the server supports HTTP/2. The client sends HTTP/2 requests to the server, assuming that the server supports HTTP/2.

Default: `false`

`"proxyOptions"`: *ProxyOptions reference, optional*

A proxy server to which requests can be submitted. Use this property to relay requests to other parts of the network. For example, use it to submit requests from an internal network to the internet.

Provide the name of a ProxyOptions object defined in the heap or an inline configuration.

Default: A heap object named `ProxyOptions`.

`"soTimeout"`: *configuration expression<duration>, optional*

Socket timeout, after which stalled connections are destroyed, expressed as a duration.

> **Tip**
>
> If `SocketTimeoutException` errors occur in the logs when you try to upload or download large files, consider increasing `soTimeout`.

Default: 10 seconds

`"temporaryStorage"`: *TemporaryStorage reference, optional*

The TemporaryStorage object to buffer the request and response, when the `streamingEnabled` property of admin.json is `false`.

Default: A heap object named `TemporaryStorage`.

`tls`: *ClientTlsOptions reference, optional*

> **Important**
>
> Use of a TlsOptions reference is deprecated; use ClientTlsOptions instead. For more information, refer to the Deprecated section of the *Release Notes*.

Configure options for connections to TLS-protected endpoints, based on ClientTlsOptions. Define the object inline or in the heap.

Default: Connections to TLS-protected endpoints are not configured.

`"retries"`: *object*, *optional*

Enable and configure retry for requests.

During the execution of a request to a remote server, if a runtime exception occurs, or a condition is met, IG waits for a delay, and then schedules a new execution of the request. IG tries until the allowed number of retries is reached or the execution succeeds.

A warning-level entry is logged if all retry attempts fail; a debug-level entry is logged if a retry succeeds.

```
"retries": {
  "enabled": configuration expression<boolean>,
  "condition": runtime expression<boolean>,
  "executor": ScheduledExecutorService reference,
  "count": configuration expression<number>,
  "delay": configuration expression<duration>,
  }
}
```

`"enabled"`: *configuration expression*<*boolean*>, *optional*

Enable retries.

Default: `true`

`"condition"`: *runtine expression*<*boolean*>, *optional*

An inline IG expression to define a condition based on the response, such as an error code.

The condition is evaluated as follows:

- If `true`, IG retries the request until the value in `count` is reached.
- If `false`, IG retries the request only if a runtime exception occurs, until the value in `count` is reached.

Default: `${false}`

`"executor"`: *ScheduledExecutorService reference*, *optional*

The ScheduledExecutorService to use for scheduling delayed execution of the request.

Default: `ScheduledExecutorService`

See also ScheduledExecutorService.

`"count"`: *configuration expression*<*number*>, *optional*

The maximum number of retries to perform. After this threshold is passed and if the request is still not successful, then the ClientHandler propagates the failure.

Retries caused by any runtime exception or triggered condition are included in the count.

Default: `5`

`"delay"`: *configuration expression<duration>, optional*

The time to wait before retrying the request.

After a failure to send the request, if the number of retries is below the threshold, a new attempt is scheduled with the executor service after this delay.

Default: `10 seconds`

The following example configures a retry when a downstream component returns a `502 Bad Gateway` response code:

```
"retries": {
    "enabled": true,
    "condition": "${response.status.code == 502}"
}
```

The following example configures the handler to retry the request only once, after a 1-minute delay:

```
{
  "retries": {
    "count": 1,
    "delay": "1 minute"
  }
}
```

The following example configures the handler to retry the request at most 20 times, every second:

```
{
  "retries": {
    "count": 20,
    "delay": "1 second"
  }
}
```

The following example configures the handler to retry the request 5 times, every 10 seconds (default values), with a dedicated executor:

```
{
  "retries": {
    "executor": {
      "type": "ScheduledExecutorService",
      "config": {
        "corePoolSize": 20
      }
    }
  }
}
```

`"circuitBreaker"`: *object, optional*

Enable and configure a circuit breaker to trip when the number of failures exceeds a configured threshold. Calls to downstream services are stopped, and a runtime exception is returned. The circuit breaker is reset after the configured delay.

```
{
  "circuitBreaker": {
    "enabled": configuration expression<boolean>,
    "maxFailures":  configuration expression<integer>,
    "openDuration": configuration expression<duration>,
    "openHandler": Handler reference,
    "slidingCounter":  object,
    "executor":  ScheduledExecutorService reference
  }
}
```

`"enabled"`: *configuration expression<boolean>, optional*

A flag to enable the circuit breaker.

Default: `true`

`"maxFailures"`: *configuration expression<number>, required*

The maximum number of failed requests allowed in the window given by `size`, before the circuit breaker trips. The value must be greater than zero.

> **Important**
>
> When `retries` is set, the circuit breaker does not count retried requests as failures. Bear this in mind when you set `maxFailures`.
> In the following example, a request can fail and then be retried three times. If it fails the third retry, the request has failed four times, but the circuit breaker counts only one failure.
>
> ```
> {
>   "retries": {
>     "count": 3,
>     "delay": "1 second"
>   }
> }
> ```

`"openDuration"`: *configuration expression<duration>, required*

The duration for which the circuit stays open after the circuit breaker trips. The `executor` schedules the circuit to be closed after this duration.

`"openHandler"`: *Handler reference, optional*

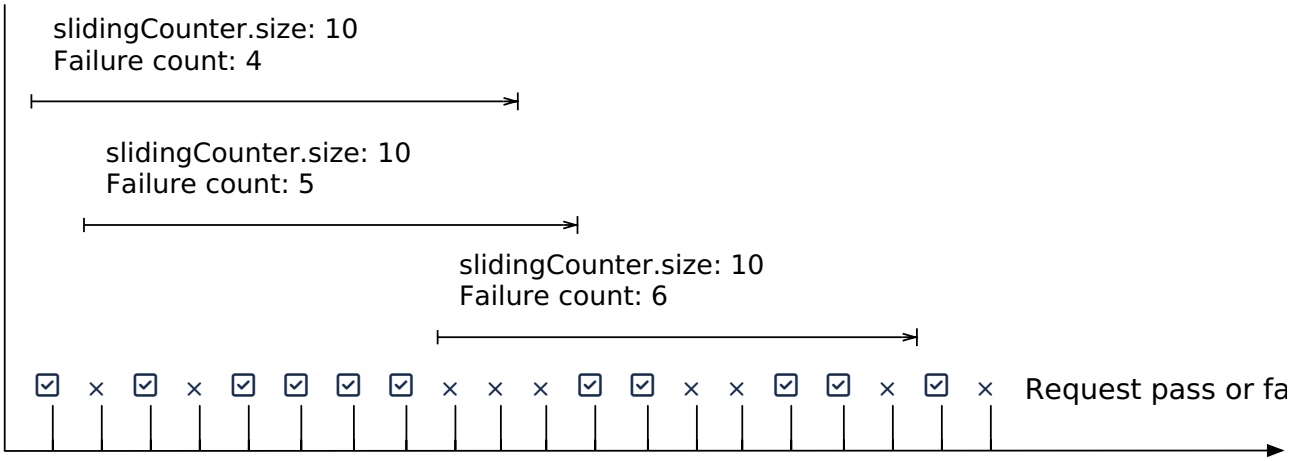
The Handler to call when the circuit is open.

Default: A handler that throws a RuntimeException with a "circuit-breaker open" message.

`"slidingCounter"`: *object*, *optional*

A sliding window error counter. The circuit breaker trips when the number of failed requests in the number of requests given by `size` reaches `maxFailures`.

The following image illustrates how the sliding window counts failed requests:




```
{
  "slidingCounter":  {
    "size": configuration expression<number>
  }
}
```

`"size"`: *configuration expression<number>*, *required*

The size of the sliding window in which to count errors.

The value of `size` must be greater than zero, and greater than the value of `maxFailures`, otherwise an exception is thrown.

`"executor"`: *ScheduledExecutorService reference*, *optional*

A ScheduledExecutorService to schedule closure of the circuit after the duration given by `openDuration`.

Default: The default ScheduledExecutorService in the heap

`"websocket"`: *object*, *optional*

Object to configure upgrade from HTTP or HTTPS protocol to WebSocket protocol.

Every key/value of the `websocket` object is evaluated as a configuration expression.

List the subprotocols that are proxied by IG in the `vertx` property of AdminHttpApplication (`admin.json`). For more information and an example of proxying WebSocket traffic, refer to WebSocket traffic

```
{
  "websocket": {
    "enabled": configuration expression<boolean>,
    "connectionTimeout": configuration expression<duration>,
    "soTimeout": configuration expression<duration>,
    "numberOfSelectors": configuration expression<number>,
    "tls": ClientTlsOptions reference,
    "proxyOptions": ProxyOptions reference,
    "vertx": object
  }
}
```

For more information, refer to The WebSocket Protocol.

`"enabled"`: *configuration expression<boolean>,optional*

Enable upgrade from HTTP protocol and HTTPS protocol to WebSocket protocol.

Default: `false`

`"connectionTimeout"`: *configuration expression<duration>, optional*

The maximum time allowed to establish a WebSocket connection.

Default: The value of handler's main `connectionTimeout`.

`"soTimeout"`: *configuration expression<duration>, optional*

The time after which stalled connections are destroyed.

> **Tip**
>
> If there can be long delays between messages, consider increasing this value. Alternatively, keep the connection active by using WebSocket ping messages in your application.

Default: The value of handler's main `soTimeout`.

`"numberOfSelectors"`: *configuration expression<number>, optional*

The maximum number of worker threads.

In deployments with a high number of simultaneous connections, consider increasing the value of this property.

Default: `2`

`"tls"`: *ClientTlsOptions reference, optional*

Configure options for connections to TLS-protected endpoints, based on a ClientTlsOptions configuration. Define a ClientTlsOptions object inline or in the heap.

Default: Use ClientTlsOptions defined for the handler

`"proxyOptions"`: *ProxyOptions reference, optional*

A proxy server to which requests can be submitted. Use this property to relay requests to other parts of the network. For example, use it to submit requests from an internal network to the internet.

Provide the name of a ProxyOptions object defined in the heap or an inline configuration.

Default: A heap object named `ProxyOptions`.

`"vertx"`: *object, optional*

Vert.x-specific configuration for the WebSocket connection, where IG does not provide its own first-class configuration. Vert.x options are described in HttpClientOptions.

For properties where IG provides its own first-class configuration, Vert.x configuration options are disallowed, and the IG configuration option takes precedence over Vert.x options configured in `vertx`. The following Vert.x configuration options are disallowed client-side:

- `alpnVersions`
- `connectTimeout`
- `enabledCipherSuites`
- `enabledSecureTransportProtocols`
- `http2ClearTextUpgrade`
- `idleTimeout`
- `idleTimeoutUnit`
- `keyCertOptions`
- `keyStoreOptions`
- `maxWaitQueueSize`
- `pemKeyCertOptions`
- `pemTrustOptions`
- `pfxKeyCertOptions`
- `pfxTrustOptions`
- `port`
- `protocolVersion`
- `proxyOptions`
- `ssl`
- `trustOptions`
- `trustStoreOptions`

- `useAlpn`
- `verifyHost`

The following example configures the Vert.x configuration when IG is acting client-side. When IG is acting server-side, configure the `connectors:vertx` property of `admin.json`:

```
{
  "vertx": {
    "maxWebSocketFrameSize": 128000,
    "maxWebSocketMessageSize": 256000,
    "compressionLevel": 4,
    "maxHeaderSize": 16384
  }
}
```

The following example configures HTTP/2 connections when IG is acting client-side. The configuration allows IG to make HTTP/2 requests with large headers. When IG is acting server-side, configure the `connectors:vertx` property of `admin.json`:

```
{
  "vertx": {
    "initialSettings": {
      "maxHeaderListSize": 16384
    }
  }
}
```

The following default `vertx` configuration provided by this handler overrides the Vert.x defaults:

- `tryUsePerFrameCompression` = `true`
- `tryUsePerMessageCompression` = `true`

"hostnameVerifier": configuration expression<enumeration>, optional

> **Important**
>
> This property is deprecated; use the ClientTlsOptions property `hostnameVerifier` instead.
> If a ReverseProxyHandler includes the deprecated `"hostnameVerifier": "ALLOW_ALL"` configuration, it takes precedence, and deprecation warnings are written to the logs.
> For more information, refer to the Deprecated section of the *Release Notes*.

The way to handle hostname verification for outgoing SSL connections. Use one of the following values:

- `ALLOW_ALL`: Allow a certificate issued by a trusted CA for any hostname or domain to be accepted for a connection to any domain.

  This setting allows a certificate issued for one company to be accepted as a valid certificate for another company. To prevent the compromise of TLS connections, use this setting in development mode only. In production, use `STRICT`.

- `STRICT`: Match the hostname either as the value of the the first CN, or any of the subject-alt names.

A wildcard can occur in the CN, and in any of the subject-alt names. Wildcards match one domain level, so `*.example.com` matches `www.example.com` but not `some.host.example.com`.

Default: `STRICT`

`"proxy"`: *Server reference, optional*

> **Important**
>
> This property is deprecated; use `proxyOptions` instead. For more information, refer to the Deprecated section of the *Release Notes*.

A proxy server to which requests can be submitted. Use this property to relay requests to other parts of the network. For example, use it to submit requests from an internal network to the internet.

If both `proxy` and `systemProxy` are defined, `proxy` takes precedence.

```
"proxy" : {
  "uri": configuration expression<uri string>,
  "username": configuration expression<string>,
  "passwordSecretId": configuration expression<secret-id>,
  "secretsProvider": SecretsProvider reference
}
```

`"uri"`: *configuration expression<uri string>, required*

URI of a server to use as a proxy for outgoing requests.

The result of the expression must be a string that represents a valid URI, but is not a real java.net.URI object.

username: configuration expression<string>, required if the proxy requires authentication

Username to access the proxy server.

`"passwordSecretId"`: *configuration expression<secret-id>, required if the proxy requires authentication*

The secret ID of the password to access the proxy server.

This secret ID must point to a GenericSecret.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for the proxy's password.

`"systemProxy"`: *boolean, optional*

> **Important**
>
> This property is deprecated; use `proxyOptions` instead. For more information, refer to the Deprecated section of the *Release Notes*.

Submit outgoing requests to a system-defined proxy, set by the following system properties or their HTTPS equivalents:

- `http.proxyHost`, the host name of the proxy server.
- `http.proxyPort`, the port number of the proxy server. The default is `80`.
- `http.nonProxyHosts`, a list of hosts that should be reached directly, bypassing the proxy.

This property can't be used with a proxy that requires a username and password. Use the property `proxy` instead.

If both `proxy` and `systemProxy` are defined, `proxy` takes precedence.

For more information, refer to Java Networking and Proxies.

Default: False.

`"keyManager"`: *Key manager reference(s), optional*

> **Important**
>
> This property is deprecated; use the `tls` property instead to configure ClientTlsOptions. For more information, refer to the Deprecated section of the *Release Notes*.

The key manager(s) that handle(s) this client's keys and certificates.

The value of this field can be a single reference, or an array of references.

Provide either the name(s) of key manager object(s) defined in the heap, or specify the configuration object(s) inline.

You can specify either a single key manager, as in `"keyManager": "MyKeyManager"`, or an array of key managers, as in `"keyManager": [ "FirstKeyManager", "SecondKeyManager" ]`.

If you do not configure a key manager, then the client cannot present a certificate, and so cannot play the client role in mutual authentication.

`"sslCipherSuites"`: *array of strings, optional*

> **Important**
>
> This property is deprecated; use the `tls` property instead to configure ClientTlsOptions. For more information, refer to the Deprecated section of the *Release Notes*.

Array of cipher suite names, used to restrict the cipher suites allowed when negotiating transport layer security for an HTTPS connection.

For information about the available cipher suite names, refer to the documentation for the Java virtual machine (JVM) where you run IG. For Oracle Java, refer to the list of JSSE Cipher Suite Names.

Default: Allow any cipher suite supported by the JVM.

`"sslContextAlgorithm"`: *string, optional*

> **Important**
>
> This property is deprecated; use the `tls` property instead to configure ClientTlsOptions. For more information, refer to the Deprecated section of the *Release Notes*.

Default: TLS

`"sslEnabledProtocols"`: *array of strings, optional*

> **Important**
>
> This property is deprecated; use the `tls` property instead to configure ClientTlsOptions. For more information, refer to the Deprecated section of the *Release Notes*.

Array of protocol names, used to restrict the protocols allowed when negotiating transport layer security for an HTTPS connection.

Default: Allow any protocol supported by the JVM.

`"trustManager"`: *Trust manager reference(s), optional*

> **Important**
>
> This property is deprecated; use the `tls` property instead to configure ClientTlsOptions. For more information, refer to the Deprecated section of the *Release Notes*.

The trust managers that handle(s) peers' public key certificates.

The value of this field can be a single reference, or an array of references.

Provide either the name(s) of TrustManager object(s) defined in the heap, or specify the configuration object(s) inline.

You can specify either a single trust manager, as in `"trustManager": "MyTrustManager"`, or an array of trust managers, as in `"trustManager": [ "FirstTrustManager", "SecondTrustManager" ]`.

If you do not configure a trust manager, then the client uses only the default Java truststore. The default Java truststore depends on the Java environment. For example, `$JAVA_HOME/lib/security/cacerts`.

More information

org.forgerock.openig.handler.ReverseProxyHandlerHeaplet

Route

Routes are JSON-encoded configuration files that you add to IG to manage requests. You can add routes in the following ways:

- Manually into the filesystem.
- Through Common REST commands. For information, refer to Routes and Common REST.
- Through Studio. For information, refer to the Studio guide.

Routes handle requests and their context, and then hand off any request they accept to a Handler.

When a route has a condition, it handles only requests that meet the condition. When a route has no condition, it handles any request.

Routes inherit settings from their parent configuration. This means that you can configure global objects in the `config.json` heap, for example, and then reference the objects by name in any other IG configuration.

Usage

```
{
  "handler": Handler reference,
  "heap": [ object, ... ],
  "condition": runtime expression<boolean>,
  "name": string,
  "session": AsyncSessionManager reference,
  "auditService": AuditService reference,
  "globalDecorators": map,
  "decorator name": Decorator object
}
```

(*)Deprecated

Properties

"handler": *Handler reference, required*

The Handler to which IG dispaches requests.

Provide the name of a Handler object defined in the heap or an inline Handler configuration object.

"heap": *array of objects, optional*

Heap object configuration for objects local to this route.

Objects referenced but not defined here are inherited from the parent.

You can omit an empty array. If you only have one object in the heap, you can inline it as the handler value.

See also Heap objects.

"condition": *runtime expression<boolean>, optional*

A condition based on the request, context, or IG runtime environment, such as system properties or environment variables.

- `true`: The request is dispatched to the route.
- `false`: The condition for the next route in the configuration is evaluated.
- No condition: the request is dispatched unconditionally to the route.

> **Tip**
>
> For debugging, log the routes for which IG evaluates a condition, and the route that matches a condition. Add the following line to a custom `$HOME/.openig/config/logback.xml`, and restart IG:
>
> ```
> <logger name="org.forgerock.openig.handler.router.RouterHandler" level="trace" />
> ```
>
> For information, refer to Manage logs.

An external request can never match a condition that uses the reserved administrative route. Therefore, routes that use these conditions are effectively ignored. For example, if `/openig` is the administrative route, a route with the following condition is ignored: `${find(request.uri.path, '^/openig/my/path')}`.

Default: `${true}`

For example conditions and requests that match them, refer to Example conditions and requests.

`"name"`: *string*, *optional*

Route name.

The Router uses the `name` property to order the routes in the configuration. If the route does not have a `name` property, the Router uses the route ID.

The route ID is managed as follows:

- When you add a route manually to the routes folder, the route ID is the value of the `_id` field. If there is no `_id` field, the route ID is the filename of the added route.
- When you add a route through the Common REST endpoint, the route ID is the value of the mandatory `_id` field.
- When you add a route through Studio, you can edit the default route ID.

  CAUTION: The filename of a route cannot be `default.json`. The route `name` property or route ID cannot be `default`.

Default: route ID

`"session"`: *AsyncSessionManager reference.* *reference*, *optional*

Stateless session implementation for this route. Define an AuthenticatedEncryptedJwtSessionManager object inline or in the heap.

When a request enters the route, IG builds a new session object for the route. The session content is available to the route's downstream handlers and filters. Session content available in the ascending configuration (a parent route or `config.json`) is not available in the new session.

When the response exits the route, the session content is serialized as a secure JWT that is encrypted and signed, and the resulting JWT string is placed in a cookie. Session information set inside the route is no longer available. The `session` references the previous session object.

Default: Do not change the session storage implementation.

For more information, refer to AsyncSessionManager, and Sessions.

`"auditService"`: *AuditService* *reference*, *optional*

An audit service for the route. Provide either the name of an AuditService object defined in the heap or an inline AuditService configuration object.

Default: No auditing of a configuration. The NoOpAuditService provides an empty audit service to the top-level heap and its child routes.

`"globalDecorators"`: *map, optional*

A map of one or more data pairs with the format `Map<String, JsonValue>`, where:

- The key is a decorator name
- The value is a decorator configuration, passed *as is* to the decorator

The following format is required:

```
{
  "globalDecorators": {
    "decorator name": "decoration configuration",
    ...
  }
}
```

All compatible objects in a route are decorated with the mapped decorator value. For more information, refer to Decorators.

In the following example, the property decorates all compatible objects in the route with a capture and timer decorator:

```
"globalDecorators": {
  "capture": "all",
  "timer": true
}
```

Default: Empty

`"decorator name"`: *Decorator object, optional*

Decorate the main handler of this route with a decorator referred to by the decorator name, and provide the configuration as described in Decorators.

Default: No decoration.

Route metrics at the Prometheus Scrape Endpoint

Route metrics at the Prometheus Scrape Endpoint have the following labels:

- `name`: Route name, for example, `My Route`.

  If the router was declared with a default handler, then its metrics are published through the route named `default`.

- `route`: Route identifier, for example, `my-route`.
- `router`: Fully qualified name of the router, for example, `gateway.main-router`.

The following table summarizes the recorded metrics:

Name[1]	Monitoring type	Description
`ig_route_request_active`	Gauge	Number of requests being processed.
`ig_route_request_total`	Counter	Number of requests processed by the router or route since it was deployed.
`ig_route_response_error_total`	Counter	Number of responses that threw an exception.
`ig_route_response_null_total`	Counter	Number of responses that were not handled by IG.
`ig_route_response_status_total`	Counter	Number of responses by HTTP status code family. The `family` label depends on the HTTP status code: • Informational (1xx) • Successful (2xx) • Redirection (3xx) • Client_error (4xx) • Server_error (5xx) • Unknown (status code >= 600)
`ig_route_response_time`	Summary	A summary of response time observations.

[1]Metric names are deprecated and expected to be replaced with names ending in _total. The information provided by the metric isn't deprecated. Other Prometheus metrics aren't affected.

Learn more in Prometheus Scrape Endpoint.

Route metrics at the Common REST Monitoring Endpoint (deprecated)

Route metrics at the Common REST Monitoring Endpoint are published with an `_id` in the following pattern:

- `heap.router-name.route.route-name.metric`

The following table summarizes the recorded metrics:

Name	Monitoring type	Description
`request`	Counter	Number of requests processed by the router or route since it was deployed.
`request.active`	Gauge	Number of requests being processed by the router or route at this moment.

Name	Monitoring type	Description
`response.error`	Counter	Number of responses that threw an exception.
`response.null`	Counter	Number of responses that were not handled by IG.
`response.status.client_error`	Counter	Number of responses with an HTTP status code `400`-`499`, indicating client error.
`response.status.informational`	Counter	Number of responses with an HTTP status code `100`-`199`, indicating that they are provisional responses.
`response.status.redirection`	Counter	Number of responses with an HTTP status code `300`-`399`, indicating a redirect.
`response.status.server_error`	Counter	Number of responses with an HTTP status code `500`-`599`, indicating server error.
`response.status.successful`	Counter	Number of responses with an HTTP status code `200`-`299`, indicating success.
`response.status.unknown`	Counter	Number of responses with an HTTP status code `600`-`699`, indicating that a request failed and was not executed.
`response.time`	Timer	Time-series summary statistics.

Learn more in Common REST Monitoring Endpoint.

Example conditions and requests

Condition	Requests that meet the condition
`"${true}"`	All requests, because this expression always evaluates to `true`.
`"${find(request.uri.path, '^/login')}"`	`http://app.example.com/login`, ...

Condition	Requests that meet the condition
`"${request.uri.host == 'api.example.com'}"`	`http://api.example.com/`, `https://api.example.com/home`, `http://api.example.com:8080/home`, ...
`"${find(contexts.client.remoteAddress, '127.0.0.1')}"`	`http://localhost:8080/keygen`, `http://127.0.0.1:8080/keygen`, ... Where `/keygen` is not mandatory and could be anything else.
`"${find(request.uri.query, 'demo=simple')}"`	`http://ig.example.com:8080/login?demo=simple`, ... For information about URI query, refer to `query` in URI.
`"${request.uri.scheme == 'http'}"`	`http://ig.example.com:8080`, ...
`"${find(request.uri.path, '^/dispatch') or find(request.uri.path, '^/mylogin')}"`	`http://ig.example.com:8080/dispatch`, `http://ig.example.com:8080/mylogin`, ...
`"${request.uri.host == 'sp1.example.com' and not find(request.uri.path, '^/saml')}"`	`http://sp1.example.com:8080/`, `http://sp1.example.com/mypath`, ... Not `http://sp1.example.com:8080/saml`, `http://sp1.example.com/saml`, ...
`"condition": "${find (request.uri.path, '&{uriPath}')}"`	`http://ig.example.com:8080/hello`, when the following property is configured: `{ "properties": { "uriPath": "hello" } }` For information about including properties in the configuration, refer to Route properties.
`"condition": "${request.headers['X-Forwarded-Host'][0] == 'service.example.com'}"`	Requests with the header `X-Forwarded-Host`, whose first value is `service.example.com`.

Condition	Requests that meet the condition
`"condition": "#{find(request.uri.path, '^/openam/oauth2/access_token') && request.entity.form['client_id'][0] == 'client-service'}"`	Requests where an OAuth 2.0 client named `client-service` issues the original access token request.
`"condition": "#{find(request.uri.path, '^/openam/oauth2/access_token') && request.entity.form['grant_type'][0] == 'client_credentials'}"`	Requests using the client credentials grant-type.

Router

A Handler that performs the following tasks:

- Defines the routes directory and loads routes into the configuration.
- Depending on the scanning interval, periodically scans the routes directory and updates the IG configuration when routes are added, removed, or changed. The router updates the IG configuration without needing to restart IG or access the route.
- Manages an internal list of routes, where routes are ordered lexicographically by route name. If a route is not named, then the route ID is used instead. For more information, refer to Route.
- Routes requests to the first route in the internal list of routes, whose condition is satisfied.

  Because the list of routes is ordered lexicographically by route name, name your routes with this in mind:

  - If a request satisfies the condition of more than one route, it is routed to the first route in the list whose condition is met.
  - Even if the request matches a later route in the list, it might never reach that route.

  If a request does not satisfy the condition of any route, it is routed to the default handler if one is configured.

The router does not have to know about specific routes in advance - you can configure the router first and then add routes while IG is running.

> **Important**
>
> Studio deploys and undeploys routes through a main router named `_router`, which is the name of the main router in the default configuration. If you use a custom `config.json`, make sure it contains a main router named `_router`.

Usage

```
{
    "name": string,
    "type": "Router",
    "config": {
        "defaultHandler": Handler reference,
        "directory": configuration expression<string>,
        "scanInterval": configuration expression<duration>
    }
}
```

An alternative value for type is RouterHandler.

Properties

`"defaultHandler"`: *Handler reference, optional*

Handler to use when a request does not satisfy the condition of any route.

Provide either the name of a handler object defined in the heap or an inline handler configuration object.

Default: If no default route is set either here or in the route configurations, IG aborts the request with an internal error.

See also Handlers.

`"directory"`: *configuration expression<string>, optional*

Directory from which to load route configuration files.

Default: The default directory for route configuration files, at `$HOME/.openig` (on Windows, `%appdata%\OpenIG`).

With the following example, route configuration files are loaded from `/path/to/safe/routes` instead of from the default directory:

```
{
  "type": "Router",
  "config": {
    "directory": "/path/to/safe/routes"
  }
}
```

> **Important**
>
> If you define multiple routers, configure `directory` so that the routers load route configuration files from different directories.

An infinite route-loading sequence is triggered when a router starts a route that, directly or indirectly, starts another router, which then loads route configuration files from the same directory.

See also Expressions.

`"scanInterval"`: *configuration expression<duration>, optional*

Time interval at which IG scans the specified directory for changes to routes. When a route is added, removed, or changed, the router updates the IG configuration without needing to restart IG or access the route.

When an integer is used for the `scanInterval`, the time unit is seconds.

To load routes at startup only, and prevent changes to the configuration if the routes are changed, set the scan interval to `disabled`.

Default: 10 seconds

Router metrics at the Prometheus Scrape Endpoint

Router metrics at the Prometheus Scrape Endpoint have the following labels:

- `fully_qualified_name`: Fully qualified name of the router, for example, `gateway.main-router`.
- `heap`: Name of the heap in which this router is declared, for example, `gateway`.
- `name`: Simple name declared in router configuration, for example, `main-router`.

The following table summarizes the recorded metrics:

Name	Monitoring type	Description
`ig_router_deployed_routes`	`Gauge`	Number of routes deployed in the configuration.

For more information about the the Prometheus Scrape Endpoint, refer to Prometheus Scrape Endpoint.

Router metrics at the Common REST Monitoring Endpoint (deprecated)

Router metrics at the Common REST Monitoring Endpoint are JSON objects, with the following form:

- `[heap name].[router name].deployed-routes`

The following table summarizes the recorded metrics:

Name	Monitoring type	Description
`deployed-routes`	`Gauge`	Number of routes deployed in the configuration.

For more information about the the Common REST Monitoring Endpoint, refer to Common REST Monitoring Endpoint.

Example

The following `config.json` file configures a Router:

```
{
  "handler": {
    "type": "Router",
    "name": "_router",
    "baseURI": "http://app.example.com:8081",
    "capture": "all"
  },
  "heap": [
    {
      "name": "JwtSession",
      "type": "JwtSession"
    },
    {
      "name": "capture",
      "type": "CaptureDecorator",
      "config": {
        "captureEntity": true,
        "_captureContext": true
      }
    }
  ]
}
```

All requests pass with the default settings to the Router. The Router scans `$HOME/.openig/config/routes` at startup, and rescans the directory every 10 seconds. If routes have been added, deleted, or changed, the router applies the changes.

The main router and any subrouters build the monitoring endpoints. For information about monitoring endpoints, refer to Monitoring endpoints.

More information

org.forgerock.openig.handler.router.RouterHandler

SamlFederationHandler (deprecated)

> **Important**
>
> This handler is deprecated; use the SamlFederationFilter instead.

A handler to play the role of SAML 2.0 Service Provider (SP).

Consider the following requirements for SamlFederationHandler:

- This handler does not support filtering; do not use it as the handler for a chain, which can include filters.
- Do not use this handler when its use depends on something in the response. The response can be handled independently of IG, and can be `null` when control returns to IG. For example, do not use this handler in a `SequenceHandler` where the `postcondition` depends on the response.
- Requests to the SamlFederationHandler must not be rebased, because the request URI must match the endpoint in the SAML metadata.

SAML in deployments with multiple instances of IG

When IG acts as a SAML service provider, session information is stored in the fedlet not the session cookie. In deployments with multiple instances of IG as a SAML service provider, it is necessary to set up sticky sessions so that requests always hit the instance where the SAML interaction was started.

For information, refer to Session state considerations in AM's *SAML v2.0 guide*.

Usage

```
{
    "name": string,
    "type": "SamlFederationHandler",
    "config": {
        "assertionMapping": map or configuration expression<map>,
        "redirectURI": configuration expression<url>,
        "secretsProvider": SecretsProvider reference,
        "assertionConsumerEndpoint": configuration expression<url>,
        "authnContext": configuration expression<string>,
        "authnContextDelimiter": configuration expression<string>,
        "logoutURI": configuration expression<url>,
        "sessionIndexMapping": configuration expression<string>,
        "singleLogoutEndpoint": configuration expression<url>,
        "singleLogoutEndpointSoap": configuration expression<url>,
        "SPinitiatedSLOEndpoint": configuration expression<url>,
        "SPinitiatedSSOEndpoint": configuration expression<url>,
        "useOriginalUri": configuration expression<boolean>,
        "subjectMapping": configuration expression<string>
    }
}
```

Properties

`"assertionMapping"`: *map or configuration expression<map>, required*

A map with the format `Map<String, String>`, where:

- Key: Session name, `localName`
- Value: SAML assertion name, `incomingName`, or a configuration expression that evaluates to the name

The following formats are allowed:

```
{
  "assertionMapping": {
    "string": "configuration expression<string>",
    ...
  }
}
```

```
{
  "assertionMapping": "configuration expression<map>"
}
```

In the following example, the session names `username` and `password` are mapped to SAML assertion names `mail` and `mailPassword` :

```
{
  "assertionMapping": {
    "username": "mail",
    "password": "mailPassword"
  }
}
```

If an incoming SAML assertion contains the following statement:

```
mail = demo@example.com
mailPassword = demopassword
```

Then the following values are set in the session:

```
username[0] = demo@example.com
password[0] = demopassword
```

For this to work, edit the `<Attribute name="attributeMap">` element in the SP extended metadata file, `$HOME/.openig/SAML/sp-extended.xml`, so that it matches the assertion mapping configured in the SAML 2.0 Identity Provider (IDP) metadata.

Because the dot character (`.`) serves as a query separator in expressions, do not use dot characters in the localName.

To prevent different handlers from overwriting each others' data, use unique localName settings when protecting multiple service providers.

`"redirectURI"`: *configuration expression<url>, required*

The page that the filter used to HTTP POST a login form recognizes as the login page for the protected application.

This is how IG and the Federation component work together to provide SSO. When IG detects the login page of the protected application, it redirects to the Federation component. Once the Federation handler validates the SAML exchanges with the IDP, and sets the required session attributes, it redirects back to the login page of the protected application. This allows the filter used to HTTP POST a login form to finish the job by creating a login form to post to the application based on the credentials retrieved from the session attributes.

`"secretsProvider"`: *SecretsProvider reference, optional*

The SecretsProvider to query for keys when AM provides signed or encrypted SAML assertions.

+ When this property isn't set, the keys are provided by direct keystore look-ups based on entries in the SP extended metadata file, `sp-extended.xml`.

+ Default: Empty.

`"assertionConsumerEndpoint"`: *configuration expression<string>, optional*

Default: `fedletapplication` (same as the Fedlet)

If you modify this attribute, change the metadata to match.

`"authnContext"`: *configuration expression<string>, optional*

Name of the session field to hold the value of the authentication context. Because the dot character (`.`) serves as a query separator in expressions, do not use dot characters in the field name.

Use this setting when protecting multiple service providers, as the different configurations must not map their data into the same fields of `session`. Otherwise different handlers can overwrite each others' data.

As an example, if you set `"authnContext": "myAuthnContext"`, then IG sets `session.myAuthnContext` to the authentication context specified in the assertion. When the authentication context is password over protected transport, then this results in the session containing `"myAuthnContext": "urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport"`.

Default: map to `session.authnContext`

`"authnContextDelimiter"`: *configuration expression<string>, optional*

The authentication context delimiter used when there are multiple authentication contexts in the assertion.

Default: `|`

`"logoutURI"`: *configuration expression<string>, optional*

Set this to the URI to visit after the user is logged out of the protected application.

You only need to set this if the application uses the single logout feature of the Identity Provider.

`"sessionIndexMapping"`: *configuration expression<string>, optional*

Name of the session field to hold the value of the session index. Because the dot character (`.`) serves as a query separator in expressions, do not use dot characters in the field name.

Use this setting when protecting multiple service providers, as the different configurations must not map their data into the same fields of `session`. Otherwise different handlers can overwrite each others' data.

As an example, if you set `"sessionIndexMapping": "mySessionIndex"`, then IG sets `session.mySessionIndex` to the session index specified in the assertion. This results in the session containing something like `"mySessionIndex": "s24ccbbffe2bfd761c32d42e1b7a9f60ea618f9801"`.

Default: map to `session.sessionIndex`

`"singleLogoutEndpoint"`: *configuration expression<string>, optional*

Default: `fedletSLORedirect` (same as the Fedlet)

If you modify this attribute, change the metadata to match.

`"singleLogoutEndpointSoap"`: *configuration expression<string>, optional*

Default: `fedletSloSoap` (same as the Fedlet)

If you modify this attribute, change the metadata to match.

`"SPinitiatedSLOEndpoint"`: *configuration expression<string>, optional*

Default: `SPInitiatedSLO`

If you modify this attribute, change the metadata to match.

`"SPinitiatedSSOEndpoint"`: *configuration expression<string>, optional*

Default: `SPInitiatedSSO`

If you modify this attribute, change the metadata to match.

`"useOriginalUri"`: *configuration expression<boolean>, optional*

When `true`, use the original URI instead of a rebased URI when validating RelayState and Assertion Consumer Location URLs. Use this property if a `baseUri` decorator is used in the route or in `config.json`.

Default: `false`

`"subjectMapping"`: *configuration expression<string>, optional*

Name of the session field to hold the value of the subject name. Because the dot character (`.`) serves as a query separator in expressions, do not use dot characters in the field name.

Use this setting when protecting multiple service providers, as the different configurations must not map their data into the same fields of `session`. Otherwise different handlers can overwrite each others' data.

As an example, if you set `"subjectMapping": "mySubjectName"`, then IG sets `session.mySubjectName` to the subject name specified in the assertion. If the subject name is an opaque identifier, then this results in the session containing something like `"mySubjectName": "vtOk+APj1s9Rr4yCka6V9pGUuzuL"`.

Default: map to `session.subjectName`

Example

For an example of how to set up IG as a SAML service provider, refer to SAML.

In the following example, IG receives a SAML 2.0 assertion from the IDP, and then logs the user in to the protected application using the username and password from the assertion:

```
{
    "name": "SamlFederationHandler",
    "type": "SamlFederationHandler",
    "config": {
        "assertionMapping": {
            "username": "mail",
            "password": "mailPassword"
        },
        "redirectURI": "/login",
        "logoutURI": "/logout"
    }
}
```

ScriptableHandler

Creates a response to a request by executing a script.

Scripts must return either a Promise<Response, NeverThrowsException> or a Response.

This section describes the usage of ScriptableHandler. For information about script properties, available global objects, and automatically imported classes, see Scripts.

Usage

```
{
    "name": string,
    "type": "ScriptableHandler",
    "config": {
        "type": configuration expression<string>,
        "file": configuration expression<string>, // Use either "file"
        "source": [ string, ... ],                // or "source", but not both
        "args": map,
        "clientHandler": Handler reference
    }
}
```

Properties

For information about properties for ScriptableHandler, refer to Scripts.

More information

org.forgerock.openig.handler.ScriptableHandler

SequenceHandler

Processes a request through a sequence of handlers and post conditions, as follows:

- A request is treated by `handler1`, and then `postcondition1` is evaluated.
- If `postcondition1` is true, the request is then treated by `handler2`, and so on.

```
{
  "handler": handler1,
  "postcondition": expression1
},
{
  "handler": handler2,
  "postcondition": expression2
},
...
```

Use this handler for multi-request processing, such as retrieving a form, extracting form content (for example, a nonce), and then submitting it in a subsequent request.

Usage

```
{
    "name": string,
    "type": "SequenceHandler",
    "config": {
        "bindings": [
            {
                "handler": Handler reference,
                "postcondition": runtime expression<boolean>
            }
        ]
    }
}
```

Properties

`"bindings"`: *array of objects, required*

A list of handler and postcondition bindings.

`"handler"`: *Handler reference, required*

The handler to dispatch the request to when it is the first handler in the bindings, or for subsequent handlers when their previous `postcondition` yields `true`.

Provide the name of a Handler object defined in the heap or an inline Handler configuration object.

`"postcondition"`: *runtime expression<boolean>, optional*

A flag to indicate that a post condition is met:

- `true`: The request is dispatched to the next handler in `bindings`.
- `false`: The sequence stops.

Postconditions are defined using IG expressions, as described in Expressions.

Default: `${true}`

More information

org.forgerock.openig.handler.SequenceHandler

StaticResponseHandler

Creates a response to a request statically, or based on something in the context.

Usage

```
{
  "name": string,
  "type": "StaticResponseHandler",
  "config": {
    "status": configuration expression<number>,
    "reason": configuration expression<string>,
    "headers": {
       configuration expression<string>: [ runtime expression<string>, ... ], ...
    },
    "trailers": {
       configuration expression<string>: [ runtime expression<string>, ... ], ...
    },
    "entity": runtime expression<string> or [ runtime expression<string>, ... ]
  }
}
```

Properties

`"status"`: *Status object*

The response status. For more information, refer to Status.

`"reason"`: *configuration expression<string>, optional*

Used only for custom HTTP status codes. For more information, refer to Response Status Codes and Status Code Registry.

`"headers"`: *map, optional*

One or more headers to set for a response, with the format `name: [ value, … ]`, where:

- *name* is a configuration expression<string> for a header name. If multiple expressions resolve to the same final string, *name* has multiple values.
- *value* one or more a runtime expression<strings> for header values.

When the property `entity` is used, set a `Content-Type` header with the correct content type value. The following example sets the content type of a message entity in the response:

```
"headers": {
  "Content-Type": [ "text/html; charset=UTF-8" ]
}
```

The following example is used in `federate-handler.json` to redirect the original URI from the request:

```
"headers": {
  "Location": [
     "http://sp.example.com:8080/saml/SPInitiatedSSO"
  ]
}
```

Default: Empty

`"trailers"`: *map, optional*

One or more trailers to set for a response, with the format `name: [ value, … ]`, where:

- *name* is a configuration expression<string> for a trailer name. If multiple expressions resolve to the same string, *name* has multiple values.

  The following trailer names are not allowed:

  - Message framing headers (for example, `Transfer-Encoding` and `Content-Length`)
  - Routing headers (for example, `Host`)
  - Request modifiers (for example, controls and conditionals such as `Cache-Control`, `Max-Forwards`, and `TE`)
  - Authentication headers (for example, `Authorization` and `Set-Cookie`)
  - `Content-Encoding`
  - `Content-Type`
  - `Content-Range`
  - `Trailer`

- *value* is one or more runtime expression<strings> for trailer values.

Default: Empty

`"entity"`: *runtime expression<string> or array of runtime expression<string>, optional*

The message entity body to include in a response.

If a `Content-Type` header is present, the entity must conform to the header and set the content length header automatically.

Methods are provided for accessing the entity as byte, string, or JSON content. For information, refer to Entity.

> **Important**
>
> Attackers during reconnaissance can use response messages to identify information about a deployment. For security, limit the amount of information in messages, and avoid using words that help identify IG.

Default: Empty

Example

```
{
  "name": "ErrorHandler",
  "type":"StaticResponseHandler",
  "config": {
    "status": 500,
    "headers": {
      "Content-Type": [ "text/html; charset=UTF-8" ]
    },
    "entity": "<html><h2>Epic #FAIL</h2></html>"
  }
}
```

```
{
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "content-type": [ "text/html" ]
      },
      "entity": [
        "<html>",
        "  <body>",
        "    <h1>Request Details</h1>",
        "    <p>The path was: ${request.uri.path}<p>",
        "    <p>The query params were: ${toString(request.queryParams)}</p>",
        "    <p>The headers were: ${toString(request.headers.entrySet())}<p>",
        "  </body>",
        "</html>"
      ]
    }
  }
}
```

More information

org.forgerock.openig.handler.StaticResponseHandler

Filters

Filter objects intercept requests and responses during processing, and change them as follows:

- Leave the request, response, and contexts unchanged. For example, the filter can simply can log the context as it passes through the filter.
- In the request flow, change any aspect of the request (such as the URL, headers, or entity), or replace the request with a new Request object.
- In the response flow, change any aspect of the response (such as the status, headers, or entity), or return a new Response instance

AllowOnlyFilter

Authorizes a request to continue processing if it satisfies at least one of the configured rules. Otherwise, passes the request to the FailureHandler or returns an HTTP 401 Unauthorized, with an empty response body.

This filter manages requests from the *last request sender*, otherwise called the *request from the last hop*, or the *request from a direct client*.

For debugging, configure the AllowOnlyFilter `name`, and add the following logger to `logback.xml`, replacing filter_name with the name:

```
org.forgerock.openig.filter.allow.AllowOnlyFilter.filter_name
```

For more information, see Manage logs.

Usage

```
{
  "name": string,
  "type": "AllowOnlyFilter",
  "config": {
    "rules": [ object, ... ],
    "failureHandler": Handler reference
  }
}
```

Properties

`"rules"`: *array of objects, required*

An array of one or more `rules` configuration objects to specify criteria for the request.

When more than one `rules` configuration object is included in the array, the request must match at least one of the configuration objects.

When more than one property is specified in the `rules` configuration (for example, `from` and `destination`) the request must match criteria for each property.

```
{
  "rules": [
    {
      "name": configuration expression<string>,
      "from": [ object, ... ],
      "destination": [ object, ... ],
      "when": configuration expression<boolean>
    },
    ...
  ]
}
```

`"name"`: *configuration expression<string>, optional*

A name for the `rules` configuration. When logging is configured for the AllowOnlyFilter, the rule name appears in the logs.

`"from"`: *array of objects, required*

An array of one or more `from` configuration objects to specify criteria about the last request sender (the direct client).

When more than one `from` configuration object is included in the array, the last request sender must match at least one of the configuration objects.

When both `ip` and `certificate` properties are included in the configuration, the last request sender must match criteria for both properties.

```
"from": [
  {
    "ip": {
      "list": [configuration expression<string>, ...],
      "resolver": runtime expression<string>
    },
    "certificate" : {
      "subjectDNs" : Pattern[]
    }
  },
  ...
]
```

`"ip"`: *object, optional*

Criteria about the IP address of the last request sender.

`"list"`: *array of configuration expression<strings>, required:*

An array of IP addresses or IP address ranges, using IPv4 or IPv6, and CIDR notation. The following example includes different formats:

```
"list": ["127.0.0.1", "::1", "192.168.0.0/16", "1234::/16"]
```

The IP address of the last request sender must match at least one of the specified IP addresses or IP address ranges.

`"resolver"`: *runtime expression<string>, optional:*

An expression that returns an IP address as a string. The following example returns an IP address from the first item in `X-Forwarded-For`:

```
"resolver": "${request.headers['X-Forwarded-For'][0]}"
```

Default: Resolve the IP address from the following items, in the following order:

1. If there is a `Forwarded` header, use the IP address of the last hop.
2. Otherwise, if there is an `X-Forwarded-For` header, use the IP address of the last hop.
3. Otherwise, use the IP address of the connection.

`"certificate"`: *array of objects, optional*

An array of `certificate` configuration objects that specify criteria about the certificate of the last request sender.

`"subjectDNs"`: *array of patterns, required:*

An array of patterns to represent the expected distinguished name of the certificate subject, the `subjectDN`. The `subjectDN` of the last request sender must match at least one of the patterns.

`"destination"`: *array of objects, optional*

An array of `destination` configuration objects to specify criteria about the request destination.

When more than one `destination` configuration object is included in the array, the request destination must match at least one of the configuration objects.

When more than one property is specified in the `destination` configuration, for example `hosts` and `ports`, the request destination must match criteria for each property.

```
"destination": [
  {
    "hosts": [pattern, ... ],
    "ports": [configuration expression<string>, ... ],
    "methods": [configuration expression<string>, ... ],
    "paths": [pattern, ... ]
  },
  ...
]
```

`"hosts"`: *array of patterns, optional*

An array of *case-insensitive* patterns to match the `request.host` attribute. Patterns are matched with the Java Pattern class.

When this property is configured, the request destination must match at least one host pattern in the array.

Default: Any host is allowed.

`"ports"`: *array of configuration expression<strings>, optional*

An array of strings to match the `request.port` attribute. Specify values in the array as follows:

- Array of single ports, for example `["80", "90"]`.
- Array of port ranges, for example `["100:200"]`.
- Array of single ports and port ranges, for example `["80", "90", "100:200"]`.

When this property is configured, the destination port must match at least one entry in the array.

Default: Any port is allowed.

`"methods"`: *array of configuration expression<strings>, optional*

An array of HTTP methods to match the `request.method` attribute.

When this property is configured, the request method must match at least one method in the array.

Default: Any method is allowed.

`"paths"`: *array of patterns, optional*

An array of *case-sensitive* patterns to match the `request.url_path` attribute. Patterns are matched with the Java Pattern class.

When this property is configured, the destination path must match at least one path in the array.

Default: Any path is allowed.

`"when"`: *runtime expression<boolean>, optional*

A flag to indicate that the request meets a condition. When `true`, the request is allowed.

The following condition is met when the first value of `h1` is `1`:

```
"when": "${request.headers['h1'][0] == '1'}"
```

Default: `${true}`

`"failureHandler"`: *Handler reference, optional*

Handler to treat the request if none of the declared rules are satisfied.

Provide either the name of a Handler object defined in the heap or an inline Handler configuration object.

Default: HTTP 401 Unauthorized, with an empty response body.

See also Handlers.

Examples

In the following example, a request is authorized if the last request sender satisfies *either* of the following conditions:

- Certificate subjectDN matches `.*CN=test$` *or* `CN=me`, *and* the IP address is in the range 1.2.3.0/24.
- IP address is 123.43.56.8.

```
"from": [
   {
     "certificate": {
          "subjectDNs": [".*CN=test$", "CN=me"]
     },
     "ip": {
       "list": ["1.2.3.0/24"]
     }
   },
   {
     "ip": {
       "list": ["123.43.56.8"]
     }
   },
]
```

In the following example, a request is authorized if the request destination satisfies *all* of the following conditions:

- The host is `myhost1.com` *or* `www.myhost1.com`
- The port is `80`.
- The method is `POST` *or* `GET`
- The path matches `/user/*`.

```
"destination": [
  {
    "hosts": ["myhost1.com", "www.myhost1.com"],
    "ports": ["80"],
    "methods": ["POST", "GET"],
    "paths": ["/user/*"]
  }
]
```

The following example authorizes a request to continue processing if the requests meets the conditions set by *either* `rule1` or `rule2`:

```
{
  "type": "AllowOnlyFilter",
  "config": {
    "rules": [
      {
        "name": "rule1",
        "from": [
          {
            "certificate": {
              "subjectDNs": [".*CN=test$", "CN=me"]
            },
            "ip": {
              "list": ["1.2.3.0/24"]
            }
          }
        ],
        "destination": [
          {
            "hosts": ["myhost1.com", "www.myhost1.com"],
            "ports": ["80"],
            "methods": ["POST", "GET"],
            "paths": ["/user/*"]
          }
        ],
        "when": "${request.headers['h1'][0] == '1'}"
      },
      {
        "name":"rule2",
        "when": "${request.headers['h1'][0] == '2'}"
      }
    ]
  }
}
```

More information

org.forgerock.openig.filter.allow.AllowOnlyFilter

AmSessionIdleTimeoutFilter

Forces the revocation of AM sessions that have been idle for a specified duration. The AmSessionIdleTimeoutFilter issues an authenticated and encrypted JWT to track activity on the AM session and conveys it within a persistent cookie.

To help honor timeout, the persistent cookie is configured to expire at the same time as the tracking token. Without a persistent cookie, the browser is more likely to clear the side-car cookie and IG is more likely to consider the session as timed out.

The tracking token contains the following parts:

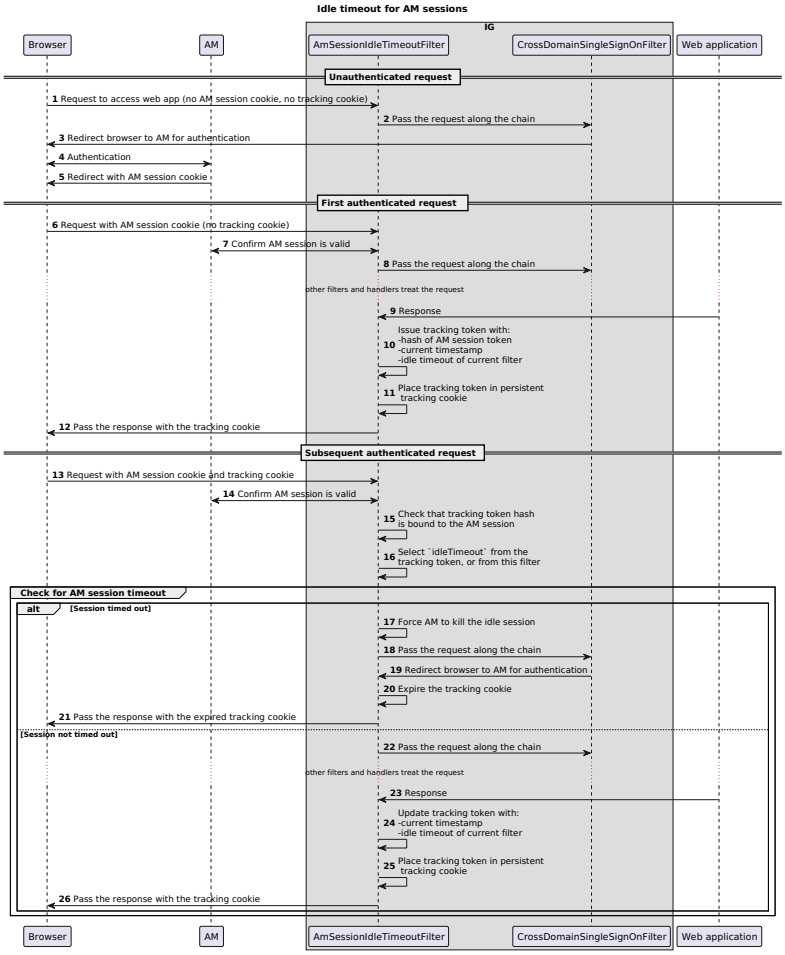
- The time when the user was last active
- A hash of the AM session cookie, used to bind the tracking token to the AM session cookie
- The idle timeout

Multiple filter instances can share the same tracking token, for example, in a clustered IG configuration, or when a federation of applications protected by authentication filters need to have a flexible idle timeout strategy.

AmSessionIdleTimeoutFilter requires the following configuration:

- In AM, client-side sessions must be enabled for the realm in which the tracking token operates. See Configure client-side sessions in AM's *Sessions Guide*.
- In AM, client-side session denylisting must be enabled. See Configure client-side session denylisting in AM's *Sessions Guide*.
- The AmSessionIdleTimeoutFilter must be placed in a route before a filter that uses the AM session token, such as a SingleSignOnFilter or PolicyEnforcementFilter.
- In production environments, and when multiple AmSessionIdleTimeoutFilters use the same tracking token, the encryption must not rely on the default configuration. It must be configured identically on each filter that uses the tracking token.

The following image shows the flow of information when an AmSessionIdleTimeoutFilter sits in front of a CrossDomainSingleSignOnFilter, to manage AM session timeout.



[1-5] When the AmSessionIdleTimeoutFilter receives an unauthenticated request, it passes the request along the chain, and the CrossDomainSingleSignOnFilter manages authentication.

[6-8] When the AmSessionIdleTimeoutFilter receives an authenticated request, it checks that the AM session token is valid, and then passes the request along the chain.

[9-10] If the AM session was valid, the AmSessionIdleTimeoutFilter issues a tracking token on the response flow, containing the following information:

- Hash of the AM session token
- Current timestamp
- Idle timeout of the current filter

If the AM session was not valid, the AmSessionIdleTimeoutFilter does nothing on the response flow.

[11-12] The AmSessionIdleTimeoutFilter places the tracking token in persistent tracking cookie, and sends it with the response, to be used in the next request.

[13-15] When the same or another AmSessionIdleTimeoutFilter receives an authenticated request with a tracking token, it checks that the AM session token is valid, and checks that tracking token hash is bound to the AM session.

[16] Depending on the strategy set by `idleTimeoutUpdate`, the AmSessionIdleTimeoutFilter selects the value for `idleTimeout` from the tracking token (set by the AmSessionIdleTimeoutFilter in a previous request) or from its own value of `idleTimeout` (if this is a different instance of AmSessionIdleTimeoutFilter).

The AmSessionIdleTimeoutFilter checks for AM session timeout. If the last activity time plus the idle timeout is before the current time, the session has timed out. For example, a session with the following values has timed out:

- last activity time: 15h30 today
- idle timeout: 5 mins
- current time: 15h40

[17-21] The AM session has timed out, so the AmSessionIdleTimeoutFilter does the following:

- Forces AM to revoke the session.
- Passes the request along the chain.
- Expires the tracking cookie on the response flow, and sends it with the response.

[22-26] The session has not timed out, so the AmSessionIdleTimeoutFilter does the following:

- Passes the request along the chain.
- Updates the tracking token on the response flow, with the current timestamp and the value for `idleTimeOut`, using the same value for that was selected in step 16.
- Places the tracking token in a persistent tracking cookie, and sends it with the response, to be used in the next request.

Usage

```
{
  "name": string,
   "type": "AmSessionIdleTimeoutFilter",
   "config": {
     "amService": AmService reference,
     "idleTimeout": configuration expression<duration>,
     "sessionToken": runtime expression<string>,
     "idleTimeoutUpdate": configuration expression<enumeration>,
     "secretsProvider": SecretsProvider reference,
     "encryptionSecretId": configuration expression<secret-id>,
     "encryptionMethod": configuration expression<string>,
     "cookie": object
   }
}
```

Properties

`"amService"`: *AmService reference, required*

The AmService that refers to the AM instance that issue tracked session token.

`"idleTimeout"`: *configuration expression<duration>, required*

The time a session can be inactive before it is considered as idle.

When an AmSessionIdleTimeoutFilter creates the tracking token, the token's value for `idleTimeout` is set by this property. When a different AmSessionIdleTimeoutFilter accesses the same tracking token, depending on the strategy set by `idleTimeoutUpdate`, the token's value for `idleTimeout` can be updated by the second AmSessionIdleTimeoutFilter.

`"sessionToken"`: *runtime expression<string>, optional*

The location of the AM session token in the request. The following example accesses the first value of the request cookie `iPlanetDirectoryPro`:

```
"sessionToken": "${request.cookies['iPlanetDirectoryPro'][0].value}"
```

For more information, refer to Find the AM session cookie name.

Default: `${request.cookies['<cookie name defined in the referenced AmService>'][0].value}`

`idleTimeoutUpdate`: *configuration expression<enumeration>, required*

When multiple AmSessionIdleTimeoutFilters use the same tracking token, this property selects whether to use the `idleTimeout` from this filter or from the tracking token.

Use one of the following values:

- `NEVER`: Use the idle timeout from the tracking token, and ignore the idle timeout from this filter.
- `ALWAYS`: Use the idle timeout from this filter, and ignore the idle timeout from the tracking token.

- `INCREASE_ONLY`: Compare the idle timeout from this filter and the tracking token, and use the longest value.
- `DECREASE_ONLY`: Compare the idle timeout from this filter and the tracking token, and use the shortest value.

Default: `ALWAYS`

`"secretsProvider"`: *SecretsProvider reference, optional*

The SecretsProvider to query for secrets to encrypt the tracking token.

`"encryptionSecretId"`: *configuration expression<secret-id>, optional*

The secret ID of the encryption key used to encrypt the tracking cookie.

This secret ID must point to a CryptoKey`.

In production environments, and when multiple AmSessionIdleTimeoutFilters use the same tracking cookie, the encryption must not rely on the default configuration. It must be configured identically on each filter that uses the cookie.

Authenticated encryption is achieved with a symmetric encryption key. Therefore, the secret must refer to a symmetric key.

For more information, refer to RFC 5116.

Default: When no `secretsProvider` is provided, IG generates a random symmetric key for authenticated encryption.

`"encryptionMethod"`: *configuration expression<string>, optional*

The algorithm to use for authenticated encryption. For information about allowed encryption algorithms, refer to RFC 7518: "enc" (Encryption Algorithm) Header Parameter Values for JWE.

Default: A256GCM

`"cookie"`: *object, optional*

Configuration of the activity tracking cookie.

```
{
  "name": configuration expression<string>,
  "domain": configuration expression<string>,
  "httpOnly": configuration expression<boolean>,
  "path": configuration expression<string>,
  "sameSite": configuration expression<enumeration>,
  "secure": configuration expression<boolean>
}
```

`"name"`: *configuration expression<string>, optional*

The cookie name.

Default: `x-ig-activity-tracker`

`"domain"`: *configuration expression<string>, optional*

Domain to which the cookie applies.

Default: The fully qualified hostname of the IG host.

`"httpOnly"`: *configuration expression<boolean>, optional*

Flag to mitigate the risk of client-side scripts accessing protected cookies.

Default: `true`

`"path"`: *configuration expression<string>, optional*

Path to apply to the cookie.

Default: `/`

`"sameSite"`: *configuration expression<enumeration>, optional*

Options to manage the circumstances in which a cookie is sent to the server. Use one of the following values to reduce the risk of CSRF attacks:

- `STRICT` : Send the cookie only if the request was initiated from the cookie domain. Not case-sensitive. Use this value to reduce the risk of cross-site request forgery (CSRF) attacks.
- `LAX` : Send the cookie only with GET requests in a first-party context, where the URL in the address bar matches the cookie domain. Not case-sensitive. Use this value to reduce the risk of cross-site request forgery (CSRF) attacks.
- `NONE` : Send the cookie whenever a request is made to the cookie domain. With this setting, consider setting `secure` to `true` to prevent browsers from rejecting the cookie. For more information, refer to SameSite cookies.

Default: Null

`"secure"`: *configuration expression<boolean>, optional*

Flag to limit the scope of the cookie to secure channels.

Default: `false`

Example

```
{
  "type": "AmSessionIdleTimeoutFilter",
  "config": {
    "sessionToken": "${request.cookies['iPlanetDirectoryPro'][0].value}",
    "amService": "AmService",
    "idleTimeout": "1 minute",
    "idleTimeoutUpdate": "ALWAYS",
    "cookie": {
      "name": "x-ig-activity-tracker",
      "domain": null,
      "path": "/",
      "secure": false,
      "httpOnly": true,
      "sameSite": null
    },
   "secretsProvider": "secrets-provider-ref",
   "encryptionMethod": "A256GCM",
   "encryptionSecretId": "crypto.key.secret.id"
  }
}
```

More information

org.forgerock.openig.openam.session.AmSessionIdleTimeoutFilter

AssignmentFilter

Verifies that a specified condition is met. If the condition is met or if no condition is specified, the value is assigned to the target. Values can be assigned before the request is handled and after the response is handled.

Usage

```
{
  "name": string,
  "type": "AssignmentFilter",
  "config": {
    "onRequest": [
      {
        "condition": runtime expression<boolean>,
        "target": lvalue-expression,
        "value": runtime expression
      }, ...
    ],
    "onResponse": [
      {
        "condition": runtime expression<boolean>,
        "target": lvalue-expression,
        "value": runtime expression
      }, ...
    ]
  }
}
```

Properties

`"onRequest"`: *array of objects, optional*

Defines a list of assignment bindings to evaluate before the request is handled.

`"onResponse"`: *array of objects, optional*

Defines a list of assignment bindings to evaluate after the response is handled.

`"condition"`: *runtime expression<boolean>, optional*

If the expression evaluates `true`, the value is assigned to the target.

Default: `${true}`

`"target"`: *<lvalue-expression>, required*

Expression that yields the target object whose value is to be set.

`"value"`: *runtime expression<object> , optional*

The value to be set in the target. The value can be a string, information from the context, or even a whole map of information.

Examples

Add info to a session

The following example assigns a value to a session. Add the filter to a route to prevent IG from clearing up empty JWTSession cookies:

```
{
  "type": "AssignmentFilter",
  "config": {
    "onRequest": [{
      "target": "${session.authUsername}",
      "value": "I am root"
    }]
  }
}
```

Capture and store login credentials

The following example captures credentials and stores them in the IG session during a login request. Notice that the credentials are captured on the request but are not marked as valid until the response returns a positive 302. The credentials could then be used to log a user in to a different application:

```
{
  "name": "PortalLoginCaptureFilter",
  "type": "AssignmentFilter",
  "config": {
    "onRequest": [
      {
        "target": "${session.authUsername}",
        "value": "${request.queryParams['username'][0]}"
      },
      {
        "target": "${session.authPassword}",
        "value": "${request.queryParams['password'][0]}"
      },
      {
        "comment": "Authentication has not yet been confirmed.",
        "target": "${session.authConfirmed}",
        "value": "${false}"
      }
    ],
    "onResponse": [
      {
        "condition": "${response.status.code == 302}",
        "target": "${session.authConfirmed}",
        "value": "${true}"
      }
    ]
  }
}
```

More information

org.forgerock.openig.filter.AssignmentFilter

AuthorizationCodeOAuth2ClientFilter

Uses OAuth 2.0 delegated authorization to authenticate end users. The filter can act as an OpenID Connect relying party or as an OAuth 2.0 client.

AuthorizationCodeOAuth2ClientFilter performs the following tasks:

- Allows the user to select an Authorization Server from one or more static client registrations or by discovery and dynamic registration.

  In static client registration, Authorization Servers are provided by Issuer, and registrations are provided by ClientRegistration.

- Redirects the user through the authentication and authorization steps of an OAuth 2.0 authorization code grant, which results in the Authorization Server returning an access token to the filter.

- When an authorization grant succeeds, injects the access token data into a configurable target in the context so that subsequent filters and handlers can access the access token. Subsequent requests can use the access token without authenticating again.

- When an authorization grant fails, the filter injects information about the failure into the OAuth2FailureContext, which is provided to the `failureHandler` object.

Service URIs

Service URIs are constructed from the `clientEndpoint`, as follows:

`clientEndpoint/login/?discovery=user-input&goto=url`

Discover and register dynamically with the end user's OpenID Provider or with the client registration endpoint as described in RFC 7591, using the value of user-input.

After successful registration, redirect the end user to the provider for authentication and authorization consent. Then redirect the user agent back to the callback client endpoint, and then the goto URI.

The goto URL must use the same scheme, host, and port as the original URI, or be a relative URI (just the path). Otherwise, the request fails with an error.

To redirect a request to a site that does not meet the goto URL criteria, change the original URI by using a ForwardedRequestFilter.

`clientEndpoint/login?registration=clientId&issuer=issuerName&goto=url`

Redirect the end user for authorization with a registration defined by the ClientRegistration properties `clientId` and `issuerName`.

The provider corresponding to the registration then authenticates the end user and obtains authorization consent before redirecting the user agent back to the callback client endpoint.

If successful, the filter saves the authorization state in the session and redirects the user agent to the goto URL.

The goto URL must use the same scheme, host, and port as the original URI, or be a relative URI (just the path). Otherwise, the request fails with an error.

To redirect a request to a site that does not meet the goto URL criteria, change the original URI by using a ForwardedRequestFilter.

`clientEndpoint/logout?goto=url`

Remove the authorization state for the end user, and redirect the request to the goto URL.

The goto URL must use the same scheme, host, and port as the original URI, or be a relative URI (just the path). Otherwise, the request fails with an error.

To redirect a request to a site that does not meet the goto URL criteria, change the original URI by using a ForwardedRequestFilter.

If no goto URL is specified in the request, use `defaultLogoutGoto`.

`clientEndpoint/callback`

Handle the callback from the OAuth 2.0 Authorization Server occuring as part of the authorization process.

If the callback is handled successfully, the filter saves the authorization state in the context at the specified target location and redirects to the URL provided to the login endpoint during login.

Other request URIs

Restore the authorization state in the specified target location, and call the next filter or handler in the chain.

Usage

```
{
  "name": string,
  "type": "AuthorizationCodeOAuth2ClientFilter",
  "config": {
    "clientEndpoint": runtime expression<uri string>,
    "failureHandler": Handler reference,
    "loginHandler": Handler reference,
    "registrations": [ ClientRegistration reference, ... ],
    "metadata": object,
    "cacheExpiration": configuration expression<duration>,
    "executor": ScheduledExecutorService reference,
    "target": lvalue-expression,
    "defaultLoginGoto": runtime expression<url>,
    "defaultLogoutGoto": runtime expression<url>,
    "requireHttps": configuration expression<boolean>,
    "requireLogin": configuration expression<boolean>,
    "revokeOauth2TokenOnLogout": configuration expression<boolean>,
    "openIdEndSessionOnLogout": configuration expression<boolean>,
    "prompt": configuration expression<string>,
    "issuerRepository": Issuer repository reference,
    "discoveryHandler": Handler reference,
    "discoverySecretId": configuration expression<secret-id>,
    "tokenEndpointAuthMethod": configuration expression<enumeration>,
    "tokenEndpointAuthSigningAlg": configuration expression<string>,
    "oAuth2SessionKey": configuration expression<string>,
    "secretsProvider": SecretsProvider reference
  }
}
```

Properties

`"clientEndpoint"`: *runtime expression*<*url*>, *required*

The URI to the client endpoint.

So that routes can accept redirects from the Authorization Server to the callback endpoint, the `clientEndpoint` must be the same as the route condition or a sub path of the route condition. For example:

- The same as the route condition:

```
"condition": "${find(request.uri.path, '^/discovery')}"
```

```
"clientEndpoint": "/discovery"
```

- As a sub path of the route condition:

```
"condition": "${find(request.uri.path, '^/home/id_token')}"
```

```
"clientEndpoint": "/home/id_token/sub-path"
```

Service URIs are constructed from the `clientEndpoint`. For example, when `clientEndpoint` is `openid`, the service URIs are `/openid/login`, `/openid/logout`, and `/openid/callback`. These endpoints are implicitly reserved, and attempts to access them directly can cause undefined errors.

The result of the expression must be a string that represents a valid URI, but is not a real `java.net.URI` object. For example, it would be incorrect to use `${request.uri}`, which is not a String but a MutableUri.

See also Expressions.

`"failureHandler"`: *Handler reference, required*

An inline handler configuration object, or the name of a handler object defined in the heap.

When the OAuth 2.0 Resource Server denies access to a resource, the failure handler can be invoked only if the error response contains a WWW-Authenticate header (meaning that there was a problem with the OAuth 2.0 exchange). All other responses are forwarded to the user agent without invoking the failure handler.

If the value of the WWW-Authenticate header is `invalid_token`, the AuthorizationCodeOAuth2ClientFilter tries to refresh the access token:

- If the token is refreshed, the AuthorizationCodeOAuth2ClientFilter tries again to access the protected resource.
- If the token is not refreshed, or if the second attempt to access the protected resource fails, the AuthorizationCodeOAuth2ClientFilter invokes the failure handler.

Consider configuring the handler to access information in OAuth2FailureContext.

`"loginHandler"`: *Handler reference, required if there are zero or multiple client registrations, optional if there is one client registration*

The handler to invoke when the user must select a registered identity provider for login. When `registrations` contains only one client registration, this handler is optional but is displayed if specified.

Provide the name of a Handler object defined in the heap or an inline handler configuration object.

When you use `loginHandler` in AuthorizationCodeOAuth2ClientFilter, retrieve the original target URI for the request from one of the following contexts:

- `originalUri` in IdpSelectionLoginContext
- `originalUri` in UriRouterContext (deprecated)
- `request.uri` (deprecated)

`"registrations"`: *array of ClientRegistration references optional*

List of client registrations to authenticate IG to the Authorization Server.

The value represents a static client registration with an Authorization Server, as described in ClientRegistration.

`"metadata"`: *<object>, required for dynamic client registration and ignored otherwise*

The values of the object are evaluated as configuration expression<strings>.

This object holds client metadata as described in OpenID Connect Dynamic Client Registration 1.0, and optionally a list of scopes. See that document for additional details and a full list of fields.

This object can also hold client metadata as described in RFC 7591, OAuth 2.0 Dynamic Client Registration Protocol. See that RFC for additional details.

The following partial list of metadata fields is not exhaustive, but includes metadata that is useful with AM as OpenID Provider:

`"redirect_uris"`: *array of configuration expression<url>, required*

The array of redirection URIs to use when dynamically registering this client.

One of the registered values **must** match the `clientEndpoint`.

`"client_name"`: *configuration expression<string>, optional*

Name of the client to present to the end user.

`"scope"`: *_configuration expression<string>, optional*

Space-separated string of scopes to request of the OpenID Provider, for example:

```
"scope": "openid profile"
```

This property is available for dynamic client registration with AM, or with Authorization Servers that support RFC 7591, *OAuth 2.0 Dynamic Client Registration Protocol*

`"cacheExpiration"`: *configuration expression<duration>, optional*

Duration for which to cache user-info resources.

IG lazily fetches user info from the OpenID provider. In other words, IG only fetches the information when a downstream Filter or Handler uses the user info. Caching allows IG to avoid repeated calls to OpenID providers when reusing the information over a short period.

Default: 10 minutes

Set this to disabled or zero to disable caching. When caching is disabled, user info is still lazily fetched.

`"executor"`: *ScheduledExecutorService reference, optional*

A ScheduledExecutorService to schedule the execution of tasks, such as the eviction of entries in the OpenID Connect user information cache.

Default: `ScheduledExecutorService`

`"target"`: *lvalue-expression*, *optional*

An expression that yields the target object. Downstream filters and handlers can use data in the target to enrich the existing request or create a new request.

When the `target` is `openid`, the following information can be provided in `${attributes.openid}`:

- `access_token`: Value of the OAuth 2.0 access token
- `scope`: Scopes associated with the OAuth 2.0 access token
- `token_type`: Authentication token type; for example, `Bearer`
- `expires_in`: Number of milliseconds until the OAuth 2.0 access token expires
- `id_token`: Value of the OpenID Connect token
- `id_token_claims`: Claims used in the OpenID Connect token
- `client_endpoint`: URL to the client endpoint
- `client_registration`: Client ID of the OAuth 2.0 client that enables IG to communicate as an OAuth 2.0 client with an authorization server
- `user_info`: Profile attributes of an authenticated user; for example, `sub`, `name`, `family_name`

Data is provided to the target as follows:

- If the authorization process completes successfully, the AuthorizationCodeOAuth2ClientFilter injects the authorization state data into the target. In the following example, a downstream StaticRequestFilter retrieves the username and password from the target to log the user in to the sample application.

```
{
  "type": "StaticRequestFilter",
  "config": {
    "method": "POST",
    "uri": "http://app.example.com:8081/login",
    "form": {
      "username": [
        "${attributes.openid.user_info.sub}"
      ],
       "password": [
        "${attributes.openid.user_info.family_name}"
      ]
    }
  }
}
```

  For information about setting up this example, refer to Authenticate Automatically to the Sample Application.

- If the failure handler is invoked, the target can be populated with information such as the exception, client registration, and error, as described in "failureHandler" in this reference page.

  Default: `${attributes.openid}`

  See also Expressions.

`"defaultLoginGoto"`: *runtime expression<url>,optional*

After successful authentication and authorization, if the user accesses the `clientEndpoint/login` endpoint without providing a landing page URL in the `goto` parameter, the request is redirected to this URI.

The goto URL must use the same scheme, host, and port as the original URI, or be a relative URI (just the path). Otherwise, the request fails with an error.

To redirect a request to a site that does not meet the goto URL criteria, change the original URI by using a ForwardedRequestFilter.

The result of the expression must be a string that represents a valid URI, but is not a real `java.net.URI` object. For example, it would be incorrect to use `${request.uri}`, which is not a String but a MutableUri.

Default: return an empty page.

`"defaultLogoutGoto"`: *runtime expression<url>,optional*

If the user accesses the `clientEndpoint/logout` endpoint without providing a goto URL, the request is redirected to this URI.

The goto URL must use the same scheme, host, and port as the original URI, or be a relative URI (just the path). Otherwise, the request fails with an error.

To redirect a request to a site that does not meet the goto URL criteria, change the original URI by using a ForwardedRequestFilter.

The result of the expression must be a string that represents a valid URI, but is not a real `java.net.URI` object. For example, it would be incorrect to use `${request.uri}`, which is not a String but a MutableUri.

Default: return an empty page.

`"requireHttps"`: *configuration expression<boolean>, optional*

Whether to require that original target URI of the request uses the HTTPS scheme.

If the received request doesn't use HTTPS, it is rejected.

Default: true.

`"requireLogin"`: *configuration expression<boolean>, optional*

Whether to require authentication for all incoming requests.

Default: true.

`"revokeOauth2TokenOnLogout"`: *configuration expression<boolean>, optional*

When `true`, call the `revocationEndpoint` defined in Issuer to revoke the access token or refresh token issued by the Authorization Server during login.

If this property is `false` or if `revocationEndpoint` in Issuer isn't defined, IG doesn't revoke the tokens.

Processing errors generate warnings in the logs but don't break the logout flow.

Default: `false`.

`"openIdEndSessionOnLogout"`: *configuration expression<boolean>, optional*

When `true`, redirect the user to the `endSessionEndpoint` defined in Issuer to log the user out of the Authorization Server. Use this properties to initiate logout from an OpenID Connect resource provider.

If this property is `false` or if `endSessionEndpoint` in Issuer isn't defined, IG doesn't redirect the user to log the user out of the authorization server.

If the user accesses the `endSessionEndpoint` endpoint without providing a goto URL, IG redirects the request to the `defaultLogoutGoto`.

For more information, refer to OpenID Connect Session Management.

Default: `false`

`"prompt"`: *configuration expression<string>, optional*

A space-separated, case-sensitive list of strings that indicate whether to prompt the end user for authentication and consent. Use in OIDC flows only.

Refer to the Authorization Server documentation for information about supported `prompt` values. For example, refer to prompt in Identity Cloud's *OAuth 2.0 guide* or prompt in AM's *OAuth 2.0 guide*.

IG provides the following values:

- `none`: Don't display authentication or consent pages. Don't use this value in the same list as `login`, `consent`, or `select_account`.
- `login`: Prompt the end user to reauthenticate even if they have a valid session on the Authorization Server.
- `consent`: Prompt the end user to consent before returning information to the Client, even if they have already consented in the session.
- `select_account`: Prompt the end user to select a user account.

Example: Prompt the end user to reauthenticate

```
"prompt": "login"
```

Example: Prompt the end user to reauthenticate and consent

```
"prompt": "login consent"
```

`"issuerRepository"`: *Issuer repository reference, optional*

A repository of OAuth 2.0 issuers, built from discovered issuers and the IG configuration.

Provide the name of an IssuerRepository object defined in the heap.

Default: Look up an issuer repository named `IssuerRepository` in the heap. If none is explicitly defined, then a default one named `IssuerRepository` is created in the current route.

See also IssuerRepository.

`"discoveryHandler"`: *Handler reference, optional*

Use this property for discovery and dynamic registration of OpenID Connect clients.

Provide either the name of a Handler object defined in the heap or an inline Handler configuration object. Usually, set this to the name of a ClientHandler configured in the heap or a chain that ends in a ClientHandler.

Default: The default ClientHandler.

See also Handlers, ClientHandler.

`"discoverySecretId"`: *configuration expression<secret-id>, required for discovery and dynamic registration*

Use this property for discovery and dynamic registration of OAuth 2.0 clients.

This secret ID must point to a CryptoKey.

Specifies the secret ID of the secret that is used to sign a JWT before the JWT is sent to the Authorization Server.

If `discoverySecretId` is used, then the `tokenEndpointAuthMethod` is always `private_key_jwt`.

`"tokenEndpointAuthMethod"`: *configuration expression<enumeration>, optional*

Use this property for discovery and dynamic registration of OAuth 2.0 clients.

The authentication method with which a client authenticates to the authorization server or OpenID provider at the token endpoint. For information about client authentication methods, refer to OpenID Client Authentication. The following client authentication methods are allowed:

- `client_secret_basic`: Clients that have received a `client_secret` value from the Authorization Server authenticate with the Authorization Server by using HTTP basic access authentication, as in the following example:

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Authorization: Basic ....
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=...
```

- `client_secret_post`: Clients that have received a `client_secret` value from the Authorization Server authenticate with the Authorization Server by including the client credentials in the request body, as in the following example:

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&;
client_id=...&
client_secret=...&
code=...
```

- `private_key_jwt`: Clients send a signed JSON Web Token (JWT) to the Authorization Server. IG builds and signs the JWT, and prepares the request as in the following example:

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=...&
client_id=<clientregistration_id>&
client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer&
client_assertion=PHNhbWxwOl ... ZT
```

  If the Authorization Server doesn't support `private_key_jwt`, a dynamic registration falls back on the method returned by the Authorization Server, for example, `client_secret_basic` or `client_secret_post`.

  If `tokenEndpointAuthSigningAlg` is not configured, the `RS256` signing algorithm is used for `private_key_jwt`.

  Consider these points for identity providers:

  - Some providers accept more than one authentication method.
  - If a provider strictly enforces how the client must authenticate, align the authentication method with the provider.
  - If a provider doesn't support the authentication method, the provider sends an HTTP 400 Bad Request response with an `invalid_client` error message, according to RFC 6749: Error Response.
  - If the authentication method is invalid, the provider sends an `IllegalArgumentException`.

Default: If `discoverySecretId` is used, then the `tokenEndpointAuthMethod` is always `private_key_jwt`. Otherwise, it is `client_secret_basic`.

`"tokenEndpointAuthSigningAlg"`: *configuration expression<string>, optional*

The JSON Web Algorithm (JWA) used to sign the JWT that is used to authenticate the client at the token endpoint. The property is used when `private_key_jwt` is used for authentication.

If the Authorization Server sends a notification to use a different algorithm to sign the JWT, that algorithm is used.

Default: If `discoverySecretId` is used, then the `tokenEndpointAuthSigningAlg` is `RS256`. Otherwise, it is not used.

`"oAuth2SessionKey"`: *configuration expression<string>, optional*

A key to identify an OAuth 2.0 session. The key can be any character string.

To share the same OAuth 2.O session when a user accesses different applications protected by IG, use the same key in each filter.

Default: The complete client endpoint URI. AuthorizationCodeOAuth2ClientFilters do not share OAuth 2.O sessions.

`"secretsProvider"`: *SecretsProvider reference, required if* `discoverySecretId` *is used*

The SecretsProvider to query for passwords and cryptographic keys.

Examples

Refer to the following sections:

- AM as a single OpenID Connect provider
- Use multiple OpenID Connect providers
- Discover and dynamically register with OpenID Connect providers

More information

org.forgerock.openig.filter.oauth2.client.OAuth2ClientFilter

Issuer

ClientRegistration

The OAuth 2.0 Authorization Framework

The OAuth 2.0 Authorization Framework: Bearer Token Usage

OpenID Connect site, in particular the list of standard OpenID Connect 1.0 scope values.

CapturedUserPasswordFilter

Makes an AM password available to IG in the following steps:

- Checks for the presence of the SessionInfoContext context, at `${contexts.amSession}`.
  - If the context isn't present, or if `sunIdentityUserPassword` is `null`, the CapturedUserPasswordFilter collects session info and properties from AM.
  - If the context is present and `sunIdentityUserPassword` is not `null`, the CapturedUserPasswordFilter uses that value for the password.
- The CapturedUserPasswordFilter decrypts the password and stores it in the CapturedUserPasswordContext, at `${contexts.capturedPassword}`.

> **Note**
>
> In Identity Cloud and from AM 7.5, the password capture and replay feature can optionally manage the replay password through AM's secret service. The secret label for the replay password must be `am.authentication.replaypassword.key`.
> For backward compatibility, if a secret isn't defined, is empty, or can't be resolved, AM manages the replay password through the AM system property `am.authentication.replaypassword.key`.

Usage

```
{
  "name": string,
   "type": "CapturedUserPasswordFilter",
   "config": {
     "amService": AmService reference,
     "keySecretId": configuration expression<secret-id>,
     "keyType": configuration expression<string>,
     "secretsProvider": SecretsProvider reference,
     "ssoToken": runtime expression<string>
   }
}
```

Properties

`"amService"`: *AmService reference, required*

The AmService heap object to use for the password. See also, AmService.

`"keySecretId"`: *configuration expression<secret-id>, required*

The secret ID for the key required decrypt the AM password.

This secret ID must point to a CryptoKey` that matches the algorithm in `"keyType"`.

> **Important**
>
> Although secrets of type `GenericSecret` are accepted, their usage is deprecated in this filter. For more information, refer to the Deprecated section of the *Release Notes*.

`"keyType"`: *configuration expression<enumeration>, required*

Algorithm to decrypt the AM password. Use one of the following values:

- `AES` AES for JWT-based AES_128_CBC_HMAC_SHA_256 encryption. For more information, refer to AES_128_CBC_HMAC_SHA_256 in the IETF *JSON Web Algorithms*.
- `DES` for DES/ECB/NoPadding

  > **Important**
  >
  > This value is deprecated, and considered unsecure. For more information, refer to the Deprecated section of the *Release Notes*.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for secrets to decrypt the user password.

`"ssoToken"`: *runtime expression<string>, required*

Location of the AM SSO token.

Default: `${request.cookiesAmService-ssoTokenHeader'][0].value}`, where `AmService-ssoTokenHeader` is the name of the header or cookie where the AmService expects to find SSO tokens.

Examples

The following example route is used to get login credentials from AM in Authenticate with credentials from AM.

```
{
  "name": "04-replay",
  "condition": "${find(request.uri.path, '^/replay')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore",
      "config": {
        "mappings": [
          {
            "secretId": "aes.key",
            "format": {
              "type": "SecretKeyPropertyFormat",
              "config": {
                "format": "BASE64",
                "algorithm": "AES"
              }
            }
          }
        ]
      }
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    },
    {
      "name": "CapturedUserPasswordFilter",
      "type": "CapturedUserPasswordFilter",
      "config": {
        "ssoToken": "${contexts.ssoToken.value}",
        "keySecretId": "aes.key",
        "keyType": "AES",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "amService": "AmService-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${true}",
```

```
              "credentials": "CapturedUserPasswordFilter",
              "request": {
                "method": "POST",
                "uri": "http://app.example.com:8081/login",
                "form": {
                  "username": [
                    "${contexts.ssoToken.info.uid}"
                  ],
                  "password": [
                    "${contexts.capturedPassword.value}"
                  ]
                }
              }
            }
          }
        ],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

More information

org.forgerock.openig.openam.CapturedUserPasswordFilter

org.forgerock.openig.openam.CapturedUserPasswordContext

CapturedUserPasswordContext

SessionInfoFilter

CertificateThumbprintFilter

Extracts a Java certificate from a trusted header or from a TLS connection, computes the SHA-256 thumbprint of that certificate, and makes the thumbprint available for the ConfirmationKeyVerifierAccessTokenResolver. Use this filter to enable verification of certificate-bound access tokens.

CertificateThumbprintFilter computes and makes available the SHA-256 thumbprint of a client certificate as follows:

- Evaluates a runtime expression and yields a `java.security.cert.Certificate`
- Hashes the certificate using SHA-256
- Base64url-encodes the result
- Stores the result in the contexts chain

The runtime expression can access or build a client certificate from any information present at runtime, such as a PEM in a header, or a pre-built certificate.

Use CertificateThumbprintFilter with ConfirmationKeyVerifierAccessTokenResolver when the IG instance is behind the TLS termination point, for example, when IG is running behind a load balancer or other ingress point.

Usage

```
{
  "name": string,
  "type": "CertificateThumbprintFilter",
  "config": {
    "certificate": runtime expression<certificate>,
    "failureHandler": Handler reference,
  }
}
```

Properties

"certificate": *runtime expression*<*certificate*>, *required*

An EL expression which, when evaluated, yields an instance of a `java.security.cert.Certificate`.

Use the following Functions in the expression to define hash, decoding, and certificate format:

- `digestSha256`, to calculate the SHA-256 hash of the certificate.
- `decodeBase64url`, to decode an incoming base64url-encoded string.
- `pemCertificate`, to convert a PEM representation string into a certificate.

See Examples.

"failureHandler": *Handler reference*, *optional*

Handler to treat the request on failure.

Provide an inline handler configuration object or the name of a handler object declared in the heap. See also Handlers.

Default: HTTP 403 Forbidden, the request stops being executed.

Examples

The following example uses the certificate associated with the incoming HTTP connection:

```
{
  "name": "CertificateThumbprintFilter-1",
  "type": "CertificateThumbprintFilter",
  "config": {
    "certificate": "${contexts.client.certificates[0]}"
  }
}
```

The following example is adapted for a deployment with NGINX as the TLS termination, where NGINX fronts IG. NGINX provides the client certificate associated with its own incoming connection in the `x-ssl-client-cert` header. The certificate is encoded as PEM, and then url-encoded:

```
{
  "name": "CertificateThumbprintFilter-2",
  "type": "CertificateThumbprintFilter",
  "config": {
    "certificate": "${pemCertificate(urlDecode(request.headers['x-ssl-client-cert'][0]))}"
  }
}
```

More information

org.forgerock.openig.filter.oauth2.cnf.CertificateThumbprintFilter

CircuitBreakerFilter

Monitors failures. When the number of failures reaches a configured failure threshold, the circuit breaker trips, and the circuit is considered *open*. Calls to downstream filters are stopped, and a runtime exception is returned.

After a configured delay, the circuit breaker is reset, and is the circuit considered *closed*. Calls to downstream filters are restored.

Usage

```
{
  "name": string,
  "type": "CircuitBreakerFilter",
  "config": {
    "maxFailures":  configuration expression<integer>,
    "openDuration": configuration expression<duration>,
    "openHandler": Handler reference,
    "slidingCounter": object,
    "executor":  ScheduledExecutorService reference
  }
}
```

Properties

`"maxFailures"`: *configuration expression<number>, required*

The maximum number of failed requests allowed in the window given by `size`, before the circuit breaker trips. The value must be greater than zero.

`"openDuration"`: *configuration expression<duration>, required*

The duration for which the circuit stays open after the circuit breaker trips. The `executor` schedules the circuit to be closed after this duration.

`"openHandler"`: *Handler reference, optional*

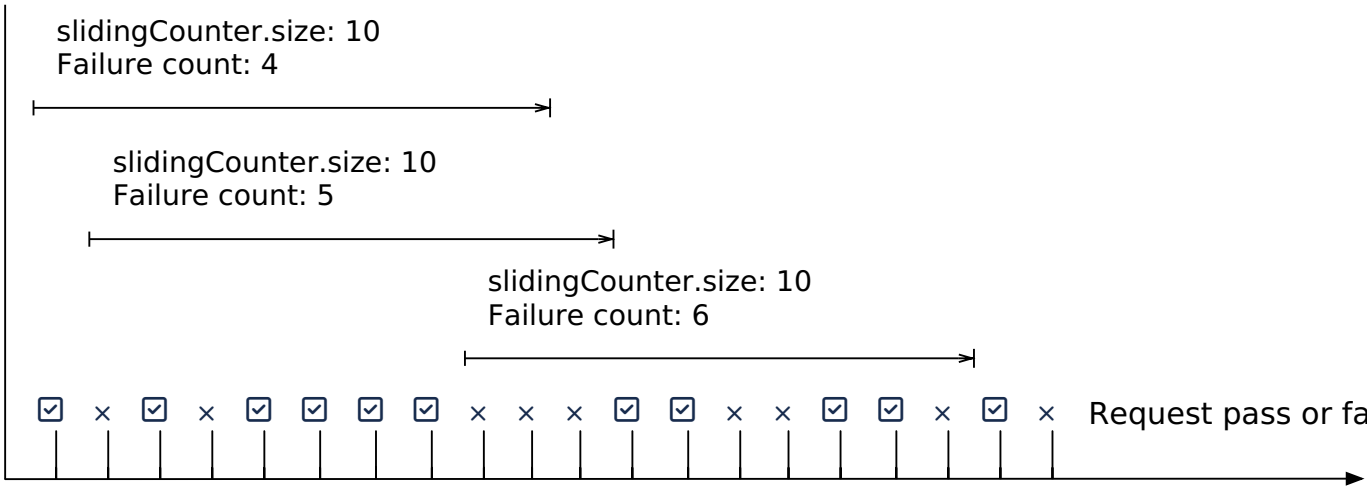
The Handler to call when the circuit is open.

Default: A handler that throws a RuntimeException with a "circuit-breaker open" message.

`"slidingCounter"`: *object*, *optional*

A sliding window error counter. The circuit breaker trips when the number of failed requests in the number of requests given by `size` reaches `maxFailures`.

The following image illustrates how the sliding window counts failed requests:




```
{
  "slidingCounter":  {
    "size": configuration expression<number>
  }
}
```

`"size"`: *configuration expression*<*number*>, *required*

The size of the sliding window in which to count errors.

The value of `size` must be greater than zero, and greater than the value of `maxFailures`, otherwise an exception is thrown.

`"executor"`: *ScheduledExecutorService reference*, *optional*

A ScheduledExecutorService to schedule closure of the circuit after the duration given by `openDuration`.

Default: The default ScheduledExecutorService in the heap

Example

In the following example, the circuit breaker opens after 11 failures in the previous 100 requests, throwing a runtime exception with a "circuit-breaker open" message. The default ScheduledExecutorService in the heap closes the circuit-breaker after 10 seconds.

```
{
  "type": "CircuitBreakerFilter",
  "config": {
    "maxFailures": 10,
    "openDuration": "10 seconds",
    "openHandler": {
      "type": "StaticResponseHandler",
      "config": {
        "status": 500,
        "headers": {
          "Content-Type": [ "text/plain" ]
        },
        "entity": "Too many failures; circuit opened to protect downstream services."
      }
    },
    "slidingCounter": {
      "size": 100
    }
  }
}
```

More information

org.forgerock.openig.filter.circuitbreaker.CircuitBreakerFilter

ClientCredentialsOAuth2ClientFilter

Authenticates OAuth 2.0 clients by using the client's OAuth 2.0 credentials to obtain an access token from an Authorization Server, and injecting the access token into the inbound request as a Bearer Authorization header. The access token is valid for the configured scopes.

The ClientCredentialsOAuth2ClientFilter obtains the client's access token by using the `client_credentials` grant type. Client authentication is provided by the `endpointHandler` property, which uses a client authentication filter, such as ClientSecretBasicAuthenticationFilter. The filter refreshes the access token as required.

Use the ClientCredentialsOAuth2ClientFilter in a service-to-service context, where services need to access resources protected by OAuth 2.0.

Usage

```
{
  "name": string,
  "type": "ClientCredentialsOAuth2ClientFilter",
  "config": {
    "secretsProvider": SecretsProvider reference,
    "tokenEndpoint": configuration expression<url>,
    "scopes": [ configuration expression<string>, ... ],
    "endpointHandler": Handler reference,
    "clientId": configuration expression<sting>, //deprecated
    "clientSecretId": configuration expression<secret-id>, //deprecated
    "handler": Handler reference //deprecated
  }
}
```

Properties

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for passwords and cryptographic keys.

`"tokenEndpoint"`: *configuration expression<url>, required*

The URL to the Authorization Server's OAuth 2.0 token endpoint.

`"scopes"`: *array of configuration expression<strings>, optional*

Array of scope strings to request from the Authorization Server.

Default: Empty, request no scopes.

`"endpointHandler"`: *Handler reference, optional*

The Handler to exchange tokens on the authorization endpoint.

Configure this property as a Chain, using one of the following client authentication filters:

- ClientSecretBasicAuthenticationFilter
- ClientSecretPostAuthenticationFilter
- PrivateKeyJwtClientAuthenticationFilter

```
{
  "name": "myHandler",
  "type": "Chain",
  "config": {
    "handler": "ForgeRockClientHandler",
    "filters": [
      {
        "type": "ClientSecretBasicAuthenticationFilter",
        "config": {
          "clientId": "myConfidentialClient",
          "clientSecretId": "my.client.secret.id",
          "secretsProvider" : "mySystemAndEnvSecretStore",
        }
      }
    ]
  }
}
```

Default: ForgeRockClientHandler

`"clientId"`: *configuration expression<string>, required*

> **Important**
>
> This property is deprecated. Use `endpointHandler` instead. For more information, refer to the Deprecated section of the *Release Notes*.

The ID of the OAuth 2.0 client registered with the Authorization Server.

If you use the deprecated properties, provide `clientId`, `clientSecretId` to obtain the client secret, which authenticates using the `client_secret_basic` method.

`"clientSecretId"`: *configuration expression<secret-id>, required*

> **Important**
>
> This property is deprecated. Use `endpointHandler` instead. For more information, refer to the Deprecated section of the *Release Notes*.

The ID to use when querying the `secretsProvider` for the client secret.

This secret ID must point to a GenericSecret.

`"handler"`: *Handler reference or inline Handler declaration, optional*

> **Important**
>
> This property is deprecated. Use `endpointHandler` instead. For more information, refer to the Deprecated section of the *Release Notes*.

The Handler to use to access the Authorization Server's OAuth 2.0 token endpoint. Provide either the name of a handler object defined in the heap or specify a handler object inline.

Default: ClientHandler

Examples

For an example, refer to Using OAuth 2.0 client credentials.

More information

org.forgerock.openig.filter.oauth2.client.ClientCredentialsOAuth2ClientFilterHeaplet

org.forgerock.openig.filter.oauth2.OAuth2ResourceServerFilterHeaplet

OAuth2ResourceServerFilter

The OAuth 2.0 Authorization Framework

The OAuth 2.0 Authorization Framework: Bearer Token Usage

ClientSecretBasicAuthenticationFilter

Supports client authentication with the method `client_secret_basic`. Clients that have received a `client_secret` value from the Authorization Server authenticate through the HTTP basic access authentication scheme, as in the following example:

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Authorization: Basic ....
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=...
```

Use this filter with an endpoint `Handler` that requires `client_secret_basic` authentication. For example, `endpointHandler` in the OAuth2TokenExchangeFilter or ClientCredentialsOAuth2ClientFilter.

Usage

```
{
  "name": string,
  "type": "ClientSecretBasicAuthenticationFilter",
  "config": {
    "clientId": configuration expression<string>,
    "clientSecretId": configuration expression<secret-id>,
    "secretsProvider": SecretsProvider reference
  }
}
```

Configuration

`"clientId"`: *configuration expression<string>, required*

The OAuth 2.0 client ID to use for authentication.

`"clientSecretId"`: *configuration expression<secret-id>, required*

The OAuth 2.0 client secret to use for authentication.

This secret ID must point to a GenericSecret.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for passwords and cryptographic keys.

Example

```
{
  "name": "ExchangeHandler",
  "type": "Chain",
  "config": {
    "handler": "ForgeRockClientHandler",
    "filters": [
      {
        "type": "ClientSecretBasicAuthenticationFilter",
        "config": {
          "clientId": "serviceConfidentialClient",
          "clientSecretId": "client.secret.id",
          "secretsProvider" : "SystemAndEnvSecretStore-1"
        }
      }
    ]
  }
}
```

ClientSecretPostAuthenticationFilter

Supports client authentication with the method `client_secret_post`. Clients that have received a `client_secret` value from the Authorization Server authenticate by including the client credentials in the request body, as in the following example:

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&;
client_id=...&
client_secret=...&
code=...
```

Use this filter with an endpoint `Handler` that requires `client_secret_post` authentication. For example, `endpointHandler` in the OAuth2TokenExchangeFilter or ClientCredentialsOAuth2ClientFilter.

Usage

```
{
  "name": string,
  "type": "ClientSecretPostAuthenticationFilter",
  "config": {
    "clientId": configuration expression<string>,
    "clientSecretId": configuration expression<secret-id>,
    "secretsProvider": SecretsProvider reference
  }
}
```

Configuration

`"clientId"`: *configuration expression<string>, required*

The OAuth 2.0 client ID to use for authentication.

`"clientSecretId"`: *configuration expression<secret-id>, required*

The OAuth 2.0 client secret to use for authentication.

This secret ID must point to a GenericSecret.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for passwords and cryptographic keys.

ConditionalFilter

Verifies that a specified condition is met. If the condition is met, the request is dispatched to a delegate Filter. Otherwise, the delegate Filter is skipped.

Use `ConditionalFilter` to easily use or skip a Filter depending on whether a condition is met. To easily use or skip a set of Filters, use a `ChainOfFilters` as the delegate Filter and define a set of Filters. For information, refer to ChainOfFilters.

Usage

```
{
    "name": string,
    "type": "ConditionalFilter",
    "config": {
        "condition": runtime expression<boolean>,
        "delegate": Filter reference
   }
}
```

Properties

`"condition"`: *runtime expression<boolean>, required*

If the expression evaluates to `true`, the request is dispatched to the delegate Filter. Otherwise the delegate Filter is skipped.

`"delegate"`: *Filter reference, required*

Filter to treat the request when the condition expression evaluates as `true`.

See also Filters.

Example

The following example tests whether a request finishes with `.js` or `.jpg`:

```
{
  "type": "Chain",
  "config": {
    "filters": [{
      "type": "ConditionalFilter",
      "config": {
        "condition": "${not (find(request.uri.path, '.js$') or find(request.uri.path, '.jpg$'))}",
        "delegate": "mySingleSignOnFilter"
      }
    }],
    "handler": "ReverseProxyHandler"
  }
}
```

If the request is to access a .js file or .jpg file, it skips the delegate SingleSignOnFilter filter declared in the heap, and passes straight to the ReverseProxyHandler.

If the request is to access another type of resource, it must pass through the delegate SingleSignOnFilter for authentication with AM before it can pass to the ReverseProxyHandler.

More information

org.forgerock.openig.filter.ConditionalFilter

ConditionEnforcementFilter

Verifies that a specified condition is met. If the condition is met, the request continues to be executed. Otherwise, the request is referred to a failure handler, or IG returns 403 Forbidden and the request is stopped.

Usage

```
{
    "name": string,
    "type": "ConditionEnforcementFilter",
    "config": {
        "condition": runtime expression<boolean>,
        "failureHandler": Handler reference
    }
}
```

Properties

`"condition"`*: runtime expression<boolean>, required*

If the expression evaluates to `true`, the request continues to be executed.

`"failureHandler"`*: Handler reference, optional*

Handler to treat the request if the condition expression evaluates as `false`.

Provide an inline handler configuration object or the name of a handler object declared in the heap. See also Handlers.

Default: HTTP 403 Forbidden, the request stops being executed.

Example

The following example tests whether a request contains a session username. If it does, the request continues to be executed. Otherwise, the request is dispatched to the `ConditionFailedHandler` failure handler.

```
{
    "name": "UsernameEnforcementFilter",
    "type": "ConditionEnforcementFilter",
    "config": {
        "condition": "${not empty (session.username)}",
        "failureHandler": "ConditionFailedHandler"
    }
}
```

More information

org.forgerock.openig.filter.ConditionEnforcementFilter

ChainOfFilters

Dispatches a request to an ordered list of filters. Use this filter to assemble a list of filters into a single filter that you can then use in different places in the configuration.

A ChainOfFilters can be placed in a configuration anywhere that a filter can be placed.

Unlike `Chain`, `ChainOfFilters` does not finish by dispatching the request to a handler. For more information, refer to Chain.

Usage

```
{
    "name": string,
    "type": "ChainOfFilters",
    "config": {
        "filters": [ Filter reference, ... ]
    }
}
```

Properties

`"filters"`: *array of Filter references, required*

An array of names of filter objects defined in the heap, and inline filter configuration objects.

The chain dispatches the request to these filters in the order they appear in the array.

See also Filters.

Example

```
{
    "name": "MyChainOfFilters",
    "type": "ChainOfFilters",
    "config": {
        "filters": [ "Filter1", "Filter2" ]
   }
}
```

More information

org.forgerock.openig.filter.ChainFilterHeaplet

CookieFilter

Manages, suppresses, and relays cookies for stateful sessions. This filter is not currently compatible with stateless sessions.

Usage

```
{
     "name": string,
     "type": "CookieFilter",
     "config": {
         "managed": [ configuration expression<string>, ... ],
         "suppressed": [ configuration expression<string>, ... ],
         "relayed": [ configuration expression<string>, ... ],
         "defaultAction": configuration expression<enumeration>
     }
}
```

Properties

`"managed"`: *array of configuration expression<strings>, optional*

A list of the names of cookies to be managed.

IG stores cookies from the protected application in the session and manages them as follows:

- Requests with a `Cookie` header: IG removes managed cookies so that protected applications cannot see them.
- Responses with a `Set-Cookie` header: IG removes managed cookies and keeps a copy of them. IG then adds the managed cookies in a `Cookie` header to future requests that traverse the CookieFilter.

`"suppressed"`: *array of configuration expression<strings>, optional*

A list of the names of cookies to be suppressed.

IG removes cookies from the request and response. Use this option to hide domain cookies, such as the AM session cookie, that are used by IG but are not usually used by protected applications.

`"relayed"`: *array of configuration expression<strings>, optional*

A list of the names of cookies to be relayed.

IG transmits cookies freely from the user agent to the remote server, and vice versa.

`"defaultAction"`: *configuration expression<enumeration>, optional*

Action to perform for cookies that do not appear in one of the above lists. Set to `MANAGE`, `SUPPRESS`, or `RELAY`.

If a cookie appears in more than one of the above lists, it is treated in the following order of precedence: `managed`, `suppressed`, `relayed`. For example, if a cookie is in both the `managed` and `relayed` lists, the cookie is managed.

Default: `"MANAGE"`.

More information

org.forgerock.openig.filter.CookieFilter

CorsFilter

Configures policies for cross-origin resource sharing (CORS), to allow cross-domain requests from user agents.

Usage

```
{
   "name": string,
   "type": "CorsFilter",
   "config": {
     "policies": [ object, ... ],
     "failureHandler": Handler reference
   }
}
```

Properties

`"policies"`: *array of objects, required*

One or more policies to apply to the request. A policy is selected when the origin of the request matches the accepted `origins` of the policy.

When multiple policies are declared, they are tried in the order that they are declared, and the first matching policy is selected.

```
{
  "acceptedOrigins": [ configuration expression<url>, ... ] or "*",
  "acceptedMethods": [ configuration expression<string>, ... ] or "*",
  "acceptedHeaders": [ configuration expression<string>, ... ] or "*",
  "exposedHeaders": [ configuration expression<string>, ... ],
  "maxAge": configuration expression<duration>,
  "allowCredentials": configuration expression<boolean>,
  "origins": [ configuration expression<url>, ... ] or "*" //deprecated
}
```

`"acceptedOrigins"`: *array of configuration expression<urls> or `"*"`, required*

A comma-separated list of *origins*, to match the origin of the CORS request. Alternatively, use `*` to allow requests from any URL.

If the request origin is not in the list of accepted origins, the failure handler is invoked or an HTTP 403 Forbidden is returned, and the request stops being executed.

Origins are URLs with a scheme, hostname, and optionally a port number, for example, `http://www.example.com`. If a port number is not defined, origins with no port number or with the default port number (80 for HTTP, 443 for HTTPS) are accepted.

Examples:

```
{
  "acceptedOrigins": [
    "http://www.example.com",
    "https://example.org:8433"
  ]
}
```

```
{
  "acceptedOrigins": "*"
}
```

`"acceptedMethods"`: *array of configuration expression<strings> or `"*"`, optional*

A comma-separated list of case-sensitive HTTP method names that are allowed when making CORS requests. Alternatively, use `*` to allow requests with any method.

In preflight requests, browsers use the `Access-Control-Request-Method` header to let the server know which HTTP method might be used in the actual request.

- If all requested methods are allowed, the requested methods are returned in the preflight response, in the `Access-Control-Allow-Methods` header.
- If any requested method is not allowed, the `Access-Control-Allow-Methods` header is omitted. The failure handler is not invoked, but the user agent interprets the preflight response as a CORS failure.

Examples:

```
{
  "acceptedMethods": [
    "GET",
    "POST",
    "PUT",
    "MyCustomMethod"
  ]
}
```

```
{
  "acceptedMethods": "*"
}
```

Default: All methods are rejected.

`"acceptedHeaders"`: *array of configuration expression<strings> or `"*"`, optional*

A comma-separated list of case-insensitive request header names that are allowed when making CORS requests. Alternatively, use `*` to allow requests with any header.

In preflight requests, browsers use the `Access-Control-Request-Headers` header to let the server know which HTTP headers might be used in the actual request.

- If all requested headers are allowed, the requested headers are returned in the preflight response, in the `Access-Control-Allow-Headers` header.
- If any requested header is not allowed, the `Access-Control-Allow-Headers` header is omitted. The failure handler is not invoked, but the user agent interprets the preflight response as a CORS failure.

Examples:

```
{
  "acceptedHeaders": [
    "iPlanetDirectoryPro",
    "X-OpenAM-Username",
    "X-OpenAM-Password",
    "Accept-API-Version",
    "Content-Type",
    "If-Match",
    "If-None-Match"
  ]
}
```

```
{
  "acceptedHeaders": "*"
}
```

Default: All requested headers are rejected.

`"exposedHeaders"`: *list of configuration expression<string>, optional*

A comma-separated list of case-insensitive response header names that are returned in the `Access-Control-Expose-Headers` header.

Only headers in this list, safe headers, and the following simple response headers are exposed to frontend JavaScript code:

- `Cache-Control`
- `Content-Language`
- `Expires`
- `Last-Modified`
- `Pragma`
- `Content-Type`

Example:

```
{
  "exposedHeaders": [
    "Access-Control-Allow-Origin",
    "Access-Control-Allow-Credentials",
    "Set-Cookie"
  ]
}
```

Default: No headers are exposed.

`"maxAge"`: *configuration expression<duration>, optional*

The maximum duration for which a browser is allowed to cache a preflight response. The value is included in the `Access-Control-Max-Age` header of preflight responses.

When this `maxAge` is greater than the browser's maximum internal value, the browser value takes precedence.

Default: 5 seconds

`"allowCredentials"`: *configuration expression<boolean>, optional*

A flag to allow requests that use credentials, such as cookies, authorization headers, or TLS client certificates.

Set to `true` to set the `Access-Control-Allow-Credentials` header to `true`, and allow browsers to expose the response to frontend JavaScript code.

Default: False

`"origins"`: *list of configuration expression<url> or "*", required*

> **Important**
>
> This property is deprecated; use `acceptedOrigins` instead. For more information, refer to the Deprecated section of the *Release Notes*.

A comma-separated list of origins, to match the origin of the CORS request. Alternatively, use `*` to allow requests from any URL.

Origins are URLs with a scheme, hostname, and optionally a port number, for example, `http://www.example.com`. If a port number is not defined, origins with no port number or with the default port number (80 for HTTP, 443 for HTTPS) are accepted.

`"failureHandler"`: *Handler reference, optional*

Handler invoked during the preflight request, when the request origin does not match any of the `acceptedOrigins` defined in `policies`.

The failure handler is not invoked when requested headers or requested methods are not allowed.

Provide an inline handler configuration object or the name of a handler object declared in the heap. See also Handlers.

Default: HTTP 403 Forbidden, the request stops being executed.

More information

org.forgerock.http.filter.cors.CorsFilter

https://fetch.spec.whatwg.org/#http-cors-protocol

CrossDomainSingleSignOnFilter

When IG and AM are running in the same domain, the SingleSignOnFilter can be used for SSO. When IG and AM are running in different domains, AM cookies aren't visible to IG because of the same-origin policy. The CrossDomainSingleSignOnFilter provides a mechanism to push tokens issued by AM to IG running in a different domain.

When this filter processes a request, it injects the CDSSO token, the session user ID, and the full claims set into the CdSsoContext. If an error occurs during authentication, information is captured in the CdSsoFailureContext.

For an example of how to configure CDSSO and information about the CDSSO data flow, refer to Cross-domain single sign-on.

WebSocket notifications for sessions

When WebSocket notifications are set up for sessions, IG receives a notification from AM when a user logs out of AM, or when the AM session is modified, closed, or times out. IG then evicts entries that are related to the event from the `sessionCache`.

For information about setting up WebSocket notifications, using them to clear the session cache, and including them in the server logs, refer to WebSocket notifications.

Usage

```
{
  "name": string,
  "type": "CrossDomainSingleSignOnFilter",
  "config": {
    "amService": AmService reference,
    "redirectEndpoint": configuration expression<url>,
    "authenticationService": configuration expression<string>,
    "authCookie": object,
    "redirectionMarker": object,
    "defaultLogoutLandingPage": configuration expression<url>,
    "logoutExpression": runtime expression<boolean>,
    "failureHandler": Handler reference,
    "verificationSecretId": configuration expression<secret-id>,
    "secretsProvider": SecretsProvider reference
  }
}
```

Properties

"amService": *AmService reference, required*

The AmService heap object to use. See AmService.

"redirectEndpoint": *configuration expression<url>, required*

The URI to which AM redirects the browser with the authentication token or an authentication error. The filter checks that the authentication was initiated by IG.

Configure this URI to be the same as that in AM.

To make sure the redirect is routed back to the CrossDomainSingleSignOnFilter, include the endpoint in the route condition in one of the following ways:

- As a sub-path of the condition path.

  For example, use the following route condition with the following endpoint:

  ```
  "condition": "${find(request.uri.path, '^/home/cdsso')}"
  ```

  ```
  "redirectEndpoint": "/home/cdsso/callback"
  ```

- To match the route condition.

  For example, use the following route condition with the following endpoint:

  ```
  "condition": "${find(request.uri.path, '^/home/cdsso')}"
  ```

```
"redirectEndpoint": "/home/cdsso"
```

With this route condition, all POST requests on the condition path are treated as AM CDSSO callbacks. Any POST requests that aren't the result of an AM CDSSO callback will fail.

- As a specific path that is not related to the condition path.

  To make sure the redirect is routed back to this filter, include the redirectEndpoint as a path in the filter condition.

  For example, use the following route condition with the following endpoint:

```
"condition": "${find(request.uri.path, '^/home/cdsso/redirect') || find(request.uri.path, '^/ig/
cdssoRedirectUri')}"
```

```
"redirectEndpoint": "/ig/cdssoRedirectUri"
```

`"authenticationService"`: *configuration expression<string>, optional*

The name of an AM authentication tree or authentication chain to use for authentication.

> **Note**
>
> Use only authentication trees with ForgeRock Identity Cloud. Authentication modules and chains are not supported.

Default: AM's default authentication tree.

For more information about authentication trees and chains, refer to Authentication nodes and trees and Authentication modules and chains in AM's *Authentication and SSO* guide.

`"authCookie"`: *object, optional*

The configuration of the cookie used to store the authentication.

```
{
  "name": configuration expression<string>,
  "domain": configuration expression<string>,
  "httpOnly": configuration expression<boolean>,
  "path": configuration expression<string>,
  "sameSite": configuration expression<enumeration>,
  "secure": configuration expression<boolean>
}
```

`"name"`: *configuration expression<string>, optional*

Name of the cookie containing the authentication token from AM.

For security, change the default name of cookies.

Default: `ig-token-cookie`

`"domain"`: *configuration expression<string>, optional*

Domain to which the cookie applies.

Set a domain only if the user agent is able to re-emit cookies on that domain on its next hop. For example, to re-emit a cookie on the domain `example.com`, the user agent must be able to access that domain on its next hop.

Default: The fully qualified hostname of the user agent's next hop.

`"httpOnly"`: *configuration expression<boolean>, optional*

Flag to mitigate the risk of client-side scripts accessing protected cookies.

Default: `true`

`"path"`: *configuration expression<string>, optional*

Path protected by this authentication.

Set a path only if the user agent is able to re-emit cookies on the path. For example, to re-emit a cookie on the path `/home/cdsso`, the user agent must be able to access that path on its next hop.

Default: The path of the request that got the `Set-Cookie` in its response.

`"sameSite"`: *configuration expression<enumeration>, optional*

Options to manage the circumstances in which a cookie is sent to the server. Use one of the following values to reduce the risk of CSRF attacks:

- `STRICT`: Send the cookie only if the request was initiated from the cookie domain. Not case-sensitive.

  Use this value to reduce the risk of cross-site request forgery (CSRF) attacks.

- `LAX`: Send the cookie only with GET requests in a first-party context, where the URL in the address bar matches the cookie domain. Not case-sensitive.

  Use this value to reduce the risk of cross-site request forgery (CSRF) attacks.

- `NONE`: Send the cookie whenever a request is made to the cookie domain. Not case-sensitive.

  With this setting, consider setting `secure` to `true` to prevent browsers from rejecting the cookie. For more information, refer to SameSite cookies.

Default: `LAX`

> **Note**
>
> For CDSSO, set `"sameSite":"none"` and `"secure":"true"`. For security reasons, many browsers require the connection used by the browser to be secure (HTTPS) for `"sameSite":"none"`. Therefore, if the connection used by the browser is not secure (HTTP), the browser might not supply cookies with `"sameSite":"none"`. For more information, refer to Authenticate with CDSSO.

`"secure"`: *configuration expression<boolean>, optional*

Flag to limit the scope of the cookie to secure channels.

Set this flag only if the user agent is able to re-emit cookies over HTTPS on its next hop. For example, to re-emit a cookie with the `secure` flag, the user agent must be connected to its next hop by HTTPS.

Default: `false`

`"redirectionMarker"`: *configuration expression<object>, optional*

A redirect marker for the CDSSO flow. If the marker is present in the CDSSO flow, the request isn't redirected for authentication.

This feature is on by default to prevent redirect loops when the session cookie isn't present in the CDSSO flow. The cookie can be absent from the flow if it doesn't include IG's domain.

```
"redirectionMarker": {
  "enabled": configuration expression<boolean>,
  "name": configuration expression<string>
}
```

`"enabled"`: *configuration expression<boolean>, optional*

- `true`: When the session is empty or invalid, IG checks the request `goto` query parameter for the presence of the redirection marker:
  - If the redirection marker is present, IG fails the request.
  - If the redirection marker isn't present, IG redirects the user agent for login.
- false: IG never checks the request `goto` query parameter for the presence of a redirection marker.

Default: `true`

`"name"`: *configuration expression<string>, optional*

The name of the redirection marker query parameter to use when `enabled` is `true`.

Default: `_ig`

`"defaultLogoutLandingPage"`: *configuration expression<url>, optional*

The URL to which a request is redirected if `logoutExpression` is evaluated as true.

If this property is not an absolute URL, the request is redirected to the IG domain name.

This parameter is effective only when `logoutExpression` is specified.

Default: None, processing continues.

`"logoutExpression"`: *runtime expression<boolean>, optional*

A flag to indicate whether a request initiates logout processing before reaching the protected application.

- `false`: The request does not initiate logout processing:
  - If a valid AM session is found, the request is forwarded to the protected application.
  - If a valid AM session is not found, the request triggers login.

- `true`: The request initiates logout processing:
  - If a valid AM session is found, the session is revoked and the request is forwarded as follows:
    - If `defaultLogoutLandingPage` is defined, the request is forwarded to the specified logout page.
    - If `defaultLogoutLandingPage` is not defined, the request is forwarded to the protected application without any other validation.
  - If a valid session is not found, the request is forwarded to the protected application without any other validation.

> **Important**
>
> To prevent unwanted access to the protected application, use `logoutExpression` with extreme caution as follows:
> - Define a `defaultLogoutLandingPage`.
> - If you don't define a `defaultLogoutLandingPage`, specify `logoutExpression` to resolve to `true` **only** for requests that target dedicated logout pages of the protected application.
>
> Consider the following examples when a `defaultLogoutLandingPage` is not configured:
> - This expression resolves to `true` **only** for requests with `/app/logout` in their path:
>
> ```
> "logoutExpression": ${startsWith(request.uri.rawPath, '/app/logout')}
> ```
>
> When a request matches the expression, the AM session is revoked and the request is forwarded to the `/app/logout` page.
> - This expression resolves to `true` for **all** requests that contain `logOff=true` in their query parameters:
>
> ```
> "logoutExpression": ${find(request.uri.query, 'logOff=true')}
> ```
>
> When a request matches the expression, the AM session is revoked and the request is forwarded to the protected application without any other validation. In this example, an attacker can bypass IG's security mechanisms by simply adding `?logOff=true` to a request.

Default: `${false}`

`"failureHandler"`: *Handler reference, optional*

Handler to treat the request if an error occurs during authentication.

If an error occurs during authentication, a CdSsoFailureContext is populated with details of the error and any associated `Throwable`. This is available to the failure handler so that it can respond appropriately.

Be aware that the failure handler does not itself play a role in user authentication. It is only invoked if there is a problem that prevents user authentication from taking place.

A number of circumstances may cause the failure handler to be invoked, including:

- The redirect endpoint is invalid.
- The redirect endpoint is invoked without a valid CDSSO token.
- The redirect endpoint is invoked inappropriately.

- An error was reported by AM during authentication.

If no failure handler is configured, the default failure handler is used.

See also Handlers.

Default: HTTP 200 OK. The response entity contains details of the error.

`"verificationSecretId"`: *configuration expression<secret-id>, required if IG can't discover and use the AM JWK set*

The secret ID for the secret to verify the signature of AM session tokens. `verificationSecretId` must point to a CryptoKey.

IG verifies the signature of AM session tokens as follows:

1. With `verificationSecretId`.
2. If that fails or if `verificationSecretId` isn't configured, by discovering and using the AM JWK set.
3. If that fails, the CrossDomainSingleSignOnFilter fails to load.

For information about how IG verifies signatures, refer to Validate the signature of signed tokens. For information about how secret stores resolve named secrets, refer to Secrets.

`"secretsProvider"`: *SecretsProvider reference, required when* `verificationSecretId` *is set*

The SecretsProvider to query for passwords and cryptographic keys.

Example

For an example that uses the CrossDomainSingleSignOnFilter, refer to Cross-domain single sign-on (CDSSO).

More information

org.forgerock.openig.openam.SingleSignOnFilter

CdSsoContext

CdSsoFailureContext

SsoTokenContext

CsrfFilter

Prevent Cross Site Request Forgery (CSRF) attacks when using cookie-based authentication, as follows:

- When a session is created or updated for a client, generate a CSRF token as a hash of the session cookie.
- Send the token in a response header to the client, and require the client to provide that header in subsequent requests.
- In subsequent requests, compare the provided token to the generated token.
- If the token is not provided or can't be validated, reject the request and return a valid CSRF token transparently in the response header.

Rogue websites that attempt CSRF attacks operate in a different website domain to the targeted website. Because of same-origin policy, rogue websites can't access a response from the targeted website, and cannot, therefore, access the CSRF token.

Usage

```
{
  "name": string,
  "type": "CsrfFilter",
  "config": {
    "cookieName": configuration expression<string>,
    "headerName": configuration expression<string>,
    "excludeSafeMethods": configuration expression<boolean>,
    "failureHandler": Handler reference
  }
}
```

Properties

"cookieName": *configuration expression<string>, required*

The name of the HTTP session cookie used to store the session ID. For example, use the following cookie names for the following processes:

- SSO with the SingleSignOnFilter: Use the name of the AM session cookie. For more information, refer to Find the AM session cookie name.
- CDSSO with the CrossDomainSingleSignOnFilter: Use the name configured in `authCookie.name`.
- OpenID Connect with the AuthorizationCodeOAuth2ClientFilter: Use the name of the IG HTTP session cookie (default, `IG_SESSIONID`). For information about the IG session cookie, refer to admin.json.
- SAML: Use the name of the IG HTTP session cookie (default, `IG_SESSIONID`). For information about the IG session cookie, refer to admin.json.

"headerName": *configuration expression<string>, optional*

The name of the header that carries the CSRF token. The same header is used to create and verify the token.

Default: `X-CSRF-Token`

"excludeSafeMethods": *configuration expression<boolean>, optional*

Whether to exclude GET, HEAD, and OPTION methods from CSRF testing. In most cases, these methods are assumed as safe from CSRF.

Default: `true`

"failureHandler": *Handler reference, optional*

Handler to treat the request if the CSRF the token is not provided or can't be validated. Provide an inline handler declaration, or the name of a handler object defined in the heap.

Although IG returns the CSRF token transparently in the response header, this handler cannot access the CSRF token.

Default: Handler that generates `HTTP 403 Forbidden`.

Example

For an example of how to harden protection against CSRF attacks, see CSRF protection.

```
{
  "name": "CsrfFilter-1",
  "type": "CsrfFilter",
  "config": {
    "cookieName": "openig-jwt-session",
    "headerName": "X-CSRF-Token",
    "excludeSafeMethods": true
  }
}
```

More information

org.forgerock.openig.filter.CsrfFilterHeaplet

DataPreservationFilter

The DataPreservationFilter triggers POST data preservation when an unauthenticated client posts HTML form data to a protected resource.

When an authentication redirect is triggered, the filter stores the data in the HTTP session, and redirects the client for authentication. After authentication, the filter generates an empty self-submitting form POST to emulate the original POST. It then replays the stored data into the request before passing it along the chain.

The data can be any POST content, such as HTML form data or a file upload.

For more information, refer to POST data preservation.

Usage

```
{
  "type": "DataPreservationFilter",
  "config": {
    "noJavaScriptMessage": configuration expression<string>,
    "maxContentLength": configuration expression<positive integer>,
    "lifetime": configuration expression<duration>
  }
}
```

Properties

`"noJavaScriptMessage"`*: configuration expression<string>, optional*

JavaScript is used to replay the preserved data from the original POST that triggered the login redirect. This property configures a message to display if the user-agent does not support JavaScript.

Default: `Javascript is disabled in your browser, click on this button to replay the preserved original request`

`"maxContentLength"`: *configuration expression<positive integer>, optional*

The maximum number of bytes of POST data the filter can preserve. The size is taken from the Content-Length header.

Default: 4096

`"lifetime"`: *configuration expression<duration>, optional*

The maximum time that the filter can store POST data in an HTTP session.

The filter deletes stored POST data when the following events occur:

- The `lifetime` has expired.
- The POST data preservation process from an earlier request hasn't completed.
- A new request arrives that triggers a new POST data preservation process.

Stored POST data is also deleted when the session expires.

Default: 5 minutes

Example

For an example of use, refer to POST data preservation.

More information

org.forgerock.openig.filter.DataPreservationFilter

AuthRedirectContext

DateHeaderFilter

Inserts the server date in an HTTP `Date` header on the response, if the `Date` header is not present.

Usage

```
{
  "type": "DateHeaderFilter"
}
```

Properties

There are no configuration properties for this filter.

Example

The following example includes a DateHeaderFilter in a chain:

```
{
  "condition": "...",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
          ...
        },
        {
          "type": "DateHeaderFilter"
        }
      ],
      "handler": {
        "name": "StaticResponseHandler-1",
        ...
      }
    }
  }
}
```

More information

For information about Date format, see RFC 7231 - Date.

This filter is also available to support Financial-Grade API, for information, see Financial-grade API Security Profile 1.0 - Part 1: Baseline

org.forgerock.openig.filter.DateHeaderFilter

EncryptedPrivateKeyJwtClientAuthenticationFilter

Supports client authentication with the `private_key_jwt` client-assertion, using a signed and encrypted JWT.

Clients send a signed and encrypted JWT to the Authorization Server. IG builds, signs and encrypts the JWT, and prepares the request as in the following example:

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=...&
client_id=<clientregistration_id>&
client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer&
client_assertion=PHNhbWxwOl ... ZT
```

Use this filter with an endpoint `Handler` that requires authentication with the `private_key_jwt` client-assertion, using an encrypted JWT. For example, the `endpointHandler` handler in the OAuth2TokenExchangeFilter.

Usage

```
{
  "name": string,
  "type": "EncryptedPrivateKeyJwtClientAuthenticationFilter",
  "config": {
    "encryptionAlgorithm": configuration expression<enumeration>,
    "encryptionMethod": configuration expression<string>,
    "encryptionSecretId": configuration expression<secret-id>,
    "clientId": configuration expression<string>,
    "tokenEndpoint": configuration expression<url>,
    "secretsProvider": SecretsProvider reference,
    "signingSecretId": configuration expression<string>,
    "signingAlgorithm": configuration expression<string>,
    "jwtExpirationTimeout": configuration expression<duration>,
    "claims": map or configuration expression<map>
  }
}
```

Configuration

`"encryptionAlgorithm"`: *configuration expression<string>, required*

The algorithm name used for encryption and decryption. Use algorithm names from Java Security Standard Algorithm Names.

`"encryptionMethod"`: *configuration expression<string>, optional*

The algorithm method to use for encryption. Use algorithms from RFC 7518, section-5.1.

`"encryptionSecretId"`: *configuration expression<secret-id>, required*

The secret-id of the keys used to encrypt the JWT.

This secret ID must point to a CryptoKey.

`"clientId"`: *configuration expression<string>, required*

The `client_id` obtained when registering with the Authorization Server.

`"tokenEndpoint"`: *configuration expression<url>, required*

The URL to the Authorization Server's OAuth 2.0 token endpoint.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for passwords and cryptographic keys.

`"signingSecretId"`: *configuration expression<string>, required*

Reference to the keys used to sign the JWT.

This secret ID must point to a CryptoKey.

`"signingAlgorithm"`: *configuration expression<string>, optional*

The JSON Web Algorithm (JWA) used to sign the JWT, such as:

- `RS256`: RSA using SHA-256
- `ES256`: ECDSA with SHA-256 and NIST standard P-256 elliptic curve
- `ES384`: ECDSA with SHA-384 and NIST standard P-384 elliptic curve
- `ES512`: ECDSA with SHA-512 and NIST standard P-521 elliptic curve

Default: `RS256`

`"jwtExpirationTimeout"`: *configuration expression<duration>, optional*

The duration for which the JWT is valid.

Default: 1 minute

`"claims"`: *map or configuration expression<map>, optional*

A map of one or more data pairs with the format `Map<String, Object>`, where:

- The key is the name of a claim used in authentication
- The value is the value of the claim, or a configuration expression that evaluates to the value

The following formats are allowed:

```
{
  "args": {
    "string": "configuration expression<string>",
    ...
  }
}
```

```
{
  "args": "configuration expression<map>"
}
```

Default: Empty

EntityExtractFilter

Extracts regular expression patterns from a message entity, and stores their values in a `target` object. Use this object in password replay, to find a login path or extract a nonce.

If the message `type` is `REQUEST`, the pattern is extracted before the request is handled. If the message `type` is `RESPONSE`, the pattern is extracted out of the response body.

Each pattern can have an associated template, which is applied to its match result.

For information, see Patterns.

Usage

```
{
    "name": string,
    "type": "EntityExtractFilter",
    "config": {
        "messageType": configuration expression<enumeration>,
        "charset": configuration expression<string>,
        "target": lvalue-expression,
        "bindings": [
            {
                "key": configuration expression<string>,
                "pattern": pattern,
                "template": pattern
            }, ...
        ]
    }
}
```

Properties

`"messageType"`: *configuration expression<enumeration>, required*

The message type to extract patterns from.

Must be `REQUEST` or `RESPONSE`.

`"charset"`: *configuration expression<string>, optional*

Overrides the character set encoding specified in message.

Default: The message encoding is used.

`"target"`: *<lvalue-expression>, required*

Expression that yields the target object that contains the extraction results.

The bindings determine what type of object is stored in the target location.

The object stored in the target location is a Map<String, String>. You can then access its content with `${target.key}` or `${target['key']}`.

See also Expressions.

`"key"`: *configuration expression<string>, required*

Name of the element in the target object to contain an extraction result.

`"pattern"`: *pattern, required*

The regular expression pattern to find in the entity.

See also Patterns.

`"template"`: *pattern-template*, *optional*

The template to apply to the pattern, and store in the named target element.

Default: store the match result itself.

See also Patterns.

Examples

Extracts a nonce from the response, which is typically a login page, and sets its value in the attributes context to be used by the downstream filter posting the login form. The nonce value would be accessed using the following expression: `${attributes.extract.wpLoginToken}`.

The pattern finds all matches in the HTTP body of the form `wpLogintokenvalue="abc"`. Setting the template to `$1` assigns the value `abc` to `attributes.extract.wpLoginToken`:

```
{
    "name": "WikiNoncePageExtract",
    "type": "EntityExtractFilter",
    "config": {
        "messageType": "response",
        "target": "${attributes.extract}",
        "bindings": [
            {
                "key": "wpLoginToken",
                "pattern": "wpLoginToken\"\\s.*value=\"(.*)\"",
                "template": "$1"
            }
        ]
    }
}
```

The following example reads the response looking for the AM login page. When found, it sets `isLoginPage = true` to be used in a SwitchFilter to post the login credentials:

```
{
    "name": "FindLoginPage",
    "type": "EntityExtractFilter",
    "config": {
        "messageType": "response",
        "target": "${attributes.extract}",
        "bindings": [
            {
                "key": "isLoginPage",
                "pattern": "OpenAM\s\(Login\)",
                "template": "true"
            }
        ]
    }
}
```

More information

org.forgerock.openig.filter.EntityExtractFilter

FapiInteractionIdFilter

Tracks the interaction ID of requests, according to the Financial-grade API (FAPI) WG, as follows:

- If a FAPI header is provided in a client request, includes the interaction ID in the `x-fapi-interaction-id` property of the response header.
- If a FAPI header is not provided in the request, includes a new Universally Unique Identifier (UUID) in the `x-fapi-interaction-id` property of the response header.
- Adds the value of `x-fapi-interaction-id` to the log.

Usage

```
{
  "name": string,
  "type": "FapiInteractionIdFilter"
}
```

Properties

There are no configuration properties for this filter.

Example

The following example, based on Validate Certificate-Bound Access Tokens, adds a FapiInteractionIdFilter to the end of the chain:

```
{
  "name": "mtls",
  "condition": "${find(request.uri.path, '/mtls')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [ {
          "name": "OAuth2ResourceServerFilter-1",
           ...
        },
        {
          "type": "FapiInteractionIdFilter"
        }
      ],
      "handler": {
          "name": "StaticResponseHandler-1",
          ...
      }
    }
  }
}
```

More information

org.forgerock.openig.filter.finance.FapiInteractionIdFilter

Financial-grade API - Part 1: Read-Only API Security Profile

FragmentFilter

Tracks the fragment part of a URI when a request triggers a login redirect, as follows:

- Before authentication, the filter captures the URI fragment information and stores it in a cookie.
- After authentication, when the request is issued again to the original URI, the filter redirects the browser to the original URI, including any URI fragment.

The full fragment capture process is described in URI fragments in redirect.

The FragmentFilter doesn't handle multiple fragment captures in parallel. If a fragment capture is in progress while IG performs another login redirect, a second fragment capture process isn't triggered and the fragment is lost.

When a browser request loads a favicon, it can cause the fragment part of a URI to be lost. Prevent problems by serving static resources with a separate route. As an example, use the route in Serve static resources.

Use this filter with SingleSignOnFilter, CrossDomainSingleSignOnFilter, AuthorizationCodeOAuth2ClientFilter, and PolicyEnforcementFilter. This filter is not required for SAML because the final redirect is done with a DispatchHandler and a StaticResponseFilter.

Usage

```
{
  "name": string,
  "type": "FragmentFilter",
  "config": {
    "fragmentCaptureEndpoint": configuration expression<string>,
    "noJavaScriptMessage": configuration expression<string>,
    "cookie": object
  }
}
```

`"fragmentCaptureEndpoint"`: *configuration expression<string>, required*

The IG endpoint used to capture the fragment form data.

Configure the endpoint to match the condition of the route in which the filter is used.

`"noJavaScriptMessage"`: *configuration expression<string>, optional*

A message to display on the fragment form when JavaScript is not enabled.

Default: No message

`"cookie"`: *object, optional*

The configuration of the cookie used to store the fragment information.

```
{
  "name": configuration expression<string>,
  "domain": configuration expression<string>,
  "httpOnly": configuration expression<boolean>,
  "path": configuration expression<string>,
  "sameSite": configuration expression<enumeration>,
  "secure": configuration expression<boolean>,
  "maxAge": configuration expression<duration>
}
```

`"name"`: *configuration expression<string>, optional*

Cookie name.

Default: `ig-fragment-cookie`

`"domain"`: *configuration expression<string>, optional*

Domain to which the cookie applies.

Default: The fully qualified hostname of the IG host.

`"httpOnly"`: *configuration expression<boolean>, optional*

Flag to mitigate the risk of client-side scripts accessing protected cookies.

Default: `true`

`"path"`: *configuration expression<string>, optional*

Path to apply to the cookie.

Default: `/`

`"sameSite"`: *configuration expression<enumeration>, optional*

Options to manage the circumstances in which a cookie is sent to the server. Use one of the following values to reduce the risk of CSRF attacks:

- `STRICT`: Send the cookie only if the request was initiated from the cookie domain. Not case-sensitive.

  Use this value to reduce the risk of cross-site request forgery (CSRF) attacks.

- `LAX`: Send the cookie only with GET requests in a first-party context, where the URL in the address bar matches the cookie domain. Not case-sensitive.

  Use this value to reduce the risk of cross-site request forgery (CSRF) attacks.

- `NONE`: Send the cookie whenever a request is made to the cookie domain. Not case-sensitive.

  With this setting, consider setting `secure` to `true` to prevent browsers from rejecting the cookie. For more information, refer to SameSite cookies.

Default: `LAX`

`"secure"`: *configuration expression<boolean>, optional*

Flag to limit the scope of the cookie to secure channels.

Default: `false`

`"maxAge"`: *configuration expression<duration>, optional*

The maximum duration for which the FragmentFilter cookie can be valid.

When this `maxAge` is greater than the browser's maximum internal value, the browser value takes precedence.

Default: 1 hour

Example

For an example of how the FragmentFilter is used in an SSO flow, refer to URI fragments in redirect.

More information

org.forgerock.openig.filter.FragmentFilter

URI Fragment

RFC 3986: Fragment

AuthRedirectContext

FileAttributesFilter

Retrieves and exposes a record from a delimiter-separated file. Lookup of the record is performed using a specified `key`, whose `value` is derived from an expression. The resulting record is exposed in an object whose location is specified by the `target` expression. If a matching record cannot be found, then the resulting object is empty.

The retrieval of the record is performed lazily; it does not occur until the first attempt to access a value in the `target`. This defers the overhead of file operations and text processing until a value is first required. This also means that the value expression is not evaluated until the object is first accessed.

Usage

```
{
     "name": string,
     "type": "FileAttributesFilter",
     "config": {
         "file": configuration expression<string>,
         "charset": configuration expression<string>,
         "separator": configuration expression<enumeration>,
         "header": configuration expression<boolean>,
         "fields": [ configuration expression<string>, ... ],
         "target": lvalue-expression,
         "key": configuration expression<string>,
         "value": runtime expression<string>
     }
}
```

For an example, refer to Password replay from a file.

Properties

`"file"`: *configuration expression<string>, required*

The file containing the record to be read.

`"charset"`: *configuration expression<string>, optional*

The character set in which the file is encoded.

Default: `"UTF-8"`.

`"separator"`: *configuration expression<enumeration>, optional*

The separator character, which is one of the following:

`COLON`

Unix-style colon-separated values, with backslash as the escape character.

`COMMA`

Comma-separated values, with support for quoted literal strings.

`TAB`

Tab-separated values, with support for quoted literal strings.

Default: `COMMA`

`"header"`: *configuration expression<boolean>,optional*

A flag to treat the first row of the file as a header row.

When the first row of the file is treated as a header row, the data in that row is disregarded and cannot be returned by a lookup operation.

Default: `true`.

`"fields"`: *array of configuration expression<strings>, optional*

A list of keys in the order they appear in a record.

If `fields` is not set, the keys are assigned automatically by the column numbers of the file.

`"target"`: *<lvalue-expression>, required*

Expression that yields the target object to contain the record.

The target object is a `Map<String, String>`, where the fields are the keys. For example, if the target is `${attributes.file}` and the record has a `username` field and a `password` field mentioned in the fields list, Then you can access the user name as `${attributes.file.username}` and the password as `${attributes.file.password}`.

See also Expressions.

`"key"`: *configuration expression<string>, required*

The key used for the lookup operation.

`"value"`: *runtime expression<string>, required*

The value to be looked-up in the file.

See also Expressions.

More information

org.forgerock.openig.filter.FileAttributesFilter

ForwardedRequestFilter

Rebase the request URI to a computed scheme, host name, and port.

Use this filter to configure redirects when a request is forwarded by an upstream application such as a TLS offloader.

Usage

```
{
  "name": string,
  "type": "ForwardedRequestFilter",
  "config": {
    "scheme": runtime expression<string>,
    "host": runtime expression<string>,
    "port": runtime expression<number>
  }
}
```

Properties

At least one of `scheme`, `host`, or `port` must be configured.

`"scheme"`: *runtime expression<string>, optional*

The scheme to which the request is rebased, for example, `https`.

Default: Not rebased to a different scheme

`"host"`: *runtime expression<string>, optional*

The host to which the request is rebased.

Default: Not rebased to a different host

`"port"`: *runtime expression<number>, optional*

The port to which the request is rebased.

Default: Not rebased to a different port

Example

In the following configuration, IG runs behind an AWS load balancer, to perform a login page redirect to an authentication party, using the original URI requested by the client.

IG can access the URI used by the load balancer to reach IG, but can't access the original request URI.

The load balancer breaks the original request URI into the following headers, and adds them to the incoming request:

- `X-Forwarded-Proto`: Scheme
- `X-Forwarded-Port`: Port
- `Host`: Original host name, and possibly the port.

```
{
  "type": "ForwardedRequestFilter",
  "config": {
    "scheme": "${request.headers['X-Forwarded-Proto'][0]}",
    "host": "${split(request.headers['Host'][0], ':')[0]}",
    "port": "${integer(request.headers['X-Forwarded-Port'][0])}"
  }
}
```

More information

org.forgerock.openig.filter.ForwardedRequestFilter

GrantSwapJwtAssertionOAuth2ClientFilter

Transforms requests for OAuth 2.0 access tokens into secure JWT bearer grant type requests. Propagates transformed requests to Identity Cloud or AM to obtain an access token.

Use this filter with Identity Cloud or AM to increase the security of less-secure grant-type requests, such as Client credentials grant requests or Resource owner password credentials grant requests.

Caution

The GrantSwapJwtAssertionOAuth2ClientFilter obtains access tokens from the `/oauth2/access_token` endpoint. To prevent unwanted or malicious access to the endpoint, make sure only a well-defined set of clients can use this filter. Consider the following options to secure access to the GrantSwapJwtAssertionOAuth2ClientFilter:

- Deploy IG on a trusted network.
- Use mutual TLS (mTLS) and X.509 certificates for authentication between clients and IG. For more information, refer to OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens.
- Configure an AllowOnlyFilter in front of the GrantSwapJwtAssertionOAuth2ClientFilter to control access within a route.
- Define restrictive Route conditions to allow access only for expected grant-type requests. For example, define a route condition that requires a specific client ID, grant-type, or scope.
- Configure a ScriptableFilter in front of the GrantSwapJwtAssertionOAuth2ClientFilter to validate requests.

For an example that uses GrantSwapJwtAssertionOAuth2ClientFilter, refer to Secure the OAuth 2.0 access token endpoint.

Usage

```
{
  "name": string,
  "type": "GrantSwapJwtAssertionOAuth2ClientFilter",
  "config": {
    "clientId": configuration expression<string>,
    "scopes": [ runtime expression<string>, ... ] or ResourceAccess reference,
    "assertion": object,
    "secretsProvider": SecretsProvider reference,
    "signature": object,
    "encryption": object,
    "failureHandler": Handler reference
  }
}
```

Properties

`"clientId"`: *configuration expression<string>, optional*

The OAuth 2.0 client ID to use for authentication.

`"scopes"`: *array of runtime expression<strings> or ResourceAccess <reference>, required*

A list of one or more scopes required by the OAuth 2.0 access token. Provide the scopes as strings or through a ResourceAccess such as a RequestFormResourceAccess or ScriptableResourceAccess:

Array of runtime expression<strings>, required if a ResourceAccess isn't used

A string, array of strings, runtime expression<string>, or array of runtime expression<string> to represent one or more scopes.

RequestFormResourceAccess <reference>

A ResourceAccess that transfers scopes from the inbound request to a JWT bearer grant-type request.

In the following example request, the ResourceAccess extracts scopes from the request:

```
$ POST 'http://openig.example.com:8081/am/oauth2/access_token'
header 'Content-Type: application/x-www-form-urlencoded'
urlencoded form-data 'grant_type=client_credentials'
urlencoded form-data 'client_id=service-account'
urlencoded form-data 'scope=fr:idm:*'
```

Default: Empty

ScriptableResourceAccess <reference>

A script that evaluates each request dynamically and returns the scopes that the request needs to access the protected resource. The script must return a `Set<String>`.

For information about the properties of ScriptableResourceAccess, refer to Scripts.

```
{
  "name": string,
  "type": "ScriptableResourceAccess",
  "config": {
    "type": configuration expression<string>,
    "file": configuration expression<string>, // Use either "file"
    "source": [ string, ... ],               // or "source", but not both.
    "args": object,
    "clientHandler": Handler reference
  }
}
```

Default: Empty

`"assertion"`: *object*, *required*

The JWT claims. The GrantSwapJwtAssertionOAuth2ClientFilter checks that all mandatory fields are present and sets the JWT expiry. The filter doesn't check the fields in `otherClaims`.

```
{
  "assertion": {
    "issuer": runtime expression<string>,
    "subject": runtime expression<string>,
    "audience": runtime expression<string>,
    "expiryTime": runtime expression<duration>,
    "otherClaims": map<string, runtime expression<string>>
  }
}
```

`"issuer"`: *string*, *required*

The JWT `iss` claim. Can't be null.

`"subject"`: *string*, *required*

The JWT `sub` claim. Can't be null.

`"audience"`: *string*, *required*

The JWT `aud` claim. Can't be null.

`"expiryTime"`: *duration*, *required*

The JWT `exp` claim. Can't be `zero` or `unlimited`.

Default: 2 minutes

`"otherClaims"`: *map* *or map*, *optional*

A map of additional JWT claims with the format `Map<String, RuntimeExpression<String>>`, where:

- Key: Claim name
- Value: Claim value

Use the following format:

```
{
  "otherClaims": {
    "string": "runtime expression<string>",
    ...
  }
}
```

The filter doesn't check `otherClaims` in the JWT.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for passwords and cryptographic keys.

`"signature"`: *object, "signature" and/or "encryption" is required*

A JWT signature to validate the authenticity of claims and data.

```
{
  "signature": {
    "secretId": configuration expression<secret-id>,
    "includeKeyId": configuration expression<boolean>
  }
}
```

`"secretId"`: *configuration expression<secret-id>, required if* `signature` *is used*

The secret ID of the key to sign the JWT. The secret ID must point to a CryptoKey.

`"includeKeyId"`: *configuration expression<boolean>, optional*

A flag to include the ID of the signature key in the JWT header:

- `true`: Include the flag
- `false`: Don't include the flag

Default: `true`

`"encryption"`: *object, "signature" and/or "encryption" is required*

Configuration to encrypt the JWT.

This property take precedence over `GrantSwapJwtAssertionOAuth2ClientFilter.signature`.

```
{
  "encryption": {
    "secretId": secret-id,
    "algorithm": configuration expression<string>,
    "method": configuration expression<enumeration>
  }
}
```

`"secretId"`: *secret-id*, *optional*

The secret ID of the key used to encrypt the JWT. The value is mapped to key `aliases` in KeyStoreSecretStore.

This secret ID must point to a CryptoKey.

`"algorithm"`: *configuration expression*<*string*>, *required*

The algorithm used to encrypt the JWT.

For information about available algorithms, refer to RFC 7518: "alg" (Algorithm) Header Parameter Values for JWE.

`"method"`: *configuration expression*<*enumeration*>, *required*

The method used to encrypt the JWT.

For information about available methods, refer to RFC 7518: "enc" (Encryption Algorithm) Header Parameter Values for JWE.

`"failureHandler"`: *Handler* <*reference*>, *optional*

Handler to manage a failed request.

Provide an inline handler configuration object or the name of a handler object declared in the heap.

Default: `500 Internal Server Error`, the request stops being executed.

Example

For an example that uses GrantSwapJwtAssertionOAuth2ClientFilter, refer to Secure the OAuth 2.0 access token endpoint.

More information

org.forgerock.openig.filter.oauth2.client.GrantSwapJwtAssertionOAuth2ClientFilter

HeaderFilter

Removes headers from and adds headers to request and response messages. Headers are added to any existing headers in the message. To replace a header, remove the header and then add it again.

Usage

```
{
  "name": string,
  "type": "HeaderFilter",
  "config": {
    "messageType": configuration expression<enumeration>,
    "remove": [ configuration expression<string>, ... ],
    "add": {
      string: [ runtime expression<string>, ... ], ...
    }
  }
}
```

Properties

`"messageType": configuration expression<enumeration>, required

The type of message for which to filter headers. Must be either `"REQUEST"` or `"RESPONSE"`.

`"remove": array of configuration expression<strings>, optional`

The names of header fields to remove from the message.

`"add"`: object, optional

One or more headers to add to a request, with the format `name: [ value, … ]`, where:

- *name* is a string for a header name.
- *value* is a runtime expression that resolves to one or more header values.

Examples

Replace host header on an incoming request

The following example replaces the host header on the incoming request with the value `myhost.com`:

```
{
  "name": "ReplaceHostFilter",
  "type": "HeaderFilter",
  "config": {
    "messageType": "REQUEST",
    "remove": [ "host" ],
    "add": {
      "host": [ "myhost.com" ]
    }
  }
}
```

Add a header to a response

The following example adds a `Set-Cookie` header to the response:

```
{
  "name": "SetCookieFilter",
  "type": "HeaderFilter",
  "config": {
    "messageType": "RESPONSE",
    "add": {
      "Set-Cookie": [ "mysession=12345" ]
    }
  }
}
```

Add headers to a request

The following example adds the headers `custom1` and `custom2` to the request:

```
{
  "name": "SetCustomHeaders",
  "type": "HeaderFilter",
  "config": {
    "messageType": "REQUEST",
    "add": {
      "custom1": [ "12345", "6789" ],
      "custom2": [ "abcd" ]
    }
  }
}
```

Add a token value to a response

The following example adds the value of session's policy enforcement token to the `pef_sso_token` header in the response:

```
{
  "type": "HeaderFilter",
  "config": {
    "messageType": "RESPONSE",
    "add": {
      "pef_sso_token": ["${session.pef_token}"]
    }
  }
}
```

Add headers and logging results

The following example adds a message to the request and response as it passes through the Chain, and the `capture` on the ReverseProxyHandler logs the result. With IG and the sample application set up as described in the Quick install, access this route on http://ig.example.com:8080/home/chain.

```
{
  "condition": "${find(request.uri.path, '^/home/chain')}",
  "handler": {
    "type": "Chain",
    "comment": "Base configuration defines the capture decorator",
    "config": {
      "filters": [
        {
          "type": "HeaderFilter",
          "comment": "Add a header to all requests",
          "config": {
            "messageType": "REQUEST",
            "add": {
              "MyHeaderFilter_request": [
                "Added by HeaderFilter to request"
              ]
            }
          }
        },
        {
          "type": "HeaderFilter",
          "comment": "Add a header to all responses",
          "config": {
            "messageType": "RESPONSE",
            "add": {
              "MyHeaderFilter_response": [
                "Added by HeaderFilter to response"
              ]
            }
          }
        }
      ],
      "handler": {
        "type": "ReverseProxyHandler",
        "comment": "Log request, pass it to the sample app, log response",
        "capture": "all",
        "baseURI": "http://app.example.com:8081"
      }
    }
  }
}
```

The chain receives the request and context and processes it as follows:

- The first `HeaderFilter` adds a header to the incoming request.
- The second `HeaderFilter` manages responses not requests, so it simply passes the request and context to the handler.
- The `ReverseProxyHandler` captures (logs) the request.
- The `ReverseProxyHandler` forwards the transformed request to the protected application.
- The protected application passes a response to the `ReverseProxyHandler`.
- The `ReverseProxyHandler` captures (logs) the response.
- The second `HeaderFilter` adds a header added to the response.

- The first `HeaderFilter` is configured to manage requests, not responses, so it simply passes the response back to IG.

The following example lists some of the HTTP requests and responses captured as they flow through the chain. You can search the log files for `MyHeaderFilter_request` and `MyHeaderFilter_response`.

```
# Original request from user-agent
GET http://ig.example.com:8080/home/chain HTTP/1.1
Accept: /
Host: ig.example.com:8080

# Add a header to the request (inside IG) and direct it to the protected application
GET http://app.example.com:8081/home/chain HTTP/1.1
Accept: /
Host: ig.example.com:8080
MyHeaderFilter_request: Added by HeaderFilter to request

# Return the response to the user-agent
HTTP/1.1 200 OK
Content-Length: 1809
Content-Type: text/html; charset=ISO-8859-1

# Add a header to the response (inside IG)
HTTP/1.1 200 OK
Content-Length: 1809
MyHeaderFilter_response: Added by HeaderFilter to response
```

More information

org.forgerock.openig.filter.HeaderFilter

HttpBasicAuthenticationClientFilter

Authenticates clients according to the HTTP basic access authentication scheme.

HTTP basic access authentication is a simple challenge and response mechanism, where a server requests credentials from a client, and the client passes them to the server in an `Authorization` header. The credentials are base-64 encoded. To protect them, use SSL encryption for the connections between the server and client. For more information, refer to RFC 2617.

> **Tip**
>
> Compare the purpose of this filter with that of the following filters:
>
> - ClientCredentialsOAuth2ClientFilter, which authenticates clients by their OAuth 2.0 credentials to obtain an access token from an Authorization Server.
> - ClientSecretBasicAuthenticationFilter, which fulfils the same role of transforming OAuth 2.0 credentials to an `Authorization` header, but is more strict for OAuth 2.0 requirements.

Use HttpBasicAuthenticationClientFilter in a service-to-service context, where services need to access resources protected by HTTP basic access authentication.

Usage

```
{
  "name": string,
  "type": "HttpBasicAuthenticationClientFilter",
  "config": {
    "username": configuration expression<string>,
    "passwordSecretId": configuration expression<secret-id>,
    "secretsProvider": SecretsProvider reference,
    "urlEncodeCredentials": configuration expression<boolean>
  }
}
```

Properties

`"username"`: *configuration expression<string>, required*

The username of the client to authenticate.

`"passwordSecretId"`: *configuration expression<string>, required*

The secret ID required to obtain the client password.

This secret ID must point to a GenericSecret.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for the `passwordSecretId`.

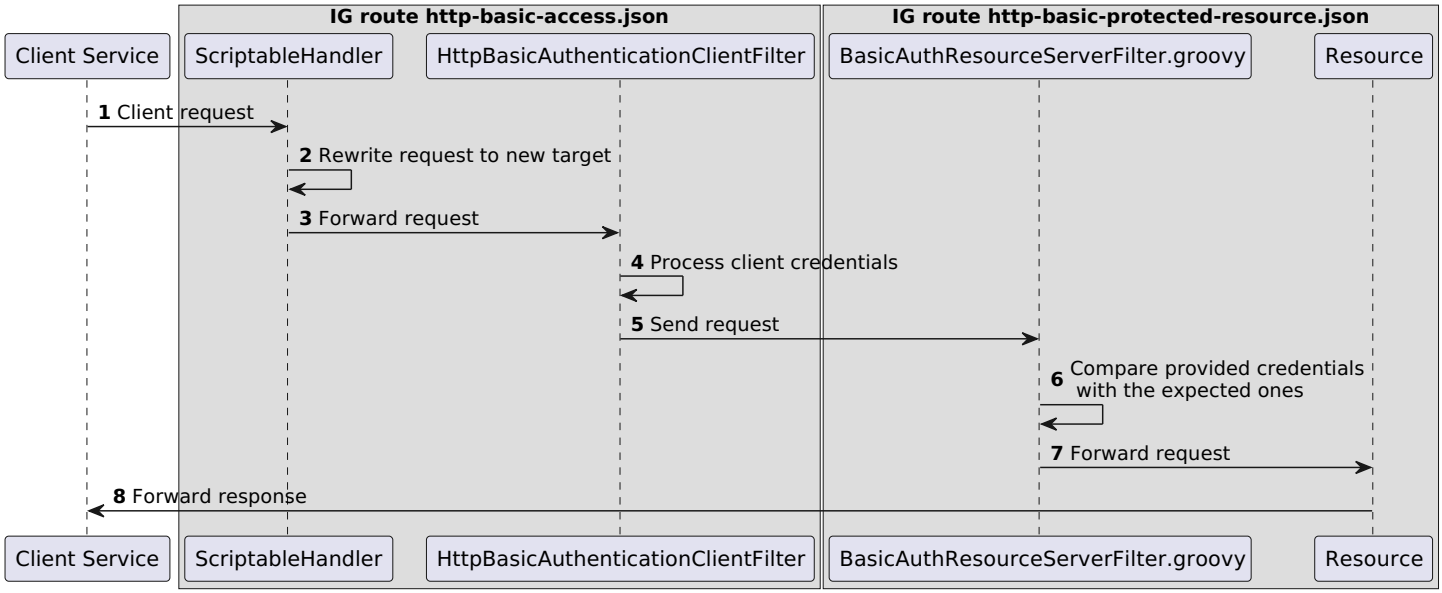
`"urlEncodeCredentials"`: *configuration expression<boolean>, optional*

Set to `true` to URL-encoded credentials before base64-encoding them.

Default: `false`

Example

The following example shows the flow of information when a client service accesses a resource protected by HTTP basic access authentication:



Set Up the Example

1. Add the following script to the IG configuration:

Linux

```
$HOME/.openig/scripts/groovy/BasicAuthResourceServerFilter.groovy
```

Windows

```
%appdata%\OpenIG\scripts\groovy\BasicAuthResourceServerFilter.groovy
```

```
/*
 * This script is a simple implementation of HTTP basic access authentication on
 * server side.
 * It expects the following arguments:
 *  - realm: the realm to display when the user agent prompts for
 *    username and password if none were provided.
 *  - username: the expected username
 *  - passwordSecretId: the secretId to find the password
 *  - secretsProvider: the SecretsProvider to query for the password
*/
import static org.forgerock.util.promise.Promises.newResultPromise;

import java.nio.charset.Charset;
import org.forgerock.util.encode.Base64;
import org.forgerock.secrets.Purpose;
import org.forgerock.secrets.GenericSecret;

String authorizationHeader = request.getHeaders().getFirst("Authorization");
if (authorizationHeader == null) {
    // No credentials provided, return 401 Unauthorized
    Response response = new Response(Status.UNAUTHORIZED);
    response.getHeaders().put("WWW-Authenticate", "Basic realm=\"" + realm + "\"");
    return newResultPromise(response);
}

return secretsProvider.getNamed(Purpose.PASSWORD, passwordSecretId)
        .thenAsync(password -> {
            // Build basic authentication string -> username:password
            StringBuilder basicAuthString = new StringBuilder(username).append(":");
            password.revealAsUtf8{ p -> basicAuthString.append(new String(p).trim()) };
            String expectedAuthorization = "Basic " +
Base64.encode(basicAuthString.toString().getBytes(Charset.defaultCharset()));
            // Incorrect credentials provided, return 403 forbidden
            if (!expectedAuthorization.equals(authorizationHeader)) {
                return newResultPromise(new Response(Status.FORBIDDEN));
            }
            // Correct credentials provided, continue.
            return next.handle(context, request);
        },
                noSuchSecretException -> { throw new RuntimeException(noSuchSecretException); });
```

The script is a simple implementation of the HTTP basic access authentication scheme. For information about scripting filters and handlers, refer to Extend.

2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/http-basic-access.json
```

Windows

```
%appdata%\OpenIG\config\routes\http-basic-access.json
```

```
{
  "name": "http-basic-access",
  "baseURI": "http://ig.example.com:8080",
  "condition" : "${find(request.uri.path, '^/http-basic-access')}",
  "heap": [
    {
      "name": "httpBasicAuthEnabledClientHandler",
      "type": "Chain",
      "capture": "all",
      "config": {
        "filters": [
          {
            "type": "HttpBasicAuthenticationClientFilter",
            "config": {
              "username": "myclient",
              "passwordSecretId": "password.secret.id",
              "secretsProvider": {
                "type": "Base64EncodedSecretStore",
                "config": {
                  "secrets": {
                    "password.secret.id": "cGFzc3dvcmQ="
                  }
                }
              }
            }
          }
        ],
        "handler": "ForgeRockClientHandler"
      }
    }
  ],
  "handler": {
    "type": "ScriptableHandler",
    "config": {
      "type": "application/x-groovy",
      "clientHandler": "httpBasicAuthEnabledClientHandler",
      "source": [
        "request.uri.path = '/http-basic-protected-resource'",
        "return http.send(context, request);"
      ]
    }
  }
}
```

Note the following features of the route:

- The route matches requests to `/http-basic-access`.
- The ScriptableHandler rewrites the request to target it to `/http-basic-protected-resource`, and then calls the HTTP client, that has been redefined to use the httpBasicAuthEnabledClientHandler.

- The httpBasicAuthEnabledClientHandler calls the HttpBasicAuthenticationClientFilter to authenticate the client, using the client's credentials.

3. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/http-basic-protected-resource.json
```

Windows

```
%appdata%\OpenIG\config\routes\http-basic-protected-resource.json
```

```
{
  "heap": [
    {
      "name": "mySecretsProvider",
      "type": "Base64EncodedSecretStore",
      "config": {
        "secrets": {
          "password.secret.id": "cGFzc3dvcmQ="
        }
      }
    }
  ],
  "name": "http-basic-protected-resource",
  "condition": "${find(request.uri.path, '^/http-basic-protected-resource')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "HttpBasicAuthResourceServerFilter",
          "type": "ScriptableFilter",
          "config": {
            "type": "application/x-groovy",
            "file": "BasicAuthResourceServerFilter.groovy",
            "args": {
              "realm": "IG Protected Area",
              "username": "myclient",
              "passwordSecretId": "password.secret.id",
              "secretsProvider": "${heap['mySecretsProvider']}"

            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><h2>Access Granted</h2></body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/http-basic-protected-resource`.
- The ScriptableFilter provides a script to implement a simple HTTP basic access authentication scheme, that compares the provided credentials with the expected credentials.
- When the client is authenticated, the StaticResponseHandler returns a message that access is granted.

4. Access the route on http://ig.example.com:8080/http-basic-access.

Because the expected credentials were provided in the request, a message shows that access is granted.

HttpBasicAuthFilter

Authenticate clients by providing the client credentials as a basic authorization header in the request. The credentials are base64-encoded.

This filter performs HTTP basic access authentication, described in RFC 2617.

Use this filter primarily for password replay scenarios, where the password is stored externally in clear text.

If challenged for authentication via a `401 Unauthorized` status code by the server, this filter retries the request with credentials attached. After an HTTP authentication challenge is issued from the remote server, all subsequent requests to that remote server that pass through the filter include the user credentials.

If authentication fails (including the case where no credentials are yielded from expressions), then processing is diverted to the specified authentication failure handler.

Usage

```
{
    "name": string,
    "type": "HttpBasicAuthFilter",
    "config": {
        "username": runtime expression<string>,
        "password": runtime expression<string>,
        "failureHandler": Handler reference,
        "cacheHeader": configuration expression<boolean>
    }
}
```

Properties

`"username"`: *runtime expression<string>, required*

The username to supply during authentication.

See also Expressions.

`"password"`: *runtime expression<string>, required*

The password to supply during authentication.

See also Expressions.

`"failureHandler"`: *Handler reference, required*

Dispatch to this Handler if authentication fails.

Provide either the name of a Handler object defined in the heap or an inline Handler configuration object.

See also Handlers.

`"cacheHeader"`: *configuration expression<boolean>,optional*

Whether or not to cache credentials in the session after the first successful authentication, and then replay those credentials for subsequent authentications in the same session.

With `"cacheHeader": false`, the filter generates the header for each request. This is useful, for example, when users change their passwords during a browser session.

Default: `true`

Example

```
{
    "name": "MyAuthenticator",
    "type": "HttpBasicAuthFilter",
    "config": {
        "username": "demo",
        "password": "password",
        "failureHandler": "AuthFailureHandler",
        "cacheHeader": false
    }
}
```

More information

org.forgerock.openig.filter.HttpBasicAuthFilter

IdTokenValidationFilter

Validates an ID token by checking the standard claims, `aud`, `exp`, and `iat`. If specified in the configuration, this filter also checks the ID token issuer and signature.

This filter passes data into the context as follows:

- If the JWT is validated, the request continues down the chain. The data is provided in the JwtValidationContext.
- If the JWT is not validated, data is provided in the JwtValidationErrorContext.

  If a failure handler is configured, the request passes to the failure handler. Otherwise, an HTTP 403 Forbidden is returned.

The `iat` claim is required, and the `iat` minus the `skewAllowance` must be before the current time on the IG clock. For information, see OpenID Connect Core 1.0 incorporating errata set 1.

Usage

```
{
   "name": string,
   "type": "IdTokenValidationFilter",
   "config": {
     "idToken": runtime expression<string>,
     "audience": configuration expression<string>,
     "issuer": configuration expression<string>,
     "skewAllowance": configuration expression<duration>,
     "verificationSecretId": configuration expression<secret-id>,
     "secretsProvider": SecretsProvider reference,
     "customizer": JwtValidatorCustomizer reference,
     "failureHandler": Handler reference
   }
}
```

Properties

`"idToken"`: *runtime expression<string>, required*

The ID token as an expression representing the JWT or signed JWT in the request. Cannot be null.

`"audience"`: *configuration expression<string>, required*

One `aud` claim to check on the JWT. Cannot be null.

`"issuer"`: *configuration expression<string>, optional*

One `iss` claim to check on the JWT. Can be null.

`"skewAllowance"`: *configuration expression<duration>, optional*

The duration to add to the validity period of a JWT to allow for clock skew between different servers.

A `skewAllowance` of 2 minutes affects the validity period as follows:

- A JWT with an `iat` of 12:00 is valid from 11:58 on the IG clock.
- A JWT with an `exp` 13:00 is expired after 13:02 on the IG clock.

Default: To support a zero-trust policy, the skew allowance is by default `zero`.

`"verificationSecretId"`: *configuration expression<secret-id>, required to verify the signature of signed tokens*

The secret ID for the secret to verify the signature of signed tokens.

This secret ID must point to a CryptoKey.

If configured, the token must be signed. If not configured, IG does not verify the signature.

For information about how signatures are validated, refer to Validate the signature of signed tokens. For information about how each type of secret store resolves named secrets, refer to Secrets.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for passwords and cryptographic keys.

`"customizer"`: *JwtValidatorCustomizer reference, optional*

A set of validation constraints for JWT claims and sub-claims. These constraints are in addition to internally-defined constraints, such as `aud`, `iss`, `exp`, and `iat`. If a claim isn't validated against a constraint, the JWT isn't validated.

The customizer doesn't override existing constraints. Defining a new constraint on an already constrained claim has an impact only if the new constraint is more restrictive.

JwtValidatorCustomizer provides a ScriptableJwtValidatorCustomizer to enrich a `builder` object by using its methods. Get more information about the following items:

- The `builder` object, at Available Objects.
- Transformer methods, to enrich the builder object, at org.forgerock.openig.util.JsonValues.
- Constraints, at org.forgerock.openig.tools.jwt.validation.Constraints.
- Other properties for ScriptableJwtValidatorCustomizer, at Scripts.

The following examples provide checks:

Check that the value of the claim `greaterThan5` *is greater than 5*

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "builder.claim('/greaterThan5', JsonValue::asInteger, isGreaterThan(5))"
    ]
  }
}
```

Check that the value of the claim `sub` *is* `george`

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "builder.claim('subname', JsonValue::asString, isEqualTo('george'))"
    ]
  }
}
```

Check that the value of the custom sub-claim is `ForgeRock`

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "builder.claim('customclaim/subclaim', JsonValue::asString, isEqualTo('ForgeRock'));"
    ]
  }
}
```

Check the value of multiple claims

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "builder.claim('aud', listOf(JsonValue::asString), contains('My App'))",
      "       .claim('iat', instant(), isInThePast())",
      "       .claim('exp', instant(), isInTheFuture());",
      "builder.claim('iss', JsonValue::asString, isEqualTo('ForgeRock AM'));"
    ]
  }
}
```

Check that the value of `val1` *is greater than* `val2`

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [ "builder.claim('/val1', JsonValue::asInteger, isGreaterThan(claim('/
val2').asInteger()))" ]
  }
}
```

Check that the value of `val1` *is greater than* `val2`*, when both are YYYY-MM-DD dates*

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "Function<JsonValue, java.time.LocalDate, Exception> asDate() {",
      "  return (jsonValue) -> java.time.LocalDate.parse(jsonValue.asString());",
      "}",
      "builder.claim('claim1', asDate(), isGreaterThan(claim('claim2').as(asDate())));"
    ]
  }
}
```

Check that the claim issuer matches the regex pattern

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [ "builder.claim('iss', JsonValue::asString, find(~/.*am\.example\.(com|org)/))" ]
  }
}
```

Default: Claims aren't validated

`"failureHandler"`: *Handler reference, optional*

Handler to treat the request on failure.

Provide an inline handler configuration object or the name of a handler object declared in the heap. See also Handlers.

Default: HTTP 403 Forbidden, the request stops being executed.

Example

Validate an id_token

1. Set up AM:
    1. Set up AM as described in Validate access tokens through the introspection endpoint.
    2. Select **Applications** > **OAuth 2.0** > **Clients**, and add the additional scope `openid` to `client-application`.
2. Set up IG:
    1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/idtokenvalidation.json
```

Windows

```
%appdata%\OpenIG\config\routes\idtokenvalidation.json
```

```
{
  "name": "idtokenvalidation",
  "condition": "${find(request.uri.path, '^/idtokenvalidation')}",
  "capture": "all",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "type": "IdTokenValidationFilter",
        "config": {
          "idToken": "<id_token_value>",
          "audience": "client-application",
          "issuer": "http://am.example.com:8088/openam/oauth2",
          "failureHandler": {
            "type": "ScriptableHandler",
            "config": {
              "type": "application/x-groovy",
              "source": [
                "def response = new Response(Status.FORBIDDEN)",
                "response.headers['Content-Type'] = 'text/html; charset=utf-8'",
                "def errors = contexts.jwtValidationError.violations.collect{it.description}",
                "def display = \"<html>Can't validate id_token:<br> ${contexts.jwtValidationError.jwt}
\"",
                "display <<=\"<br><br>For the following errors:<br> ${errors.join(\"<br>\")}</html>\"",
                "response.entity=display as String",
                "return response"
              ]
            }
          },
          "verificationSecretId": "verify",
          "secretsProvider": {
            "type": "JwkSetSecretStore",
            "config": {
              "jwkUrl": "http://am.example.com:8088/openam/oauth2/connect/jwk_uri"
            }
          }
        }
      }],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body>Validated id_token:<br> ${contexts.jwtValidation.value}</body></html>"
        }
      }
    }
  }
}
```

Notice the following features of the route:

- The route matches requests to `/idtokenvalidation`.
- A SecretsProvider declares a JwkSetSecretStore to validate secrets for signed JWTs, which specifies the URL to a JWK set on AM that contains the signing keys.

- The property `verificationSecretId` is configured with a value. If this property is not configured, the filter does not verify the signature of tokens.
- The JwkSetSecretStore specifies the URL to a JWK set on AM, that contains verification keys identified by a `kid`. The signature of the token is verified as follows:
  - If the value of a `kid` in the JWK set matches a `kid` in the the signed JWT, the JwkSetSecretStore verifies the signature.
  - If the JWT doesn't have a `kid`, or if the JWK set doesn't contain a key with the same value, the JwkSetSecretStore looks for valid secrets with the same purpose as the value of `verificationSecretId`.
- If the filter validates the token, the StaticResponseHandler displays the token value from the context `${contexts.jwtValidation.value}`. Otherwise, the ScriptableHandler displays the token value and a list of violations from the context `${contexts.jwtValidationError.violations}`

3. Test the setup:

   1. In a terminal window, use a `curl` command similar to the following to retrieve an id_token:

      ```
      $ curl -s \
      --user "client-application:password" \
      --data "grant_type=password&username=demo&password=Ch4ng31t&scope=openid" \
      http://am.example.com:8088/openam/oauth2/access_token

      {
       "access_token":"...",
       "scope":"openid",
       "id_token":"...",
       "token_type":"Bearer",
       "expires_in":3599
      }
      ```

      - In the route, replace `<id_token_value>` with the value of the `id_token` returned in the previous step.

   2. Access the route on http://ig.example.com:8080/idtokenvalidation.

      The validated token is displayed.

      - In the route, invalidate the token by changing the value of the audience or issuer, and then access the route again.

        The value of the token, and the reasons that the token is invalid, are displayed.

More information

org.forgerock.openig.filter.oauth2.client.IdTokenValidationFilterHeaplet

org.forgerock.openig.filter.jwt.JwtValidationContext

org.forgerock.openig.filter.jwt.JwtValidationErrorContext

OpenID Connect Core 1.0 incorporating errata set 1

JwtBuilderFilter

Collects data at runtime, packs it in a JSON Web Token (JWT), and places the resulting JWT into the JwtBuilderContext.

Configure JwtBuilderFilter to create a signed JWT, a signed then encrypted JWT, or an encrypted JTW:

- Sign the JWT so that an application can validate the authenticity of the claims/data. The JWT can be signed with a shared secret or private key, and verified with a shared secret or corresponding public key.
- Encrypt the JWT to reduce the risk of a data breach.

For a flexible way to pass identity or other runtime information to the protected application, use this filter with a HeaderFilter.

To enable downstream filters and handlers to verify signed and/or encrypted JWTs built by this filter, use this filter with a JwkSetHandler.

Usage

```
{
  "name": string,
  "type": "JwtBuilderFilter",
  "config": {
    "template": map or runtime expression<map>,
    "secretsProvider": SecretsProvider reference,
    "signature": object,
    "encryption": object
  }
}
```

Properties

`"template"`: *map or runtime expression<map>, required*

A map of one or more data pairs with the format `Map<String, Object>`, where:

- The key is the name of a data field
- The value is a data object, or a runtime expression that evaluates to a data object

The following formats are allowed:

```
{
  "template": {
    "string": "runtime expression<object>",
    ...
  }
}
```

```
{
  "template": "runtime expression<map>"
}
```

In the following example, the property is a map whose values are runtime expressions that evaluate to objects in the context:

```
{
  "template": {
    "name": "${contexts.userProfile.commonName}",
    "email": "${contexts.userProfile.rawInfo.mail[0]}",
    "address": "${contexts.userProfile.rawInfo.postalAddress[0]}",
    "phone": "${contexts.userProfile.rawInfo.telephoneNumber[0]}"
  }
}
```

In the following example, the property is a runtime expression that evaluates to a map with the format `Map<String, Object>`:

```
{
  "template": "${contexts.attributes}"
}
```

Use the `now` dynamic binding to dynamically set the value of an attribute that represents time. For example, set the value of attributes to a defined time after the expressions are evaluated, as follows:

```
{
  "name": "JwtBuilderFilter-1",
  "type": "JwtBuilderFilter",
    "config": {
    "template": {
      "iat": "${now.epochSeconds}",
      "nbf": "${now.plusSeconds(10).epochSeconds}",
      "exp": "${now.plusSeconds(20).epochSeconds}"
    },
    "secretsProvider": "FileSystemSecretStore-1",
      "signature": {
        "secretId": "id.key.for.signing.jwt",
        "algorithm": "RS512"
    }
  }
}
```

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for JWT signing or encryption keys.

`"signature"`: *object, "signature" and/or "encryption" is required*

A JWT signature to allow the authenticity of the claims/data to be validated. A signed JWT can be encrypted.

`JwtBuilderFilter.encryption` takes precedence over this property.

```
{
  "signature": {
    "secretId": configuration expression<secret-id>,
    "includeKeyId": configuration expression<secret-id>,
    "algorithm": configuration expression<string>,
    "encryption": object
  }
}
```

`"secretId"`: *configuration expression<secret-id>, required if* `signature` *is used*

The secret ID of the key to sign the JWT.

This secret ID must point to a CryptoKey.

`"includeKeyId"`: *configuration expression<boolean>, optional*

When `true`, include the ID of the signature key in the JWT header.

Default: `true`

`"algorithm"`: *configuration expression<string>, optional*

The algorithm to sign the JWT.

The following algorithms are supported but not necessarily tested in IG:

- Algorithms described in RFC 7518: Cryptographic Algorithms for Digital Signatures and MACs.

  For RSASSA-PSS, you must install Bouncy Castle. For information, refer to The Legion of the Bouncy Castle.

- From IG 6.1, `Ed25519` described in CFRG Elliptic Curve Diffie-Hellman (ECDH) and Signatures in JSON Object Signing and Encryption (JOSE).

  Default: RS256

`"encryption"`: *object, optional*

Configuration to encrypt the JWT signature.

```
{
  "encryption": {
    "secretId": configuration expression<secret-id>,
    "algorithm": configuration expression<string>,
    "method": configuration expression<string>
  }
}
```

`"secretId"`: *configuration expression<secret-id>, optional*

The secret ID of the key used to encrypt the JWT signature. The value is mapped to key `aliases` in KeyStoreSecretStore.

This secret ID must point to a CryptoKey.

`"algorithm"`: *configuration expression<string>, required*

The algorithm used to encrypt the JWT signature.

For information about available algorithms, refer to RFC 7518: "alg" (Algorithm) Header Parameter Values for JWE.

`"method"`: *configuration expression<string>, required*

The method used to encrypt the JWT signature.

For information about available methods, refer to RFC 7518: "enc" (Encryption Algorithm) Header Parameter Values for JWE.

`"encryption"`: *object, "signature" and/or "encryption" is required*

Configuration to encrypt the JWT.

This property take precedence over `JwtBuilderFilter.signature`.

```
{
  "encryption": {
    "secretId": secret-id,
    "algorithm": configuration expression<string>,
    "method": configuration expression<enumeration>
  }
}
```

`"secretId"`: *secret-id, optional*

The secret ID of the key used to encrypt the JWT. The value is mapped to key `aliases` in KeyStoreSecretStore.

This secret ID must point to a CryptoKey.

`"algorithm"`: *configuration expression<string>, required*

The algorithm used to encrypt the JWT.

For information about available algorithms, refer to RFC 7518: "alg" (Algorithm) Header Parameter Values for JWE.

`"method"`: *configuration expression<enumeration>, required*

The method used to encrypt the JWT.

For information about available methods, refer to RFC 7518: "enc" (Encryption Algorithm) Header Parameter Values for JWE.

Examples

For examples, refer to Passing data along the chain

More information

org.forgerock.openig.filter.JwtBuilderFilter

org.forgerock.openig.filter.JwtBuilderContext

JwtValidationFilter

Validates an unsigned, signed, encrypted, or signed and encrypted JWT. The order of signing and encryption isn't important; a JWT can be signed and then encrypted, or encrypted and then signed.

If the JWT is validated, the request continues down the chain and data is provided in the JwtValidationContext.

If the JWT isn't validated, data is provided in the JwtValidationErrorContext. If a failure handler is configured, the request passes to the failure handler. Otherwise, an HTTP 403 Forbidden is returned.

Usage

```
{
  "name": string,
  "type": "JwtValidationFilter",
  "config": {
    "jwt": runtime expression<string>,
    "verificationSecretId": configuration expression<secret-id>,
    "decryptionSecretId": configuration expression<secret-id>,
    "secretsProvider": SecretsProvider reference,
    "skewAllowance": configuration expression<duration>,
    "customizer": JwtValidatorCustomizer reference,
    "failureHandler": Handler reference
  }
}
```

Properties

`"jwt"`: *runtime expression<string>, required*

The value of the JWT in the request. Cannot be null.

`"verificationSecretId"`: *configuration expression<secret-id>, required to verify the signature of signed tokens*

The secret ID for the secret to verify the signature of signed tokens.

This secret ID must point to a CryptoKey.

If configured, the token must be signed. If not configured, IG does not verify the signature.

For information about how signatures are validated, refer to Validate the signature of signed tokens. For information about how each type of secret store resolves named secrets, refer to Secrets.

`"decryptionSecretId"`: *configuration expression<secret-id>, required if AM secures access tokens with encryption*

The secret ID for the secret to verify the encryption of tokens.

This secret ID must point to a CryptoKey.

If configured, the token must be encrypted. If not configured, IG doesn't verify the encryption.

For information about how each type of secret store resolves named secrets, see Secrets.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for passwords and cryptographic keys.

`"customizer"`: *JwtValidatorCustomizer reference, optional*

A set of validation constraints for JWT claims and sub-claims. These constraints are in addition to internally-defined constraints, such as `aud`, `iss`, `exp`, and `iat`. If a claim isn't validated against a constraint, the JWT isn't validated.

The customizer doesn't override existing constraints. Defining a new constraint on an already constrained claim has an impact only if the new constraint is more restrictive.

JwtValidatorCustomizer provides a ScriptableJwtValidatorCustomizer to enrich a `builder` object by using its methods. Get more information about the following items:

- The `builder` object, at Available Objects.
- Transformer methods, to enrich the builder object, at org.forgerock.openig.util.JsonValues.
- Constraints, at org.forgerock.openig.tools.jwt.validation.Constraints.
- Other properties for ScriptableJwtValidatorCustomizer, at Scripts.

The following examples provide checks:

Check that the value of the claim `greaterThan5` *is greater than 5*

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "builder.claim('/greaterThan5', JsonValue::asInteger, isGreaterThan(5))"
    ]
  }
}
```

Check that the value of the claim `sub` *is* `george`

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "builder.claim('subname', JsonValue::asString, isEqualTo('george'))"
    ]
  }
}
```

Check that the value of the custom sub-claim is `ForgeRock`

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "builder.claim('customclaim/subclaim', JsonValue::asString, isEqualTo('ForgeRock'));"
    ]
  }
}
```

Check the value of multiple claims

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "builder.claim('aud', listOf(JsonValue::asString), contains('My App'))",
      "       .claim('iat', instant(), isInThePast())",
      "       .claim('exp', instant(), isInTheFuture());",
      "builder.claim('iss', JsonValue::asString, isEqualTo('ForgeRock AM'));"
    ]
  }
}
```

Check that the value of `val1` *is greater than* `val2`

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [ "builder.claim('/val1', JsonValue::asInteger, isGreaterThan(claim('/
val2').asInteger()))" ]
  }
}
```

Check that the value of `val1` *is greater than* `val2`*, when both are YYYY-MM-DD dates*

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "Function<JsonValue, java.time.LocalDate, Exception> asDate() {",
      "  return (jsonValue) -> java.time.LocalDate.parse(jsonValue.asString());",
      "}",
      "builder.claim('claim1', asDate(), isGreaterThan(claim('claim2').as(asDate())));"
    ]
  }
}
```

Check that the claim issuer matches the regex pattern

```
"customizer": {
  "type": "ScriptableJwtValidatorCustomizer",
  "config": {
    "type": "application/x-groovy",
    "source": [ "builder.claim('iss', JsonValue::asString, find(~/.*am\.example\.(com|org)/))" ]
  }
}
```

Default: Claims aren't validated

`"skewAllowance"`: *configuration expression<duration>, optional*

The duration to add to the validity period of a JWT to allow for clock skew between different servers.

A `skewAllowance` of 2 minutes affects the validity period as follows:

- A JWT with an `iat` of 12:00 is valid from 11:58 on the IG clock.
- A JWT with an `exp` 13:00 is expired after 13:02 on the IG clock.

Default: To support a zero-trust policy, the skew allowance is by default `zero`.

`"failureHandler"`: *Handler reference, optional*

Handler to treat the request on failure.

Provide an inline handler configuration object or the name of a handler object declared in the heap. See also Handlers.

Default: HTTP 403 Forbidden, the request stops being executed.

Example

For an example of using JwtValidationFilter, refer to JWT validation.

More information

org.forgerock.openig.filter.jwt.JwtValidationFilter

org.forgerock.openig.filter.jwt.JwtValidationContext

org.forgerock.openig.filter.jwt.JwtValidationErrorContext

OpenID Connect Core 1.0 incorporating errata set 1

LocationHeaderFilter

For a response that generates a redirect to the proxied application, this filter rewrites the Location header on the response to redirect the user to IG.

Usage

```
{
    "name": string,
    "type": "LocationHeaderFilter",
    "config": {
        "baseURI": runtime expression<url>
    }
}
```

An alternative value for type is RedirectFilter.

Properties

`"baseURI"`: *runtime expression<url>,optional*

The base URI of the IG instance. This is used to rewrite the Location header on the response.

The result of the expression must be a string that represents a valid URI, but is not a real `java.net.URI` object. For example, it would be incorrect to use `${request.uri}`, which is not a String but a MutableUri.

Default: Redirect to the original URI specified in the request.

See also Expressions.

Example

In the following example, IG listens on `https://ig.example.com:443` and the application it protects listens on `http://app.example.com:8081`. The filter rewrites redirects that would normally take the user to locations under `http://app.example.com:8081` to go instead to locations under `https://ig.example.com:443`.

```
{
     "name": "LocationRewriter",
     "type": "LocationHeaderFilter",
     "config": {
         "baseURI": "https://ig.example.com:443/"
      }
}
```

More information

org.forgerock.openig.filter.LocationHeaderFilter

OAuth2ClientFilter

In IG 7.2, this filter was renamed to AuthorizationCodeOAuth2ClientFilter.

For backward compatibility, the name OAuth2ClientFilter can still be used in routes in this release. However, to prevent problems in future releases, update your configuration as soon as possible.

OAuth2ResourceServerFilter

Validates a request containing an OAuth 2.0 access token. The filter expects an OAuth 2.0 token from the HTTP Authorization header of the request, such as the following example header, where the OAuth 2.0 access token is `1fc…ec9`:

```
Authorization: Bearer 1fc...ec9
```

The filter performs the following tasks:

- Extracts the access token from the request header.
- Uses the configured access token resolver to resolve the access token against an Authorization Server, and validate the token claims.
- Checks that the token has the scopes required by the filter configuration.
- Injects the access token info into the OAuth2Context.

The following errors can occur during access token validation:

Error	Response from the filter to the user agent
Combination of the filter configuration and access token result in an invalid request to the Authorization Server.	`HTTP 400 Bad Request`
There is no access token in the request header.	`HTTP 401 Unauthorized WWW-Authenticate: Bearer realm="IG"`
The access token isn't valid, for example, because it has expired.	`HTTP 401 Unauthorized`
The access token doesn't have all of the scopes required in the OAuth2ResourceServerFilter configuration.	`HTTP 403 Forbidden`

Usage

```
{
  "name": string,
  "type": "OAuth2ResourceServerFilter",
  "config": {
    "accessTokenResolver": AccessTokenResolver reference,
    "cache": object,
    "executor": Executor service reference,
    "requireHttps": configuration expression<boolean>,
    "realm": configuration expression<string>,
    "scopes": [ runtime expression<string>, ... ] or ScriptableResourceAccess reference
  }
}
```

An alternative value for type is OAuth2RSFilter.

Properties

`"accessTokenResolver"`: *AccessTokenResolver reference, required*

Resolves an access token against an Authorization Server. Configure one of the following access token resolvers:

- TokenIntrospectionAccessTokenResolver
- StatelessAccessTokenResolver
- ConfirmationKeyVerifierAccessTokenResolver
- ScriptableAccessTokenResolver

To decorate an AccessTokenResolver, add the decoration at the `accessTokenResolver` level. The following example uses the default timer decorator to record the time that a TokenIntrospectionAccessTokenResolver takes to process a request:

```
{
  "accessTokenResolver": {
    "type": "TokenIntrospectionAccessTokenResolver",
    "config": {
      ...
    },
  "timer": true
  }
}
```

`"cache"`: *object, optional*

Configuration of caching for OAuth 2.0 access tokens. By default, access tokens are not cached. For an alternative way of caching of OAuth 2.0 access tokens, configure CacheAccessTokenResolver.

When an access token is cached, IG can reuse the token information without repeatedly asking the Authorization Server to verify the access token. When caching is disabled, IG must ask the Authorization Server to verify the access token for each request.

When an access_token is revoked on AM, the CacheAccessTokenResolver can delete the token from the cache when both of the following conditions are true:

- The `notification` property of AmService is enabled.
- The delegate AccessTokenResolver provides the token metadata required to update the cache.

When a refresh_token is revoked on AM, all associated access tokens are automatically and immediately revoked.

```
"cache": {
  "enabled": configuration expression<boolean>,
  "defaultTimeout": configuration expression<duration>,
  "maxTimeout": configuration expression<duration>,
  "amService": AmService reference,
  "onNotificationDisconnection": configuration expression<enumeration>
}
```

`enabled`: *configuration expression<boolean>, optional*

Enable or disable caching.

Default: `false`

`defaultTimeout`: *configuration expression<duration>, optional*

The duration for which to cache an OAuth 2.0 access token if it doesn't provide a valid expiry value.

If an access token provides an expiry value that falls *before* the current time plus the `maxTimeout`, IG uses the token expiry value.

The following example caches access tokens for these times:

- One hour, if the access token doesn't provide a valid expiry value.
- The duration specified by the token expiry value, when the token expiry value is shorter than one day.
- One day, when the token expiry value is longer than one day.

```
"cache": {
  "enabled": true,
  "defaultTimeout": "1 hour",
  "maxTimeout": "1 day"
}
```

Default: `1 minute`

`maxTimeout`: *configuration expression<duration>, optional*

The maximum duration for which to cache OAuth 2.0 access tokens.

If an access token provides an expiry value that falls *after* the current time plus the `maxTimeout`, IG uses the `maxTimeout`.

The duration cannot be `zero` or `unlimited`.

`"amService"`: *AmService reference, optional*

The AmService to use for the WebSocket notification service. To evict revoked access tokens from the cache, enable the `notifications` property of AmService.

`onNotificationDisconnection`: *configuration expression<enumeration>, optional*

An `amService` must be configured for this property to have effect.

The strategy to manage the cache when the WebSocket notification service is disconnected, and IG receives no notifications for AM events. If the cache is not cleared it can become outdated, and IG can allow requests on revoked sessions or tokens.

Cached entries that expire naturally while the notification service is disconnected are removed from the cache.

Use one of the following values:

- `NEVER_CLEAR`
  - When the notification service is disconnected:
    - Continue to use the existing cache.
    - Deny access for requests that are not cached, but do not update the cache with these requests.
  - When the notification service is reconnected:
    - Continue to use the existing cache.
    - Query AM for incoming requests that are not found in the cache, and update the cache with these requests.
- `CLEAR_ON_DISCONNECT`
  - When the notification service is disconnected:
    - Clear the cache.
    - Deny access to all requests, but do not update the cache with these requests.
  - When the notification service is reconnected:
    - Query AM for all requests that are not found in the cache. (Because the cache was cleared, the cache is empty after reconnection.)
    - Update the cache with these requests.
- `CLEAR_ON_RECONNECT`
  - When the notification service is disconnected:
    - Continue to use the existing cache.
    - Deny access for requests that are not cached, but do not update the cache with these requests.
  - When the notification service is reconnected:
    - Query AM for all requests that are not found in the cache. (Because the cache was cleared, the cache is empty after reconnection.)
    - Update the cache with these requests.

Default: `CLEAR_ON_DISCONNECT`

`"executor"`: *Executor service reference, optional*

An executor service to schedule the execution of tasks, such as the eviction of entries in the access token cache.

Default: `ScheduledExecutorService`

See also ScheduledExecutorService.

`"requireHttps"`: *configuration expression<boolean>, optional*

Whether to require that original target URI of the request uses the HTTPS scheme.

If the received request doesn't use HTTPS, it is rejected.

Default: true.

`"realm"`: *configuration expression<string>, optional*

HTTP authentication realm to include in the WWW-Authenticate response header field when returning an HTTP 401 Unauthorized status to a user agent that need to authenticate.

Default: OpenIG

`"scopes"`: *array of runtime expression<strings> or ResourceAccess <reference>, required*

A list of one or more scopes required by the OAuth 2.0 access token. Provide the scopes as strings or through a ResourceAccess such as a ScriptableResourceAccess:

Array of runtime expression<strings>, required if a ResourceAccess isn't used

A string, array of strings, runtime expression<string>, or array of runtime expression<string> to represent one or more scopes.

ScriptableResourceAccess <reference>

A script that evaluates each request dynamically and returns the scopes that the request needs to access the protected resource. The script must return a `Set<String>`.

For information about the properties of ScriptableResourceAccess, refer to Scripts.

```
{
  "name": string,
  "type": "ScriptableResourceAccess",
  "config": {
    "type": configuration expression<string>,
    "file": configuration expression<string>, // Use either "file"
    "source": [ string, ... ],               // or "source", but not both.
    "args": object,
    "clientHandler": Handler reference
  }
}
```

Default: Empty

Examples

For examples using OAuth2ResourceServerFilter, see Act as an OAuth 2.0 resource server.

More information

org.forgerock.openig.filter.oauth2.OAuth2ResourceServerFilterHeaplet

org.forgerock.http.oauth2.OAuth2Context

org.forgerock.http.oauth2.AccessTokenInfo

OAuth2Context

ConfirmationKeyVerifierAccessTokenResolver

TokenIntrospectionAccessTokenResolver

StatelessAccessTokenResolver

ScriptableAccessTokenResolver

The OAuth 2.0 Authorization Framework

The OAuth 2.0 Authorization Framework: Bearer Token Usage

OAuth2TokenExchangeFilter

Identifies a client's access token or ID token (a *subject token*), and communicates with an authorization service, such as AM, to exchange it for a new token (an *issued token*):

- When the OAuth2TokenExchangeFilter succesfully exchanges a token, it injects the issued token and its scopes into the OAuth2TokenExchangeContext.
- When the OAuth2TokenExchangeFilter fails to exchange a token, it injects information about the failure into the OAuth2FailureContext, which is provided to the `failureHandler`.

The scopes for issued token can be *restricted* or *expanded* by the authorization services:

- Restricted when the token scopes are a subset of those available to the subject token.
- Expanded when they have scopes that are not included in the subject token.

Use this filter in the *impersonation* use case. For more information, refer to Token Exchange in AM's *OAuth 2.0 guide*.

Usage

```
{
  "name": string,
  "type": "OAuth2TokenExchangeFilter",
    "config": {
      "subjectToken": runtime expression<string>,
      "amService": AmService reference,
      "endpoint": configuration expression<url>,
      "subjectTokenType": configuration expression<string>,
      "requestedTokenType": configuration expression<string>,
      "scopes": [ runtime expression<string>, ... ] or ScriptableResourceAccess reference,
      "resource": configuration expression<url>,
      "audience": configuration expression<string>,
      "endpointHandler": Handler reference,
      "failureHandler": Handler reference
  }
}
```

Configuration

`"subjectToken"`: *runtime expression<string>, required*

The location of the subject token in the inbound request.

`"amService"`: *AmService reference, required if* `endpoint` *is not configured*

The AmService to use as the authorization service.

Configure either 'amService' or 'endpoint'. If both are configured, 'amService' takes precedence.

`"endpoint"`: *configuration expression<url>, required if* `amService` *is not configured*

The URI for the authorization service.

Configure either 'amService' or 'endpoint'. If both are configured, 'amService' takes precedence.

`"subjectTokenType"`: *configuration expression<string>, optional*

The subject token type.

Default: `URN_ACCESS_TOKEN`

`"requestedTokenType"`: *configuration expression<string>, optional*

The type of token being requested.

Default: `URN_ACCESS_TOKEN`

`"scopes"`: *array of runtime expression<strings> or ResourceAccess <reference>, required*

A list of one or more scopes required by the OAuth 2.0 access token. Provide the scopes as strings or through a ResourceAccess such as a ScriptableResourceAccess:

Array of runtime expression<strings>, required if a ResourceAccess isn't used

A string, array of strings, runtime expression<string>, or array of runtime expression<string> to represent one or more scopes.

ScriptableResourceAccess <reference>

A script that evaluates each request dynamically and returns the scopes that the request needs to access the protected resource. The script must return a `Set<String>`.

For information about the properties of ScriptableResourceAccess, refer to Scripts.

```
{
  "name": string,
  "type": "ScriptableResourceAccess",
  "config": {
    "type": configuration expression<string>,
    "file": configuration expression<string>, // Use either "file"
    "source": [ string, ... ],               // or "source", but not both.
    "args": object,
    "clientHandler": Handler reference
  }
}
```

Default: Empty

`"resource"`: *configuration expression<url>, optional*

The target service URI where the issued token is intended to be used.

`"audience"`: *configuration expression<url>, optional*

The target service name where the token is intended to be used.

`"endpointHandler"`: *Handler reference, optional*

The handler to exchange tokens on the authorization endpoint.

Configure this property as a Chain, using one of the following filters for client authentication:

- ClientSecretBasicAuthenticationFilter
- ClientSecretPostAuthenticationFilter
- EncryptedPrivateKeyJwtClientAuthenticationFilter
- PrivateKeyJwtClientAuthenticationFilter

```
{
  "name": "EndpointHandler",
  "type": "Chain",
  "config": {
    "handler": "ForgeRockClientHandler",
    "filters": [
      {
        "type": "ClientSecretBasicAuthenticationFilter",
        "config": {
          "clientId": "serviceConfidentialClient",
          "clientSecretId": "client.secret.id",
          "secretsProvider" : "SystemAndEnvSecretStore-1",
        }
      }
    ]
  }
}
```

Default: ForgeRockClientHandler

`"failureHandler"`: *Handler <reference>, optional*

Handler to manage a failed request.

Provide an inline handler configuration object or the name of a handler object declared in the heap. The handler can access information in the OAuth2FailureContext.

Default: `500 Internal Server Error`, the request stops being executed.

Example

For an example of how this filter is used, refer to Token exchange.

More information

org.forgerock.http.oauth2.OAuth2TokenExchangeFilter

OAuth2TokenExchangeContext

OAuth2FailureContext

The OAuth 2.0 Authorization Framework

PasswordReplayFilter

Extracts credentials from AM and replays them to a login page or to the next filter or handler in the chain. The PasswordReplayFilter does not retry failed authentication attempts.

> **Important**
>
> The PasswordReplayFilter filter uses the AM Post Authentication Plugin `com.sun.identity.authentication.spi.JwtReplayPassword`. The plugin is triggered for AM authentication chains but not currently for AM authentication trees.
> Don't use the PasswordReplayFilter with AM authentication trees.

Usage

```
{
  "name": string,
  "type": "PasswordReplayFilter",
  "config": {
    "request": object,
    "loginPage": runtime expression<boolean>,
    "loginPageContentMarker": pattern,
    "credentials": Filter reference,
    "loginPageExtractions": [ object, ... ]
  }
}
```

Properties

`"request"`: *<object>, required*

The HTTP request message that replays the credentials.

```
{
  "request": object,
    "method": config expression<string>,
    "uri": runtime expression<string>,
    "version": configuration expression<string>,
    "entity": runtime expression<string>,
    "headers": map,
    "form": map
}
```

For information about the properties of `request`refer to Request.

The JSON object of `request` is the `config` content of a StaticRequestFilter.

`"loginPage"`: *runtime expression<boolean>, required unless* `loginPageContentMarker` *is defined*

`true`: Direct the request to a login page, extract credentials, and replay them.

`false`: Pass the request unchanged to the next filter or handler in the chain.

The following example expression resolves to `true` when the request is an HTTP GET, and the request URI path is `/login`:

```
${find(request.uri.path, '/login') and (request.method == 'GET')}
```

`"loginPageContentMarker"`: *pattern, required unless* `loginPage` *is defined*

A pattern that matches when a response entity is a login page.

For an example route that uses this property, refer to Login form with password replay and cookie filters.

See also Patterns.

`"credentials"`: *Filter reference, optional*

Filter that injects credentials, making them available for replay. Consider using a `FileAttributesFilter` or an `SqlAttributesFilter`.

When this is not specified, credentials must be made available to the request by other means.

See also Filters.

`"loginPageExtractions"`: *array of <objects>, optional*

Objects to extract values from the login page entity.

```
{
  "loginPageExtractions": [
    {
      "name": string,
      "pattern": pattern
    },
    ...
  ]
}
```

For an example route that uses this property, refer to Login which requires a hidden value from the login page.

The extract configuration array is a series of configuration objects. To extract multiple values, use multiple extract configuration objects. Each object has the following fields:

`"name"`: *string*, *required*

Name of the field where the extracted value is put.

The names are mapped into `attributes.extracted`.

For example, if the name is `nonce`, the value can be obtained with the expression `${attributes.extracted.nonce}`.

The name `isLoginPage` is reserved to hold a boolean that indicates whether the response entity is a login page.

`"pattern"`: *pattern*, *required*

The regular expression pattern to find in the entity.

The pattern must contain one capturing group. (If it contains more than one, only the value matching the first group is placed into `attributes.extracted`.)

For example, suppose the login page entity contains a nonce required to authenticate, and the nonce in the page looks like `nonce='n-0S6_WzA2Mj'`. To extract `n-0S6_WzA2Mj`, set `"pattern": " nonce='(.*)'"`.

Example

The following example authenticates requests using static credentials when the request URI path is `/login`. This PasswordReplayFilter example does not include any mechanism for remembering when authentication has already been successful, it simply replays the authentication every time that the request URI path is `/login`:

```
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "type": "PasswordReplayFilter",
        "config": {
          "loginPage": "${request.uri.path == '/login'}",
          "request": {
            "method": "POST",
            "uri": "https://www.example.com:8444/login",
            "form": {
              "username": [
                "MY_USERNAME"
              ],
              "password": [
                "MY_PASSWORD"
              ]
            }
          }
        }
      }],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

For additional examples, refer to Configuration templates, and the Javadoc for the PasswordReplayFilter class.

More information

org.forgerock.openig.filter.PasswordReplayFilterHeaplet

PingOneApiAccessManagementFilter

> **Important**
>
> The PingOneApiAccessManagementFilter is available in Technology preview. It isn't yet supported, may be functionally incomplete, and is subject to change without notice.

Use the PingOneApiAccessManagementFilter with PingOne's API Access Management, where the PingOne API moderates requests and responses as follows:

- Allows requests, optionally instructing IG to edit the requests.
- Rejects requests, instructing IG on how to respond to the client, for example, with an HTTP 403 and a custom message.
- Instructs IG to update responses from the backend. For example, the instructions can be to remove content from the response body or to add or remove headers.

The filter sends the following elements to the PingOne API for the request:

- Client IP address
- Client port

- HTTP method used
- URL targeted
- HTTP version used
- HTTP headers
- HTTP content (when `includeBody` = `true` and the content is JSON)

The filter sends the following elements to the PingOne API for the response:

- Original URL queried
- Original method called
- HTTP status code
- HTTP status message
- HTTP version
- HTTP headers
- HTTP content (when `includeBody` = `true` and the content is JSON)

Usage

```
{
  "name": string,
  "type": "PingOneApiAccessManagementFilter",
    "config": {
      "gatewayServiceUri": configuration expression<url>,
      "secretsProvider": SecretsProvider reference,
      "gatewayCredentialSecretId": configuration expression<secret-id>,
      "includeBody": configuration expression<boolean>,
      "sidebandHandler": Handler reference
  }
}
```

Configuration

`"gatewayServiceUri"`: *configuration expression<url>, required*

The URL of the API gateway in the PingOne API.

To find the URL, go to your PingOne Authorize environment, select **Authorization** > **API gateways**, and note the value of the > **Service URL**.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for the credential to access the PingOne API.

`"gatewayCredentialSecretId"`: *configuration expression<secret-id>, required*

The secret ID of the PingOne API credential.

The secret ID must point to a GenericSecret in the `secretsProvider`.

To add the credential, go to your PingOne Authorize environment, select **Authorization** > **API gateways**, and select your gateway.

`"includeBody"`: *configuration expression<boolean>, optional*

A flag to include the body of requests and responses sent from IG to the PingOne API.

> **Note**
>
> IG includes the body only when the body is in JSON format.

> **Note**
>
> Including the body in every request and response can impact the HTTP exchange latency.

Default: `true`

`"sidebandHandler"`: *Handler reference, optional*

An HTTP client handler to use to contact the PingOne API.

The handler sends requests and responses to the Ping Sideband API. It then processes Ping Sideband API decisions to accept, reject, or rewrite requests and responses.

Default: ForgeRockClientHandler

More information

org.forgerock.openig.ping.PingOneApiAccessManagementFilter

PolicyEnforcementFilter

Requests and enforces policy decisions from AM. For more information, refer to IG's Policy enforcement and AM's Authorization guide.

Attributes and advices are stored in the `policyDecision` context. For information, refer to PolicyDecisionContext.

When the PolicyEnforcementFilter is preceded by a SingleSignOnFilter or CrossDomainSingleSignOnFilter in a Chain, it can respond to the following advice types from AM:

- `AuthLevel`: The minimum authentication level at which a user agent must authenticate to access a resource.
- `AuthenticateToService`: The name of an authorization chain or service to which a user agent must authenticate to access a resource.
- `AuthenticateToRealm`: The name of a realm to which a user agent must authenticate to access a resource.
- `AuthScheme`: The name of an authentication module to which a user agent must authenticate to access a resource, the policy set name, and the authentication timeout.
- `Transaction`: The additional actions that a user agent must perform before having a one-time access to the protected resource.

When the PolicyEnforcementFilter isn't preceded by a SingleSignOnFilter or CrossDomainSingleSignOnFilter in a Chain, it can't respond to advices from AM. Requests that return policy decisions with advices fail with an HTTP 403 Forbidden.

Notes on configuring policies in AM

In the AM policy, remember to configure the `Resources` parameter with the URI of the protected application.

The request URI from IG must match the `Resources` parameter defined in the AM policy. If the URI of the incoming request is changed before it enters the policy filter (for example, by rebasing or scripting), remember to change the `Resources` parameter in AM policy accordingly.

WebSocket notifications for policy changes

When WebSocket notifications are set up for changes to policies, IG receives a notification from AM when a policy decision is created, deleted, or updated.

For information about setting up WebSocket notifications, using them to clear the policy cache, and including them in the server logs, refer to WebSocket Notifications.

Usage

```
{
  "name": string,
  "type": "PolicyEnforcementFilter",
  "config": {
    "amService": AmService reference,
    "pepRealm": configuration expression<string>,
    "ssoTokenSubject": runtime expression<string>,
    "jwtSubject": runtime expression<string>,
    "claimsSubject": map or runtime expression<map>,
    "cache": object,
    "application": configuration expression<string>,
    "environment": map or runtime expression<map>,
    "failureHandler": Handler reference,
    "resourceUriProvider": ResourceUriProvider reference,
    "authenticateResponseRequestHeader": configuration expression<string>,
    "useLegacyAdviceEncoding": configuration expression<boolean> //deprecated
  }
}
```

Properties

"amService": *AmService reference, required*

The AM instance to use for policy decisions.

"pepRealm": *configuration expression<string>, optional*

The AM realm where the policy set is located.

Default: The realm declared for `amService`.

`"ssoTokenSubject"`: *runtime expression<string>, required if neither of the following properties are present:* `jwtSubject`, `claimsSubject`

The AM token ID string for the subject making the request to the protected resource.

`ssoTokenSubject` can take the value of the session token from the following sources:

- When the PolicyEnforcementFilter is preceded by a SingleSignOnFilter, `${contexts.ssoToken.value}`.
- When the PolicyEnforcementFilter is preceded by a CrossDomainSingleSignOnFilter, `${contexts.ssoToken.value}` or `${contexts.cdsso.value}`.
- When the PolicyEnforcementFilter isn't preceded by a SingleSignOnFilter or CrossDomainSingleSignOnFilter, `ssoTokenSubject` usually points to the token value.

  The token value can be in the request message, a header, or a cookie. For example, the `ssoTokenSubject` can point to a header value such as `${request.headers.cookie name}`, where `cookie name` is the AM session cookie name.

  Requests that return a policy decision with advices fail with an HTTP 403 and no advice handling.

`"jwtSubject"`: *runtime expression<string>, required if neither of the following properties are present:* `ssoTokenSubject`, `claimsSubject`

The JWT string for the subject making the request to the protected resource.

To use the raw id_token (base64, not decoded) returned by the OpenID Connect Provider during authentication, place an `AuthorizationCodeOAuth2ClientFilter` filter before the PEP filter, and then use `${attributes.openid.id_token}` as the expression value.

See also AuthorizationCodeOAuth2ClientFilter and Expressions.

`"claimsSubject"`: *map or runtime expression<map>, required if neither of the following properties are present:* `jwtSubject`, `` `"ssoTokenSubject` ``

A map of one or more data pairs with the format `Map<String, Object>`, where:

- The key is the name of a claim
- The value is a claim object, or a runtime expression that evaluates to a claims object

The following formats are allowed:

```
{
  "claimsSubject": {
    "string": "runtime expression<object>",
    ...
  }
}
```

```
{
  "claimsSubject": "runtime expression<map>"
}
```

The claim `"sub"` must be specified; other claims are optional.

In the following example, the property is a map whose first value is a runtime expression that evaluates to a JWT claim for the subject, and whose second value is a JWT claim for the subject:

```
"claimsSubject": {
  "sub": "${attributes.subject_identifier}",
  "iss": "am.example.com"
}
```

In the following example, the property is a runtime expression that evaluates to a map with the format `Map<String, Object>`:

```
"claimsSubject": "${attributes.openid.id_token_claims}"
```

For an example that uses `claimsSubject` as a map, refer to Example policy enforcement using claimsSubject on this reference page.

`"application"`: *configuration expression<string>, optional*

The ID of the AM policy set to use when requesting policy decisions.

Default: `iPlanetAMWebAgentService`, provided by AM's default policy set

`cache`: *object, optional*

Enable and configure caching of policy decisions from AM, based on *Caffeine*. For more information, see the GitHub entry, Caffeine.

When a request matches a cached policy decision, IG can reuse the decision without asking AM for a new decision. When caching is disabled, IG must ask AM to make a decision for each request.

```
{
  "cache": {
    "enabled": configuration expression<boolean>,
    "defaultTimeout": configuration expression<duration>,
    "executor": Executor service reference,
    "maximumSize": configuration expression<number>,
    "maximumTimeToCache": configuration expression<duration>,
    "onNotificationDisconnection": configuration expression<enumeration>
  }
}
```

Default: Policy decisions are not cached.

> **Note**
>
> Policy decisions that contain advices are never cached.

The following code example caches AM policy decisions without advices for these times:

- One hour, when the policy decision doesn't provide a `ttl` value.
- The duration specified by the `ttl`, when `ttl` is shorter than one day.
- One day, when `ttl` is longer than one day.

```
"cache": {
  "enabled": true,
  "defaultTimeout": "1 hour",
  "maximumTimeToCache": "1 day"
}
```

`enabled`: *configuration expression<boolean>, optional*

Enable or disable caching of policy decisions.

Default: `false`

`defaultTimeout`: *configuration expression<duration>, optional*

The default duration for which to cache AM policy decisions.

If an AM policy decision provides a valid `ttl` value to specify the time until which the policy decision remains valid, IG uses that value or the `maxTimeout`.

Default: `1 minute`

`"executor"`: *Executor service reference, optional*

An executor service to schedule the execution of tasks, such as the eviction of entries in the cache.

Default: `ForkJoinPool.commonPool()`

`"maximumSize"`: *configuration expression<number>, optional*

The maximum number of entries the cache can contain.

Default: Unlimited/unbound.

`maximumTimeToCache`: *configuration expression<duration>, optional*

The maximum duration for which to cache AM policy decisions.

If the `ttl` value provided by the AM policy decision is after the current time plus the `maximumTimeToCache`, IG uses the `maximumTimeToCache`.

The duration cannot be `zero` or `unlimited`.

`onNotificationDisconnection`*: configuration expression<enumeration>, optional*

The strategy to manage the cache when the WebSocket notification service is disconnected, and IG receives no notifications for AM events. If the cache is not cleared it can become outdated, and IG can allow requests on revoked sessions or tokens.

Cached entries that expire naturally while the notification service is disconnected are removed from the cache.

Use one of the following values:

- `NEVER_CLEAR`
  - When the notification service is disconnected:
    - Continue to use the existing cache.
    - Deny access for requests that are not cached, but do not update the cache with these requests.
  - When the notification service is reconnected:
    - Continue to use the existing cache.
    - Query AM for incoming requests that are not found in the cache, and update the cache with these requests.
- `CLEAR_ON_DISCONNECT`
  - When the notification service is disconnected:
    - Clear the cache.
    - Deny access to all requests, but do not update the cache with these requests.
  - When the notification service is reconnected:
    - Query AM for all requests that are not found in the cache. (Because the cache was cleared, the cache is empty after reconnection.)
    - Update the cache with these requests.
- `CLEAR_ON_RECONNECT`
  - When the notification service is disconnected:
    - Continue to use the existing cache.
    - Deny access for requests that are not cached, but do not update the cache with these requests.
  - When the notification service is reconnected:
    - Query AM for all requests that are not found in the cache. (Because the cache was cleared, the cache is empty after reconnection.)
    - Update the cache with these requests.

Default: `CLEAR_ON_DISCONNECT`

`"environment"`: *map or runtime expression<map>, optional*

A map of one or more data pairs with the format `Map<String, Object>`, where:

- The key is the name of a field in the request environment or context, such as a request header
- The value is the object to forward to AM with a policy decision request, or a runtime expression that evaluates to the object

The following formats are allowed:

```
{
  "claimsSubject": {
    "string": "runtime expression<object>",
    ...
  }
}
```

```
{
  "claimsSubject": "runtime expression<map>"
}
```

AM uses environment conditions to set the circumstances under which a policy applies. For example, environment conditions can specify that the policy applies only during working hours or only when accessing from a specific IP address.

Forward any HTTP header or any value that the AM policy definition can use.

In the following example, the property is a map whose values are runtime expressions that evaluate to request headers, an ID token, and the IP address of the subject making the request:

```
"environment": {
  "H-Via": "${request.headers['Via']}",
  "H-X-Forwarded-For": "${request.headers['X-Forwarded-For']}",
  "H-myHeader": "${request.headers['myHeader']}",
  "id_token": [
    "${attributes.openid.id_token}"
  ],
  "IP": [
    "${contexts.client.remoteAddress}"
  ]
}
```

`"failureHandler"`: *Handler reference, optional*

Handler to treat the request if it is denied by the policy decision.

In the following example, the `failureHandler` is a chain with a scriptable filter. If there are some advices with the policy decision, the script recovers the advices for processing. Otherwise, it passes the request to the `StaticResponseHandler` to display a message.

```
"failureHandler": {
  "type": "Chain",
  "config": {
    "filters": [
      {
        "type": "ScriptableFilter",
        "config": {
          "type": "application/x-groovy",
          "source": [
            "if (contexts.policyDecision.advices['MyCustomAdvice'] != null) {",
            "  return handleCustomAdvice(context, request)",
            "} else {",
            "  return next.handle(context, request)",
            "}"
          ]
        }
      }
    ],
    "handler": {
      "type": "StaticResponseHandler",
      "config": {
        "status": 403,
        "headers": {
          "Content-Type": [ "text/plain; charset=UTF-8" ]
        },
        "entity": "Restricted area. You do not have sufficient privileges."
      }
    }
  }
}
```

Provide an inline handler configuration object or the name of a handler object declared in the heap. See also Handlers.

Default: HTTP 403 Forbidden, the request stops being executed.

`"resourceUriProvider"`: *ResourceUriProvider reference, optional*

Use one of the following providers to return a resource URL to include in policy decision requests to AM:

- RequestResourceUriProvider
- ScriptableResourceUriProvider

The PolicyEnforcementFilter uses the returned resource URL to identify the policy decision in the policy cache.

When a request matches a cached policy decision, IG can reuse the decision without asking AM for a new decision. When caching is disabled, IG must ask AM to make a decision for each request.

Default: `RequestResourceUriProvider` configured to use the request URI with all query parameters included.

Tip

Maximize the cache hit ratio by managing the returned resource URL in conjuction with AM policies.

Strip all query parameters from the returned resource URL

Consider the following AM policy that matches requests on the specified path. The policy ignores query parameters:

```
http://ig.example.com:8080/app
```

The following requests match the path but have additional query parameters:

```
http://ig.example.com:8080/app?day=monday
http://ig.example.com:8080/app?day=monday&place=london
http://ig.example.com:8080/app?day=monday&place=london&building=x
```

When `includeQueryParams` in RequestResourceUriProvider is `true`, the ResourceUriProvider includes all query parameters in requests for policy decisions. The PolicyEnforcementFilter requests a policy desicion for the first request `/app?day=monday` and caches the descision. The second request `app?day=monday&place=london` doesn't match the cached decision so the PolicyEnforcementFilter requests another policy decision and adds it to the cache. Similarly for the third request.

When `includeQueryParams` in RequestResourceUriProvider is `false`, the ResourceUriProvider strips all query parameters from the requests. The PolicyEnforcementFilter requests a policy decision for the first request without query parameters and caches the policy desicion. The following two requests without query parameters match the cached decision and IG uses the cached decision without consulting AM.

Include only specified query parameters in the returned resource URL

Consider a similar example where an AM policy matches requests on the specified path but also requires one query parameter:

```
http//ig.example.com:8080/app?day=monday
```

The following requests match the path and query parameter but two of them have additional query parameters:

```
http://ig.example.com:8080/app?day=monday
http://ig.example.com:8080/app?day=monday&place=london
http://ig.example.com:8080/app?day=monday&place=london&building=x
```

Because the policy requires a query parameter, you can't use RequestResourceUriProvider to strip all query parameters from the requests.

Instead, use ScriptableResourceUriProvider to include the `?day=monday` query parameter but strip all other query parameters.

Note

Query order is important. The following queries are semantically the same but don't match: `?day=monday&place=london` and `?place=london&day=monday`.

```
"resourceUriProvider": {
  "type": "ScriptableResourceUriProvider",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "// Define a list of parameters to keep",
      "def keepOnly = { [ 'place', 'day' ].contains(it.key) }",
      "// Build a new URI based on the original request URI",
      "return new MutableUri(request.uri).with { uri ->",
      "  // Build a filtered and normalized query string",
      "  uri.rawQuery = new Form().with { form ->",
      "    // Keep only the wanted parameters and sort by name",
      "    form.addAll(request.queryParams.findAll(keepOnly).sort())",
      "    return form.toQueryString()",
      "  }",
      "  // Return the full modified URI",
      "  return uri.toASCIIString()",
      "}"
    ]
  }
}
```

`authenticateResponseRequestHeader`: *configuration expression<string>, optional*

A header to include in a request to manage the way IG handles policy advices from AM. The header name and value is case-insensitive. The header value can be set as follows:

- `HEADER`: Return policy advices in a `WWW-Authenticate` header as base64-encoded JSON in a parameter called `advices`.
- Any other value: Return policy advices as parameters in a redirect response (default).

For information about how the header is used in policy enforcement, refer to Deny requests with advices in a header.

Default: `x-authenticate-response`

`useLegacyAdviceEncoding`: *configuration expression<boolean>, optional*

> **Important**
>
> The use of this property is deprecated and should be used only to support SDK in legacy installations. Refer to the Deprecated section of the *Release Notes*.

- True: Do not encode advices
- False: Encode advices with the encoder used by the AM version

Default: False

Examples

For examples of policy enforcement, refer to Policy enforcement.

More information

org.forgerock.openig.openam.PolicyEnforcementFilter

org.forgerock.openig.openam.PolicyDecisionContext

PolicyDecisionContext

AM's Authorization guide

PrivateKeyJwtClientAuthenticationFilter

Supports client authentication with the `private_key_jwt` client-assertion, using an unencrypted JWT.

Clients send a signed JWT to the Authorization Server. IG builds and signs the JWT, and prepares the request as in the following example:

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=...&
client_id=<clientregistration_id>&
client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer&
client_assertion=PHNhbWxwOl ... ZT
```

Use this filter with an endpoint handler that requires authentication with the with the `private_key_jwt` client-assertion, using an unencrypted JWT. For example, the `endpointHandler` handler in the OAuth2TokenExchangeFilter.

Usage

```
{
  "name": string,
  "type": "PrivateKeyJwtClientAuthenticationFilter",
  "config": {
    "clientId": configuration expression<string>,
    "tokenEndpoint": configuration expression<url>,
    "secretsProvider": SecretsProvider reference,
    "signingSecretId": configuration expression<secret-id>,
    "signingAlgorithm": configuration expression<string>,
    "jwtExpirationTimeout": configuration expression<duration>,
    "claims": map or configuration expression<map>
  }
}
```

Configuration

`"clientId"`: *configuration expression<string>, required*

The `client_id` obtained when registering with the Authorization Server.

`"tokenEndpoint"`: *configuration expression<url>, required*

The URL to the Authorization Server's OAuth 2.0 token endpoint.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for passwords and cryptographic keys.

`"signingSecretId"`: *configuration expression<string>, required*

Reference to the keys used to sign the JWT.

This secret ID must point to a CryptoKey.

`"signingAlgorithm"`: *configuration expression<string>, optional*

The JSON Web Algorithm (JWA) used to sign the JWT, such as:

- `RS256`: RSA using SHA-256
- `ES256`: ECDSA with SHA-256 and NIST standard P-256 elliptic curve
- `ES384`: ECDSA with SHA-384 and NIST standard P-384 elliptic curve
- `ES512`: ECDSA with SHA-512 and NIST standard P-521 elliptic curve

Default: `RS256`

`"jwtExpirationTimeout"`: *configuration expression<duration>, optional*

The duration for which the JWT is valid.

Default: 1 minute

`"claims"`: *map or configuration expression<map>, optional*

A map of one or more data pairs with the format `Map<String, Object>`, where:

- The key is the name of a claim used in authentication
- The value is the value of the claim, or a configuration expression that evaluates to the value

The following formats are allowed:

```
{
  "args": {
    "string": "configuration expression<string>",
    ...
  }
}
```

```
{
  "args": "configuration expression<map>"
}
```

Default: Empty

ResourceOwnerOAuth2ClientFilter

> **Important**
>
> This filter uses the *Resource Owner Password Credentials* grant type. According to information in the The OAuth 2.0 Authorization Framework, minimize use of this grant type and utilize other grant types whenever possible. Use this filter in a service-to-service context, where services need to access resources protected by OAuth 2.0.

Authenticates OAuth 2.0 clients by using the resource owner's OAuth 2.0 credentials to obtain an access token from an Authorization Server, and injecting the access token into the inbound request as a Bearer Authorization header.

Client authentication is provided by the `endpointHandler` property, which uses a client authentication filter.

The ResourceOwnerOAuth2ClientFilter refreshes the access token as required.

For more information, refer to RFC 6749 - Resource Owner Password Grant.

Usage

```
{
  "name": string,
  "type": "ResourceOwnerOAuth2ClientFilter",
  "config": {
    "username": configuration expression<string>,
    "passwordSecretId": configuration expression<secret-id>,
    "secretsProvider": SecretsProvider reference,
    "tokenEndpoint": configuration expression<url>,
    "scopes": [ configuration expression<string>, ... ],
    "endpointHandler": Handler reference
  }
}
```

Properties

`"username"`: *configuration expression<string>, required*

The resource owner username to supply during authentication.

`"passwordSecretId"`: *configuration expression<secret-id>, required*

The secret ID to obtain the resource owner password.

This secret ID must point to a GenericSecret.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for passwords and cryptographic keys.

`"tokenEndpoint"`: *configuration expression<url>, required*

The URL to the Authorization Server's OAuth 2.0 token endpoint.

`"scopes"`: *array of configuration expression<strings>, optional*

Array of scope strings to request from the Authorization Server.

Default: Empty, request no scopes.

`"endpointHandler"`: *Handler reference, optional*

The Handler to exchange tokens on the authorization endpoint.

Configure this property as a Chain, using one of the following client authentication filters:

- ClientSecretBasicAuthenticationFilter
- ClientSecretPostAuthenticationFilter
- PrivateKeyJwtClientAuthenticationFilter

```
{
  "name": "myHandler",
  "type": "Chain",
  "config": {
    "handler": "ForgeRockClientHandler",
    "filters": [
      {
        "type": "ClientSecretBasicAuthenticationFilter",
        "config": {
          "clientId": "myConfidentialClient",
          "clientSecretId": "my.client.secret.id",
          "secretsProvider" : "mySystemAndEnvSecretStore",
        }
      }
    ]
  }
}
```

Default: ForgeRockClientHandler

Examples

For an example, refer to Using OAuth 2.0 resource owner password credentials.

More information

org.forgerock.openig.filter.oauth2.client.ResourceOwnerOAuth2ClientFilterHeaplet

org.forgerock.openig.filter.oauth2.OAuth2ResourceServerFilterHeaplet

OAuth2ResourceServerFilter

The OAuth 2.0 Authorization Framework

The OAuth 2.0 Authorization Framework: Bearer Token Usage

SamlFederationFilter

Initiates the login or logout of a SAML 2.0 Service Provider (SP) with a SAML 2.0 Identity Provider (IDP). Login is initiated for requests that don't:

- Trigger a logout expression
- Match a SAML endpoint
- Include a valid session

Requests with a valid session are passed along the chain.

SAML in deployments with multiple instances of IG

When IG acts as a SAML service provider, session information is stored in the fedlet not the session cookie. In deployments with multiple instances of IG as a SAML service provider, it is necessary to set up sticky sessions so that requests always hit the instance where the SAML interaction was started.

For information, refer to Session state considerations in AM's *SAML v2.0 guide*.

Usage

```
{
  "name": string,
  "type": "SamlFederationFilter",
  "config": {
    "redirectURI": configuration expression<url>,
    "assertionMapping": map or configuration expression<map>,
    "subjectMapping": configuration expression<string>,
    "sessionIndexMapping": configuration expression<string>,
    "authnContext": configuration expression<string>,
    "authnContextDelimiter": configuration expression<string>,
    "assertionConsumerEndpoint": configuration expression<url>,
    "SPinitiatedSSOEndpoint": configuration expression<url>,
    "SPinitiatedSLOEndpoint": configuration expression<url>,
    "singleLogoutEndpoint": configuration expression<url>,
    "singleLogoutEndpointSoap": configuration expression<url>,
    "useOriginalUri": configuration expression<boolean>,
    "logoutExpression": runtime expression<boolean>,
    "logoutURI": configuration expression<url>,
    "redirectionMarker": object,
    "secretsProvider": SecretsProvider reference,
    "spEntityId": configuration expression<string>,
    "idpEntityId": configuration expression<string>,
    "failureHandler": Handler reference
  }
}
```

Properties

`"redirectURI"`: *configuration expression<url>, required*

The URI to use if there is no RelayState value.

`"assertionMapping"`: *map or configuration expression<map>, required*

A map with the format `Map<String, String>`, where:

- Key: Session name, `localName`
- Value: SAML assertion name, `incomingName`, or a configuration expression that evaluates to the name

The following formats are allowed:

```
{
  "assertionMapping": {
    "string": "configuration expression<string>",
    ...
  }
}
```

```
{
  "assertionMapping": "configuration expression<map>"
}
```

In the following example, the session names `username` and `password` are mapped to SAML assertion names `mail` and `mailPassword`:

```
{
  "assertionMapping": {
    "username": "mail",
    "password": "mailPassword"
  }
}
```

If an incoming SAML assertion contains the following statement:

```
mail = demo@example.com
mailPassword = demopassword
```

Then the following values are set in the session:

```
username[0] = demo@example.com
password[0] = demopassword
```

For this to work, edit the `<Attribute name="attributeMap">` element in the SP extended metadata file, `$HOME/.openig/SAML/sp-extended.xml`, so that it matches the assertion mapping configured in the SAML 2.0 Identity Provider (IDP) metadata.

Because the dot character (`.`) serves as a query separator in expressions, do not use dot characters in the localName.

To prevent different handlers from overwriting each others' data, use unique localName settings when protecting multiple service providers.

`"subjectMapping"`: *configuration expression<string>, optional*

Name of the session field to hold the value of the subject name. Because the dot character (`.`) serves as a query separator in expressions, do not use dot characters in the field name.

Use this setting when protecting multiple service providers, as the different configurations must not map their data into the same fields of `session`. Otherwise different handlers can overwrite each others' data.

As an example, if you set `"subjectMapping": "mySubjectName"`, then IG sets `session.mySubjectName` to the subject name specified in the assertion. If the subject name is an opaque identifier, then this results in the session containing something like `"mySubjectName": "vt0…zuL"`.

Default: map to `session.subjectName`

`"sessionIndexMapping"`: *configuration expression<string>, optional*

Name of the session field to hold the value of the session index. Because the dot character (`.`) serves as a query separator in expressions, do not use dot characters in the field name.

Use this setting when protecting multiple service providers, as the different configurations must not map their data into the same fields of `session`. Otherwise different handlers can overwrite each others' data.

As an example, if you set `"sessionIndexMapping": "mySessionIndex"`, then IG sets `session.mySessionIndex` to the session index specified in the assertion. This results in the session containing something like `"mySessionIndex": "s24c…801"`.

Default: map to `session.sessionIndex`

`"authnContext"`: *configuration expression<string>, optional*

Name of the session field to hold the value of the authentication context. Because the dot character (`.`) serves as a query separator in expressions, do not use dot characters in the field name.

Use this setting when protecting multiple service providers, as the different configurations must not map their data into the same fields of `session`. Otherwise different handlers can overwrite each others' data.

As an example, if you set `"authnContext": "myAuthnContext"`, then IG sets `session.myAuthnContext` to the authentication context specified in the assertion. When the authentication context is password over protected transport, then this results in the session containing `"myAuthnContext": "urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport"`.

Default: map to `session.authnContext`

`"authnContextDelimiter"`: *configuration expression<string>, optional*

The authentication context delimiter used when there are multiple authentication contexts in the assertion.

Default: `|`

`"assertionConsumerEndpoint"`: *configuration expression<string>, optional*

Part of the URI that designates the consumer endpoint as defined in the SP metadata shared with the IDP.

If you modify this attribute, change the metadata to match.

Default: `fedletapplication`

`"SPinitiatedSSOEndpoint"`: *configuration expression<string>, optional*

Part of the URI that designates the SP initiated SSO endpoint.

If you modify this attribute, change the metadata to match.

Default: `SPInitiatedSSO`

`"SPinitiatedSLOEndpoint"`: *configuration expression<string>, optional*

Part of the URI that designates the SP initiated SLO endpoint.

If you modify this attribute, change the metadata to match.

Default: `SPInitiatedSLO`

`"singleLogoutEndpoint"`: *configuration expression<string>, optional*

Part of the URI that designates the SP SLO endpoint as defined in the SP metadata shared with the IDP.

If you modify this attribute, change the metadata to match.

Default: `fedletSLORedirect` (same as the Fedlet)

`"singleLogoutEndpointSoap"`: *configuration expression<string>, optional*

Part of the URI that designates the SP SLO SOAP endpoint as defined in the SPs metadata shared with the IDP.

If you modify this attribute, change the metadata to match.

Default: `fedletSloSoap` (same as the Fedlet)

`"useOriginalUri"`: *configuration expression<boolean>, optional*

When `true`, use the original URI instead of a rebased URI to validate RelayState and Assertion Consumer Location URLs. Use this property if a `baseUri` decorator is used in the route or in `config.json`.

Default: `true`

`"logoutExpression"`: *runtime expression<boolean>, optional*

A flag to indicate whether a request initiates logout processing before reaching the protected application.

- `false`: The request does not initiate logout processing:
  - If a valid SAML session is found, the request is forwarded to the protected application.
  - If a valid SAML session is not found, the request triggers the SAML login flow.

- `true`: The request initiates logout processing:
  - If a valid SAML session is found, the request triggers the SAML logout flow:
    - If there is a RelayState URL parameter, the request is forwarded to that URL. RelayState provides backwards compatibility for SamlFederationHandler.
    - If there is no RelayState URL parameter and `logoutURI` is defined, the request is forwarded to the logout page.
    - If there is no RelayState URL parameter and `logoutURI` is not defined, the request is forwarded to the protected application without any other validation.
  - If a valid session is not found, the request is forwarded to the protected application without any other validation.

Important

To prevent unwanted access to the protected application, use `logoutExpression` with extreme caution as follows:

- Define a `logoutURI`.
- If you don't define a `logoutURI`, specify `logoutExpression` to resolve to `true` **only** for requests that target dedicated logout pages of the protected application.

Consider the following examples when a `logoutURI` is not defined:

- This expression resolves to `true` **only** for requests with `/app/logout` in their path:

```
"logoutExpression": "${startsWith(request.uri.rawPath, '/app/logout')}"
```

When a request matches the expression, the SAML session is revoked and the request is forwarded to the `/app/logout` page.

- This expression resolves to `true` for **all** requests that contain `logOff=true` in their query parameters:

```
"logoutExpression": "${find(request.uri.query, 'logOff=true')}"
```

When a request matches the expression, the SAML session is revoked and the request is forwarded to the protected application without any other validation. In this example, an attacker can bypass IG's security mechanisms by simply adding `?logOff=true` to a request.

Default: `${false}`

`"logoutURI"`: *configuration expression<string>, optional*

The URL to which a request is redirected if `logoutExpression` is evaluated as `true` or when the protected application uses the single logout feature of the Identity Provider.

Default: None, processing continues.

`"redirectionMarker"`: *configuration expression<object>, optional*

A redirect marker for the SAML flow. If the marker is present in the flow, the request isn't redirected for authentication.

This feature is on by default to prevent redirect loops when the session cookie isn't present in the SAML flow. The cookie can be absent from the flow if it doesn't include IG's domain.

```
"redirectionMarker": {
  "enabled": configuration expression<boolean>,
  "name": configuration expression<string>
}
```

`"enabled"`*: configuration expression<boolean>, optional*

- `true`: When the session is empty or invalid, IG checks the request `goto` query parameter for the presence of the redirection marker:
  - If the redirection marker is present, IG fails the request.
  - If the redirection marker isn't present, IG redirects the user agent for login.
- false: IG never checks the request `goto` query parameter for the presence of a redirection marker.

Default: `true`

`"name"`*: configuration expression<string>, optional*

The name of the redirection marker query parameter to use when `enabled` is `true`.

Default: `_ig`

`"secretsProvider"`*: SecretsProvider reference, optional*

The SecretsProvider to query for keys when AM provides signed or encrypted SAML assertions.

When this property isn't set, the keys are provided by direct keystore look-ups based on entries in the SP extended metadata file, `sp-extended.xml`.

`"spEntityId"`*: configuration expression<string>, optional*

The entity ID that this SP represents. Configure this property when more than one SP is defined in the metadata.

Default:

- When no SPs are defined in the metadata an error is generated.
- When there one SP defined in the metadata the filter uses that SP.
- When there is more than one SP defined in the metadata the filter uses the first SP in the list of discovered metadata and logs a warning. Because ordering is not deterministic, the discovered SP can be the wrong SP.

`"idpEntityId"`*: configuration expression<string>, optional*

The entity ID that this IDP represents. Configure this property when more than one IDP is defined in the metadata.

Default:

- When no IDPs are defined in the metadata an error is generated.
- When one IDP is defined in the metadata the filter uses that IDP.
- When there is more than one IDP defined in the metadata the filter uses the first IDP in the list of discovered metadata and logs a warning. Because ordering is not deterministic, the discovered IDP can be the wrong IDP.

`"failureHandler"`: *Handler reference, optional*

Handler to invoke when SAML processing fails.

Provide an inline handler configuration object or the name of a handler object declared in the heap. See also Handlers.

Default: Return an error response containing a SAML processing error.

ScriptableFilter

Processes requests and responses by executing a Groovy script. Executed scripts must return one of the following:

- Promise<Response, NeverThrowsException>
- Response

To execute the next element in a chain (a filter or a handler), the script must call the expression `next.handle(context, request)`. If the script does not call `next.handle(context, request)`, the chain flow breaks and the script has to build and return its own response by calling one of the following expressions:

- `return myResponse`
- `return newResultPromise(myResponse)`

Actions on the response returned from the downstream flow must be performed in the Promise's callback methods.

For information about script properties, available global objects, and automatically imported classes, refer to Scripts. For information about creating scriptable objects in Studio, refer to Scripts in Studio and Configure scriptable throttling.

Usage

```
{
    "name": string,
    "type": "ScriptableFilter",
    "config": {
        "type": configuration expression<string>,
        "file": configuration expression<string>, // Use either "file"
        "source": [ string, ... ],               // or "source", but not both.
        "args": map,
        "clientHandler": Handler reference
    }
}
```

Properties

For information about properties for ScriptableFilter, refer to Scripts.

Examples

For an example scriptable filter that recovers policy advices from AM, see the `failureHandler` property of PolicyEnforcementFilter.

More information

Scripts

org.forgerock.openig.filter.ScriptableFilter

SessionInfoFilter

Calls the AM endpoint for session information, and makes the data available as a new context to downstream IG filters and handlers. For information, refer to SessionInfoContext.

WebSocket notifications for sessions

When WebSocket notifications are set up for sessions, IG receives a notification from AM when a user logs out of AM, or when the AM session is modified, closed, or times out. IG then evicts entries that are related to the event from the `sessionCache`.

For information about setting up WebSocket notifications, using them to clear the session cache, and including them in the server logs, refer to WebSocket notifications.

Usage

```
{
  "name": string,
  "type": "SessionInfoFilter",
  "config": {
    "amService": AmService reference,
    "ssoToken": runtime expression<string>
  }
}
```

Properties

`"amService"`: *AmService reference, required*

The AmService object to use for communication with AM.

The following `sessionProperties`, are retrieved from AM:

- When `sessionProperties` in AmService is configured, listed session properties with a value.
- When `sessionProperties` in AmService is not configured, all session properties with a value.
- Properties with a value that are required by IG but not specified by `sessionProperties` in AmService. For example, when the session cache is enabled, session properties related to the cache are automatically retrieved.

Properties with a value are returned, properties with a null value are not returned.

`"ssoToken"`: *runtime expression<string>, optional*

Location of the AM SSO or CDSSO token.

This property can take the following values:

- `${contexts.ssoToken.value}`, when the SingleSignOnFilter is used for authentication

- `${contexts.ssoToken.value}` or `${contexts.cdsso.token}`, when the CrossDomainSingleSignOnFilter is used for authentication
- `${request.headers['mySsoToken'][0]}`, where the SSO or CDSSO token is the first value of the `mySsoToken` header in the request.

Default: `${request.cookies['AmService-ssoTokenHeader'][0].value}`, where `AmService-ssoTokenHeader` is the name of the header or cookie where the AmService expects to find SSO or CDSSO tokens.

Examples

For an example that uses the SessionInfoFilter, refer to Retrieve a Username From the sessionInfo Context.

More information

org.forgerock.openig.openam.SessionInfoFilter

org.forgerock.openig.openam.SessionInfoContext

SessionInfoContext

AM's Authorization guide

SetCookieUpdateFilter

Updates the attribute values of Set-Cookie headers in a cookie. This filter facilitates the transition to the SameSite and secure cookie settings required by newer browsers. Use SetCookieUpdateFilter at the beginning of a chain to guarantee security along the chain.

Set-Cookie headers must conform to grammar in RFC 6265: Set-Cookie.

Usage

```
{
  "name": string,
  "type": "SetCookieUpdateFilter",
  "config": {
   "cookies": {
     "cookie-name": {
       "attribute-name": "attribute-value",
       ...
     }
     ...
   }
 }
}
```

Properties

`"cookies"`: *map*, *required*

Configuration that matches case-sensitive cookie names to response cookies, and specifies how matching cookies attribute values should be updated. Each cookie begins with a name-value pair, where the value is one or more attribute-value pairs.

cookie-name: pattern, required

The name of a cookie contained in the `Set-Cookie` header, as a pattern.

To change the attribute value on all existing cookies, specify `.*`.

If a cookie is named more than once, either explicitly or by the wildcard (`*`), the rules are applied to the cookie in the order they appear in the map.

In the following example, the SameSite attribute of the CSRF cookie first takes the value `none`, and then that value is overwritten by the value `LAX`.

```
"cookies": {
  "CSRF": {
    "value": "myValue",
    "secure": ${true},
    "SameSite": "none"
  }
  ".*": {
    "SameSite": "LAX"
  }
}
```

attribute-name: enumeration, required

A case-insensitive enumeration of a Set-Cookie attribute name.

Attribute names include `SameSite`, `secure`, `http-only`, `value`, `expires`, `Max-Age`, `path`, and `domain`. For more information, refer to RFC 6265: Set-Cookie.

Use the `now` dynamic binding to dynamically set the value of a cookie attribute that represents time. For example, set the value of the attribute `expires` to one day after the expression is evaluated, as follows:

```
{
  "type": "SetCookieUpdateFilter",
    "config": {
      "cookies": {
        ".*": {
          "expires": "${now.plusDays(1).rfc1123}",
...
```

attribute-value: runtime expression<string, boolean, or integer>, required

The replacement value for the named attribute. The value must conform to the expected type for the attribute name:

- `secure`: runtime expression<boolean>. Required if `SameSite` is `none`
- `http-only`: runtime expression<boolean>.
- `Max-Age`: runtime expression<number>.
- `SameSite`, and all other attribute names: runtime expression<string>.

For all values except `expires`, specify `${previous}` to reuse the existing value for the attribute. The following example adds five seconds to the `Max-Age` attribute:

```
"Max-Age": "${integer(previous+5)}",
```

If the named the Set-Cookie header doesn't contain the named attribute, `${previous}` returns null.

Examples

The following example updates attributes of all existing Set-Cookie headers:

```
{
  "name": "SetCookieUpdateFilter",
  "condition": "${find(request.uri.path, '/home')}",
  "baseURI": "http://app.example.com:8081",
  "heap": [],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "type": "SetCookieUpdateFilter",
        "config": {
          "cookies": {
            ".*": {
              "SameSite": "LAX",
              "domain": "ig.example.com",
              "Max-Age": "${session.maxAge}",
              "Secure": "${true}",
              "expires": 155...833
            }
          }
        }
      }],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

More information

org.forgerock.openig.filter.SetCookieUpdateFilter

SingleSignOnFilter

When this filter processes a request, it injects the SSO token, the session user ID, and the full claims set into the SsoTokenContext.

For an example of how to configure SSO and information about the SSO data flow, refer to Single sign-on.

Tip

To prevent issues with performance when accessing large resources, such as .jpg and .js files, consider using the SingleSignOnFilter with the following options:

- The `sessionCache`, so that IG can reuse session token information without repeatedly asking AM to verify the session token.
- A ConditionalFilter, so that requests to access large resources skip the SingleSignOnFilter. For an example configuration, see the example in ConditionalFilter.

Note

When AM is using CTS-based sessions, it does not monitor idle time for client-side sessions, and so refresh requests are ignored.
When the SingleSignOnFilter is used for authentication with AM, after a time AM can view the session as idle even though the user continues to interact with IG. The user session can eventually time out.
When AM is using CTS-based sessions, use the `sessionIdleRefresh` property of AmService to refresh idle sessions, and prevent unwanted timeouts.

WebSocket notifications for sessions

When WebSocket notifications are set up for sessions, IG receives a notification from AM when a user logs out of AM, or when the AM session is modified, closed, or times out. IG then evicts entries that are related to the event from the `sessionCache`.

For information about setting up WebSocket notifications, using them to clear the session cache, and including them in the server logs, refer to WebSocket notifications.

Usage

```
{
  "name": string,
  "type": "SingleSignOnFilter",
    "config": {
      "amService": AmService reference,
      "authenticationService": configuration expression<string>,
      "redirectionMarker": object,
      "defaultLogoutLandingPage": configuration expression<url>,
      "loginEndpoint": runtime expression<url>,
      "logoutExpression": runtime expression<boolean>
  }
}
```

Properties

`"amService"`: *AmService reference, required*

An AmService object to use for the following properties:

- `agent`, the credentials of the IG agent in AM. When the agent is authenticated, the token can be used for tasks such as getting the user's profile, making policy evaluations, and connecting to the AM notification endpoint.
- `realm`: Realm of the IG agent in AM.

- `url`, the URL of an AM service to use for session token validation and authentication when `loginEndpoint` is not specified.
- `ssoTokenHeader`, the name of the cookie that contains the session token created by AM.
- `amHandler`, the handler to use when communicating with AM to validate the token in the incoming request.
- `sessionCache`, the configuration of a cache for session information from AM.
- `version`: The version of the AM server.

  The AM version is derived as follows, in order of precedence:

  - Discovered value: AmService discovers the AM version. If `version` is configured with a different value, AmService ignores the value of `version` and issues a warning.
  - Value in `version`: AmService cannot discover the AM version, and `version` is configured.
  - Default value of AM 6: AmService cannot discover the AM version, and `version` is not configured.

`redirectionMarker`: *object, optional*

A redirect marker for the SSO flow. If the marker is present in the SSO flow, the request isn't redirected for authentication.

This feature is on by default to prevent redirect loops when the session cookie isn't present in the SSO flow. The cookie can be absent from the flow if it doesn't include IG's domain.

```
"redirectionMarker": {
  "enabled": configuration expression<boolean>,
  "name": configuration expression<string>
}
```

`"enabled"`: *configuration expression<boolean>, optional*

- `true`: When the session is empty or invalid, IG checks the request `goto` query parameter for the presence of the redirection marker:
  - If the redirection marker is present, IG fails the request.
  - If the redirection marker isn't present, IG redirects the user agent for login.
- false: IG never checks the request `goto` query parameter for the presence of a redirection marker.

Default: `true`

`"name"`: *configuration expression<string>, optional*

The name of the redirection marker query parameter to use when `enabled` is `true`.

Default: `_ig`

`"authenticationService"`: *configuration expression<string>,optional*

The name of an AM authentication tree or authentication chain to use for authentication.

> **Note**
>
> Use only authentication trees with ForgeRock Identity Cloud. Authentication modules and chains are not supported.

Default: AM's default authentication tree.

For more information about authentication trees and chains, refer to Authentication nodes and trees and Authentication modules and chains in AM's *Authentication and SSO* guide.

`"defaultLogoutLandingPage"`: *configuration expression<url>, optional*

The URL to which a request is redirected if `logoutExpression` is evaluated as true.

If this property is not an absolute URL, the request is redirected to the IG domain name.

This parameter is effective only when `logoutExpression` is specified.

Default: None, processing continues.

`"loginEndpoint"`: *runtime expression<url>, optional*

The URL of a service instance for the following tasks:

- Manage authentication and the location to which the request is redirected after authentication.
- Process policy advices after an AM policy decision denies a request with supported advices. The PolicyEnforcementFilter redirects the request to this URL, with information about how to meet the conditions in the advices.

  For examples of different advice types, and the conditions that cause AM to return advices, see AM's Authorization guide. For information about supported advice types in IG, refer to PolicyEnforcementFilter.

  Default: The value of `url` in `amService`

Authentication can be performed in the following ways:

- Directly through AM, with optional authentication parameters in the query string, such as `service`, `module`, and `realm`. For a list of authentication parameters that you can include in the query string, see Authenticating (browser) in AM's *Authentication and SSO* guide.

  The value must include a redirect with a `goto` parameter.

  The following example uses AM as the authentication service, and includes the `service` authentication parameter:

  ```
  "loginEndpoint": "https://am.example.com/openam?service=TwoFactor&goto=$
  {urlEncodeQueryParameterNameOrValue(contexts.router.originalUri)}"
  ```

- Through the URL of another application, with optional authentication parameters in the query string, such as `service`, `module`, and `realm`. The application must create a session with an AM instance to set an SSO token and return the request to the redirect location.

  The value can optionally include a redirect with a `goto` parameter or different parameter name.

The following example uses an authentication service that is not AM, and includes a redirect parameter:

```
"loginEndpoint": "https://authservice.example.com/auth?redirect=$
{urlEncodeQueryParameterNameOrValue(contexts.router.originalUri)}"
```

When using this option, review the cookie domains to make sure cookies set by the authentication server are properly conveyed to the IG instance.

`"logoutExpression"`: *runtime expression<boolean>, optional*

A flag to indicate whether a request initiates logout processing before reaching the protected application.

- `false`: The request does not initiate logout processing:
  - If a valid AM session is found, the request is forwarded to the protected application.
  - If a valid AM session is not found, the request triggers login.
- `true`: The request initiates logout processing:
  - If a valid AM session is found, the session is revoked and the request is forwarded as follows:
    - If `defaultLogoutLandingPage` is defined, the request is forwarded to the specified logout page.
    - If `defaultLogoutLandingPage` is not defined, the request is forwarded to the protected application without any other validation.
  - If a valid session is not found, the request is forwarded to the protected application without any other validation.

Important

To prevent unwanted access to the protected application, use `logoutExpression` with extreme caution as follows:

- Define a `defaultLogoutLandingPage`.
- If you don't define a `defaultLogoutLandingPage`, specify `logoutExpression` to resolve to `true` **only** for requests that target dedicated logout pages of the protected application.

Consider the following examples when a `defaultLogoutLandingPage` is not configured:

- This expression resolves to `true` **only** for requests with `/app/logout` in their path:

  ```
  "logoutExpression": ${startsWith(request.uri.rawPath, '/app/logout')}
  ```

  When a request matches the expression, the AM session is revoked and the request is forwarded to the `/app/logout` page.
- This expression resolves to `true` for **all** requests that contain `logOff=true` in their query parameters:

  ```
  "logoutExpression": ${find(request.uri.query, 'logOff=true')}
  ```

  When a request matches the expression, the AM session is revoked and the request is forwarded to the protected application without any other validation. In this example, an attacker can bypass IG's security mechanisms by simply adding `?logOff=true` to a request.

Default: `${false}`

More information

org.forgerock.openig.openam.SingleSignOnFilter

org.forgerock.openig.openam.SsoTokenContext

SsoTokenContext

SqlAttributesFilter

> **Caution**
>
> This filter uses synchronous architecture. Accessing the filter `target` triggers a call to the database that blocks the executing thread until the database responds.
> Consider the performance impact of this filter, especially for deployments with a small number of gateway units (therefore, a small number of executing threads) and a long execution time for the JDBC call.

Executes a SQL query through a prepared statement and exposes its first result. Parameters in the prepared statement are derived from expressions. The query result is exposed in an object whose location is specified by the `target` expression. If the query yields no result, then the resulting object is empty.

The execution of the query is performed lazily; it does not occur until the first attempt to access a value in the target. This defers the overhead of connection pool, network and database query processing until a value is first required. This also means that the parameters expressions is not evaluated until the object is first accessed.

Usage

```
{
  "name": string,
  "type": "SqlAttributesFilter",
   "config": {
     "dataSource": JdbcDataSource reference,
     "preparedStatement": configuration expression<string>,
     "parameters": [ runtime expression<string>, ... ],
     "target": lvalue-expression
   }
}
```

Properties

`"dataSource"`: *JdbcDataSource reference, required*

The JdbcDataSource to use for connections. Configure JdbcDataSource as described in JdbcDataSource.

`"preparedStatement"`: *configuration expression<string>, required*

The parameterized SQL query to execute, with `?` parameter placeholders.

`"parameters"`: *array of runtime expressions<strings>, optional*

The parameters to evaluate and include in the execution of the prepared statement.

See also Expressions.

`"target"`: *<lvalue-expression>, required*

Expression that yields the target object containing the query results. For example, `${target.sql.queryresult}`.

Access to `target` triggers a call to the database that blocks the executing thread until the database responds.

See also Expressions.

Example

Using the user's session ID from a cookie, query the database to find the user logged in and set the profile attributes in the attributes context:

```
{
        "name": "SqlAttributesFilter",
        "type": "SqlAttributesFilter",
        "config": {
              "target": "${attributes.sql}",
              "dataSource": "java:comp/env/jdbc/mysql",
              "preparedStatement": "SELECT f.value AS 'first', l.value AS 'last', u.mail AS 'email',
GROUP_CONCAT(CAST(r.rid AS CHAR)) AS 'roles' FROM sessions s INNER JOIN users u ON ( u.uid = s.uid AND u.status = 1 )
LEFT OUTER JOIN profile_values f ON ( f.uid = u.uid AND f.fid = 1 ) LEFT OUTER JOIN profile_values l ON ( l.uid =
u.uid AND l.fid = 2 ) LEFT OUTER JOIN users_roles r ON ( r.uid = u.uid ) WHERE (s.sid = ? AND s.uid <> 0) GROUP BY
s.sid;",
              "parameters": [ "${request.cookies[keyMatch(request.cookies,'JSESSION1234')][0].value}" ]
         }
 }
```

Lines are folded for readability in this example. In your JSON, keep the values for `"preparedStatement"` and `"parameters"` on one line.

More information

org.forgerock.openig.sql.SqlAttributesFilter

StaticRequestFilter

Creates a new request, replacing any existing request. The request can include an entity specified in the `entity` parameter. Alternatively, the request can include a form, specified in the `form` parameter, which is included in an entity encoded in `application/x-www-form-urlencoded` format if request method is `POST`, or otherwise as (additional) query parameters in the URI. The `form` and `entity` parameters cannot be used together when the `method` is set to `POST`.

Usage

```
{
  "name": string,
  "type": "StaticRequestFilter",
  "config": {
    "method": configuration expression<string>,
    "uri": runtime expression<url>,
    "version": configuration expression<string>,
    "headers": {
      configuration expression<string>: [ runtime expression<string>, ... ], ...
    },
    "form": {
      configuration expression<string>: [ runtime expression<string>, ... ], ...
    },
    "entity": runtime expression<string>
  }
}
```

Properties

`"method"`: *configuration expression<string>, required*

The HTTP method to be performed on the resource; for example, `GET`.

`"uri"`: *runtime expression<url>, required*

The fully-qualified URI of the resource being accessed; for example, `http://www.example.com/resource.txt`.

The result of the expression must be a string that represents a valid URI, but is not a real `java.net.URI` object. For example, it would be incorrect to use `${request.uri}`, which is not a string but a mutable URI.

`"version"`: *configuration expression<string>, optional*

Protocol version.

Default: `"HTTP/1.1"`

`"headers"`: *map, optional*

One or more headers to set for a request, with the format `name: [ value, … ]`, where:

- *name* is a configuration expression<string> that resolve to a header name. If multiple expressions resolve to the same final string, *name* has multiple values.
- *value* is one or more runtime expression<strings> that resolve to header values.

In the following example, the header name is the value of the system variable defined in `cookieHeaderName`. The header value is stored in `contexts.ssoToken.value`:

```
"headers": {
  "${application['header1Name']}": [
    "${application['header1Value'}"
  ]
}
```

Default: Empty

`"form"`: *map*, *optional*

A form to include in the request and/or `application/x-www-form-urlencoded` entity, as name-value pairs, where:

- *name* is a configuration expression<string> that resolves to a form parameter name.
- *value* is one or more runtime expression<strings> that resolve to form parameter values.

When a Request `method` is `POST`, `form` is mutually exclusive with `entity`.

Examples:

- In the following example, the field parameter names and values are hardcoded in the form:

```
"form": {
  "username": [
    "demo"
  ],
  "password": [
    "password"
  ]
}
```

- In the following example, the values take the first value of `username` and `password` provided in the session:

```
"form": {
  "username": [
    "${session.username[0]}"
  ],
  "password": [
    "${session.password[0]}"
  ]
}
```

- In the following example, the name of the first field parameter takes the value of the expression `${application['formName']}` when it is evaluated at startup. The values take the first value of `username` and `password` provided in the session:

```
"form": {
  "${application['formName']}": [
    "${session.username[0]}"
  ],
  "${application['formPassword']}": [
    "${session.password[0]}"
  ]
}
```

Default: Empty

`"entity"`: *runtime expression*<*string*>, *optional*

The message entity body to include in a request.

When a Request `method` is `POST`, `entity` is mutually exclusive with `form`.

Methods are provided for accessing the entity as byte, string, or JSON content. For information, refer to Entity.

> **Important**
>
> Attackers during reconnaissance can use messages to identify information about a deployment. For security, limit the amount of information in messages, and avoid using words that help identify IG.

Default: Empty

Example

In the following example, IG replaces the browser's original HTTP GET request with an HTTP POST login request containing credentials to authenticate to the sample application. For information about how to set up and test this example, refer to the Quick install.

```
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "StaticRequestFilter",
          "config": {
            "method": "POST",
            "uri": "http://app.example.com:8081/login",
            "form": {
              "username": [
                "demo"
              ],
              "password": [
                "Ch4ng31t"
              ]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${find(request.uri.path, '^/static')}"
}
```

More information

org.forgerock.openig.filter.StaticRequestFilter

SwitchFilter

Verifies that a specified condition is met. If the condition is met or no condition is specified, the request is diverted to the associated handler, with no further processing by the switch filter.

Usage

```
{
    "name": string,
    "type": "SwitchFilter",
    "config": {
        "onRequest": [
            {
                "condition": runtime expression<boolean>,
                "handler": Handler reference
            },
            ...
        ],
        "onResponse": [
            {
                "condition": runtime expression<boolean>,
                "handler": Handler reference
            },
            ...
        ]
    }
}
```

Properties

`"onRequest"`: *array of objects, optional*

Conditions to test (and handler to dispatch to, if `true`) before the request is handled.

`"onResponse"`: *array of objects, optional*

Conditions to test (and handler to dispatch to, if `true`) after the response is handled.

`"condition"`: *runtime expression<boolean>, optional*

A flag to indicate that a condition is met:

- `true`: The request is dispatched to the handler.
- `false`: The request is not dispatched to the handler, and the next condition in the list is tried.

  When the last condition in the list returns `false`, the request is passed to the next filter or handler in the chain.

Default: `${true}`

`"handler"`: *Handler reference, required*

The Handler to which IG dispaches the request if the associated condition yields `true`.

Provide the name of a Handler object defined in the heap or an inline Handler configuration object.

Example

This example intercepts the response if it is equal to 200 and executes the LoginRequestHandler. This filter might be used in a login flow where the request for the login page must go through to the target, but the response should be intercepted in order to send the login form to the application. This is typical for scenarios where there is a hidden value or cookie returned in the login page, which must be sent in the login form:

```
{
    "name": "SwitchFilter",
    "type": "SwitchFilter",
    "config": {
        "onResponse": [
            {
                "condition": "${response.status.code == 200}",
                "handler": "LoginRequestHandler"
            }
        ]
    }
}
```

More information

org.forgerock.openig.filter.SwitchFilter

ThrottlingFilter

Limits the rate that requests pass through a filter. The maximum number of requests that a client is allowed to make in a defined time is called the *throttling rate*.

When the throttling rate is reached, IG issues an HTTP status code 429 `Too Many Requests` and a `Retry-After` header, whose value is rounded up to the number of seconds to wait before trying the request again.

```
GET http://ig.example.com:8080/home/throttle-scriptable HTTP/1.1
. . .
HTTP/1.1 429 Too Many Requests
Retry-After: 10
```

Usage

```
{
    "name": string,
    "type": "ThrottlingFilter",
    "config": {
        "requestGroupingPolicy": runtime expression<string>,
        "throttlingRatePolicy": ThrottlingPolicy reference,  //Use either "throttlingRatePolicy"
        "rate": {                                            //or "rate", but not both.
            "numberOfRequests": configuration expression<number>,
            "duration": configuration expression<duration>
        },
        "cleaningInterval": configuration expression<duration>,
        "executor": ScheduledExecutorService reference
    }
}
```

Properties

`"requestGroupingPolicy"`: *runtime expression<string>, optional*

An expression to identify the partition to use for the request. In many cases the partition identifies an individual client that sends requests, but it can also identify a group that sends requests. The expression can evaluate to the client IP address or user ID, or an OpenID Connect subject/issuer.

The value for this expression must not be null.

Default: Empty string; all requests share the same partition

See also Expressions.

`"throttlingRatePolicy"`: *ThrottlingPolicy reference, required if* `rate` *is not used*

A reference to, or inline declaration of, a policy to apply for throttling rate. The following policies can be used:

- MappedThrottlingPolicy
- ScriptableThrottlingPolicy
- DefaultRateThrottlingPolicy

This value for this parameter must not be null.

`"rate"`: *object, required if* `throttlingRatePolicy` *is not used*

The throttling rate to apply to requests. The rate is calculated as the number of requests divided by the duration:

`"numberOfRequests"`: *configuration expression<integer>, required*

The number of requests allowed through the filter in the time specified by `"duration"`.

`"duration"`: *configuration expression<duration>, required*

A time interval during which the number of requests passing through the filter is counted.

`"cleaningInterval"`: *configuration expression<duration>, optional*

The time to wait before cleaning outdated partitions. The value must be more than zero but not more than one day.

`"executor"`: *ScheduledExecutorService reference, optional*

An executor service to schedule the execution of tasks, such as the clean up of partitions that are no longer used.

Default: `ScheduledExecutorService`

See also ScheduledExecutorService.

Examples

- Example of a Mapped Throttling Policy
- Example of a Scriptable Throttling Policy

The following route defines a throttling rate of 6 requests/10 seconds to requests. For information about how to set up and test this example, refer to Configure Simple Throttling.

```
{
  "name": "00-throttle-simple",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/throttle-simple')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ThrottlingFilter",
          "name": "ThrottlingFilter-1",
          "config": {
            "requestGroupingPolicy": "",
            "rate": {
              "numberOfRequests": 6,
              "duration": "10 s"
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

More information

org.forgerock.openig.filter.throttling.ThrottlingFilterHeaplet

TokenTransformationFilter

Transforms a token issued by AM to another token type.

The TokenTransformationFilter makes the result of the token transformation

available to downstream handlers in the `sts` context. For information, refer to StsContext.

The current implementation uses REST Security Token Service (STS) APIs to transform an OpenID Connect ID Token (`id_token`) into a SAML 2.0 assertion. The subject confirmation method is Bearer, as described in Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0.

The TokenTransformationFilter makes the result of the token transformation available to downstream handlers in the `issuedToken` property of the `${contexts.sts}` context.

The TokenTransformationFilter configuration references a REST STS instance that must be set up in AM before the TokenTransformationFilter can be used. The REST STS instance exposes a preconfigured transformation under a specific REST endpoint. For information about setting up a REST STS instance, see the AM documentation.

Errors that occur during the token transformation cause a error response to be returned to the client and an error message to be logged for the IG administrator.

Usage

```
{
    "name": "string",
    "type": "TokenTransformationFilter",
    "config": {
        "amService": AmService reference,
        "idToken": runtime expression<string>,
        "instance": configuration expression<string>,
        "username": configuration expression<string>, //deprecated
        "password": configuration expression<string>  //deprecated
    }
}
```

Properties

`"amService"`: *AmService reference, required*

The AmService heap object to use for the following properties:

- `agent`, the credentials of the IG agent in AM, to authenticate IG as an AM REST STS client, and to communicate WebSocket notifications from AM to IG. This credentials are evaluated when the route is initialized
- `url`, the URL of an AM service to use for session token validation and authentication. Authentication and REST STS requests are made to this service.
- `realm`, the AM realm containing the following information:
  - The AM application that can make the REST STS request and whose credentials are the username and password.
  - The STS instance described by the instance field.
- `ssoTokenHeader`, the name of the HTTP header that provides the SSO token for the REST STS client subject.
- `amHandler`, the handler to use for authentication and STS requests to AM.

`"idToken"`: *runtime expression<string>, required*

The value of the OpenID Connect ID token. The expected value is a string that is the JWT encoded `id_token`.

`"instance"`: *configuration expression<string>, required*

An expression evaluating to the name of the REST STS instance.

This expression is evaluated when the route is initialized, so the expression cannot refer to `request` or `contexts`.

`"username"`: *string, required*

> **Important**
>
> The use of this property is deprecated; use the AmService property `agent` instead. For more information, refer to the Deprecated section of the *Release Notes*.

The username to authenticate IG as an AM REST STS client.

`"password"`: *expression, required*

> **Important**
>
> The use of this property is deprecated; use the AmService property `agent` instead. For more information, refer to the Deprecated section of the *Release Notes*.

The password to authenticate IG as an AM REST STS client.

Example

The following example shows a configuration for a TokenTransformationFilter:

```
{
  "type": "TokenTransformationFilter",
  "config": {
    "amService": "MyAmService",
    "idToken": "${attributes.openid.id_token}",
    "instance": "openig"
  }
}
```

For an example of how to set up and test the TokenTransformationFilter, refer to Transform OpenID Connect ID tokens into SAML assertions.

More information

org.forgerock.openig.openam.TokenTransformationFilter

org.forgerock.openig.openam.StsContext

StsContext

TransactionIdOutboundFilter

Inserts the ID of a transaction into the header of a request.

The default TransactionIdOutboundFilter is created by IG, and used in ForgeRockClientHandler, as follows:

```
{
     "name": "ForgeRockClientHandler",
     "type": "ForgeRockClientHandler",
     "config": {
         "filters": [ "TransactionIdOutboundFilter" ],
         "handler": "ClientHandler"
     }
}
```

More information

org.forgerock.http.filter.TransactionIdOutboundFilter

UmaFilter

This filter acts as a policy enforcement point, protecting access as a User-Managed Access (UMA) resource server. Specifically, this filter ensures that a request for protected resources includes a valid requesting party token with appropriate scopes before allowing the response to flow back to the requesting party.

Usage

```
{
    "name": string,
    "type": "UmaFilter",
    "config": {
        "protectionApiHandler": Handler reference,
        "umaService": UmaService reference,
        "realm": configuration expression<string>
    }
}
```

Properties

`"protectionApiHandler"`: *Handler reference, required*

The handler to use when interacting with the UMA Authorization Server for token introspection and permission requests, such as a ClientHandler capable of making an HTTPS connection to the server.

For information, refer to Handlers.

`"umaService"`: *UmaService reference, required*

The UmaService to use when protecting resources.

For information, refer to UmaService.

`"realm"`: *configuration expression<string>, optional*

The UMA realm set in the response to a request for a protected resource that does not include a requesting party token enabling access to the resource.

Default: `uma`

More information

User-Managed Access (UMA) Profile of OAuth 2.0

org.forgerock.openig.uma.UmaResourceServerFilter

UriPathRewriteFilter

Rewrite a URL path, using a bidirectional mapping:

- In the request flow, `fromPath` is mapped to `toPath`.
- In the response flow, `toPath` is mapped to `fromPath`.

IG overwrites a response header only when all of the following conditions are true:

- The response includes a header such as `Location` or `Content-Location`
- The URI of the response header matches the mapping
- The value of response header is a relative path or its `scheme://host:port` value matches the base URI.

Usage

```
{
  "name": string,
  "type": "UriPathRewriteFilter",
  "config": {
    "mappings": object,
    "failureHandler": Handler reference
  }
}
```

Properties

`"mappings"`: *object, required*

One or more bidirectional mappings between URL paths. For example mappings, request scenarios, and an example route, refer to Examples.

```
{
  "mappings": {
    "/fromPath1": "/toPath1",
    "/fromPath2": "/toPath2",
    ...
  }
}
```

Paths are given by a `configuration expression<string>`. Consider the following points when you define paths:

- The incoming URL must start with the mapping path.
- When more than one mapping applies to a URL, the most specific mapping is used.
- Duplicate `fromPath` values are removed without warning.
- Trailing slashes `/` are removed from path values.
- If the response includes a `Location` or `Content-Location` header with a `toPath` in its URL, the response is rewritten with `fromPath`.

`"failureHandler"`: *handler reference, optional*

Failure handler to be invoked if an invalid URL is produced when the request path is mapped, or when the response `Location` or `Content-Location` header URI path is reverse-mapped.

Provide an inline handler declaration, or the name of a handler object defined in the heap. See also Handlers.

Default: HTTP 500

Examples

Valid and invalid mapping examples

The following mapping examples are valid:

- Single `fromPath` and `toPath`

```
"mappings": {
  "/fromPath1": "/toPath1",
  "/fromPath2": "/toPath2"
}
```

- Expressions in the `fromPath` and `toPath`

```
"mappings": {
  "/${join(array(`fromPath`, 'path1'), `/`)}": "/${join(array(`toPath`, 'path2'), `/`)}"
}
```

- Expressions in the `fromPath` and `toPath` that use predefined heap properties

```
"mappings": {
  "${fromPath}": "${toPath}"
}
```

- No mappings—the configuration is valid, but has no effect

```
"mappings": { }
```

- Duplicate `toPath`

```
"mappings": {
  "/fromPath1": "/toPath",
  "/fromPath2": "/toPath"
}
```

- Duplicate `fromPath`—the configuration is overwritten without warning

```
"mappings": {
  "/fromPath": "/toPath1",
  "/fromPath": "/toPath2"
}
```

The following mapping examples are not valid

- No `toPath`

```
"mappings": {
  "/fromPath": ""
}
```

```
"mappings": {
  "/fromPath": "${unknown}"
}
```

- Invalid `toPath`

```
"mappings": {
  "/fromPath": "${invalidExpression}"
}
```

- No `fromPath`

```
"mappings": {
  "": "/toPath"
}
```

```
"mappings": {
    "${unknown}": "/toPath"
}
```

- Invalid `fromPath`

```
"mappings": {
    "${invalidExpression}": "/toPath"
}
```

Example request scenarios

Description	Mapping	Inbound URI	Rewritten URI
Basic path	`"mappings": { "/fromPath": "/toPath" }`	http://example.com/fromPath/remainder	http://example.com/toPath/remainder
Root context, where the inbound request URI has a `/` path segment	`"mappings": { "/": "/rootcontext" }`	http://example.com/	http://example.com/rootcontext/
Root context, where the inbound URI has a `/` path segment	`"mappings": { "/rootcontext": "/" }`	http://example.com/rootcontext/	http://example.com/
Root context, where the inbound request URI has an empty path	`"mappings": { "/": "/rootcontext" }`	http://example.com	http://example.com/rootcontext
Root context, where the rewritten URI has an empty path	`"mappings": { "/rootcontext": "/" }`	http://example.com/rootcontext	http://example.com

Description	Mapping	Inbound URI	Rewritten URI
Root context, with path remainder	`"mappings": { "/": "/rootcontext" }`	http://example.com/remainder	http://example.com/rootcontext/remainder
Root context, with path remainder	`"mappings": { "/rootcontext": "/" }`	http://example.com/rootcontext/remainder	http://example.com/remainder
Root context, where the trailing `/` on `toPath` is ignored	`"mappings": { "/": "/rootcontext/" }`	http://example.com/remainder	http://example.com/rootcontext/remainder
Path with dot-segments:	`"mappings": { "/fromPath": "/toPath1/../toPath2" }`	http://example.com/fromPath	http://example.com/toPath1/../toPath2
Path with syntax:	`"mappings": { "/fromPath;v=1.1": "/toPath,1.1" }`	http://example.com/fromPath;v=1.1	http://example.com/toPath,1.1
Path with syntax:	`"mappings": { "/$fromPath": "/$toPath" }`	http://example.com/$fromPath	http://example.com/$toPath
Path with query parameters	`"mappings": { "/fromPath": "/toPath" }`	http://example.com/fromPath?param1&param2=2	http://example.com/toPath?param1&param2=2
Path with fragment	`"mappings": { "/fromPath": "/toPath" }`	http://example.com/fromPath#fragment	http://example.com/toPath#fragment

Example route

The example route changes a request URL as follows:

- The `baseURI` overrides the scheme, host, and port of a request URL.
- The UriPathRewriteFilter remaps the path of a request URL.

  Requests to `http://ig.example.com:8080/mylogin` are mapped to `http://app.example.com:8081/login`.

  Requests to `http://ig.example.com:8080/welcome` are mapped to `http://app.example.com:8081/home`.

  Requests to `http://ig.example.com:8080/other` are mapped to `http://app.example.com:8081/not-found`, and result in an HTTP 404.

  Requests to `http://ig.example.com:8080/badurl` are mapped to the invalid URL `http://app.example.com:8081[`, and invoke the failure handler.

```
{
  "name": "UriPathRewriteFilter",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "UriPathRewriteFilter",
          "config": {
            "mappings": {
              "/mylogin": "/login",
              "/welcome": "/home",
              "/other": "/not-found",
              "/badurl": "["
            },
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "status": 500,
                "headers": {
                  "Content-Type": [
                    "text/plain"
                  ]
                },
                "entity": "Invalid URL produced"
              }
            }
          }
        }
      ],
      "handler": "ClientHandler"
    }
  }
}
```

More information

org.forgerock.openig.filter.UriPathRewriteFilter

RFC 3986: Path

UserProfileFilter

Queries AM to retrieve the profile attributes of an user identified by their `username`.

Only profile attributes that are enabled in AM can be returned by the query. The `roles` field is not returned.

The data is made available to downstream IG filters and handlers, in the context UserProfileContext.

Usage

```
{
  "name": string,
  "type": "UserProfileFilter",
  "config": {
    "username": runtime expression<string>,
    "userProfileService": UserProfileService reference
  }
}
```

Properties

`"username"`*: runtime expression<string>, required*

The username of an AM subject. This filter retrieves profile attributes for the subject.

`"userProfileService"`*: UserProfileService reference, required*

The service to retrieve profile attributes from AM, for the subject identified by `username`.

```
"userProfileService": {
  "type": "UserProfileService",
  "config": {
    "amService": AmService reference,
    "cache": object,
    "profileAttributes": [ configuration expression<string>, ... ],
    "realm": configuration expression<string>
  }
}
```

`"amService"`*: AmService reference, required*

The AmService heap object to use for the following properties:

- `agent`, the credentials of the IG agent in AM. When the agent is authenticated, the token can be used for tasks such as getting the user's profile, making policy evaluations, and connecting to the AM notification endpoint.
  - `url`: URL of the AM server where the user is authenticated.
  - `amHandler`: Handler to use when communicating with AM to fetch the requested user's profile.
- `realm`: Realm of the IG agent in AM.

- `version`: The version of the AM server.

  The AM version is derived as follows, in order of precedence:

  - Discovered value: AmService discovers the AM version. If `version` is configured with a different value, AmService ignores the value of `version` and issues a warning.
  - Value in `version`: AmService cannot discover the AM version, and `version` is configured.
  - Default value of AM 6: AmService cannot discover the AM version, and `version` is not configured.

`"cache"`: *object*, *optional*

Caching of AM user profiles, based on *Caffeine*. For more information, see the GitHub entry, Caffeine.

When caching is enabled, IG can reuse cached profile attributes without repeatedly querying AM. When caching is disabled, IG must query AM for each request, to retrieve the required user profile attributes.

Default: No cache.

```
"cache": {
    "enabled": configuration expression<boolean>,
    "executor": Executor reference,
    "maximumSize": configuration expression<number>,
    "maximumTimeToCache": configuration expression<duration>,
}
```

`enabled`: *configuration expression*<*boolean*>,*optional*

Enable or disable caching of user profiles. When `false`, the cache is disabled but the cache configuration is maintained.

Default: `true` when `cache` is configured

`executor`: *Executor reference*, *optional*

An executor service to schedule the execution of tasks, such as the eviction of entries in the cache.

Default: `ForkJoinPool.commonPool()`

`"maximumSize"`: *configuration expression*<*number*>, *optional*

The maximum number of entries the cache can contain.

Default: Unlimited/unbound

`maximumTimeToCache`: *configuration expression*<*duration*>, *required*

The maximum duration for which to cache user profiles.

The duration cannot be `zero`.

`profileAttributes`: *array of configuration expression*<*strings*>, *optional*

List of one or more fields to return and store in UserProfileContext.

Field names are defined by the underlying repository in AM. When AM is installed with the default configuration, the repository is ForgeRock Directory Services.

The following convenience accessors are provided for commonly used fields:

- `cn`: Retrieved through `${contexts.userProfile.commonName}`
- `dn`: Retrieved through `${contexts.userProfile.distinguishedName}`
- `realm`: Retrieved through `${contexts.userProfile.realm}`
- `username`: Retrieved through `${contexts.userProfile.username}`

All other available fields can be retrieved through `${contexts.userProfile.rawInfo}` and `${contexts.userProfile.asJsonValue()}`.

When `profileAttributes` is configured, the specified fields **and** the following fields are returned: `username`, `_id`, and `_rev`.

Default: All available fields are returned.

`"realm"`: *configuration expression<string>, optional*

The AM realm where the subject is authenticated.

Default: The realm declared for `amService`.

Example

For examples that use the UserProfileFilter, see Pass profile data downstream.

More information

org.forgerock.openig.openam.UserProfileFilter

org.forgerock.openig.tools.userprofile.UserProfileService

org.forgerock.openig.openam.UserProfileContext

UserProfileContext

AM's Authorization guide

Decorators

Decorators are heap objects to extend what other objects can do. IG defines `baseURI`, `capture`, and `timer` decorators that you can use without explicitly configuring them.

For more information, refer to Decorators.

BaseUriDecorator

Overrides the scheme, host, and port of the existing request URI, rebasing the URI and so making requests relative to a new base URI. Rebasing changes only the scheme, host, and port of the request URI. Rebasing does not affect the path, query string, or fragment.

Decorator Usage

```
{
    "name": string,
    "type": "BaseUriDecorator"
}
```

A BaseUriDecorator does not have configurable properties.

IG creates a default BaseUriDecorator named baseURI at startup time in the top-level heap, so you can use baseURI as the decorator name without adding the decorator declaration

Decorated Object Usage

```
{
    "name": string,
    "type": string,
    "config": object,
    decorator name: runtime expression<url>
}
```

`"name"`*: string, required except for inline objects*

The unique name of the object, just like an object that isn't decorated.

`"type"`*: <string>, required*

The class name of the decorated object, which must be either a Filters or a Handlers.

`"config"`*: object required unless empty*

The configuration of the object, just like an object that is not decorated

decorator name: runtime expression<url>, required

The scheme, host, and port of the new base URI. The port is optional when using the defaults (80 for HTTP, 443 for HTTPS).

The value of the string must not contain underscores, and must conform to the syntax specified in RFC 3986.

Examples

Add a custom decorator to the heap named myBaseUri:

```
{
    "name": "myBaseUri",
    "type": "BaseUriDecorator"
}
```

Set a Router's base URI to `https://www.example.com:8443`:

```
{
    "name": "Router",
    "type": "Router",
    "myBaseUri": "https://www.example.com:8443/"
}
```

More information

org.forgerock.openig.decoration.baseuri.BaseUriDecorator

CaptureDecorator

Captures request and response messages in SLF4J logs, named in this format:

```
org.forgerock.openig.decoration.capture.CaptureDecorator.<decoratorName>.<decoratedObjectName>
```

If the decorated object isn't named, the object path is used in the log.

> **Important**
>
> During debugging, consider using a CaptureDecorator to capture the entity and context of requests and responses. However, increased logging consumes resources, such as disk space, and can cause performance issues. In production, reduce logging by disabling the CaptureDecorator properties `captureEntity` and `captureContext`, or setting `maxEntityLength`.

For information about using default or custom logging, refer to Manage logs.

Decorator Usage

```
{
  "name": string,
  "type": "CaptureDecorator",
  "config": {
    "captureEntity": configuration expression<boolean>,
    "captureContext": configuration expression<boolean>,
    "maxEntityLength": configuration expression<number>,
    "masks": object
  }
}
```

`"captureEntity"`: *configuration expression<boolean>, optional*

A flag for capture of the message entity:

- `true`: Capture the request and response message entity and write it to the logs. The message entity is the body of the HTTP message, which can be a JSON document, XML, HTML, image, or other information.

  When the message is binary, IG writes a `[binary entity]`.

  When streaming is enabled in admin.json, the decorator interrupts streaming for the captured request or response until the whole entity is captured.

- `false`: Don't capture the message entity.

If the `Content-Type` header is set for a request or response, the decorator uses it to decode the request or response messages, and then writes them to the logs. If the `Content-Type` header isn't set, the decorator doesn't write the request or response messages to the logs.

Default: `false`

`"captureContext"`: *configuration expression<boolean>, optional*

A flag for capture of contextual data about the handled request, such as client, session, authentication identity, authorization identity, or any other state information associated with the request:

- `true`: Capture contextual data about the handled request.

  The context is captured as JSON. The context chain is used when processing the request inside IG in the filters and handlers.

- `false`: Don't capture contextual data about the handled request.

Default: `false`

`"maxEntityLength"`: *configuration expression<number>, optional*

The maximum number of bytes that can be captured for an entity. This property is used when `captureEntity` is `true`.

If the captured entity is bigger than `maxEntityLength`, everything up to `maxEntityLength` is captured, and an `[entity truncated]` message is written in the log.

Set `maxEntityLength` to be big enough to allow capture of normal entities, but small enough to prevent excessive memory use or `OutOfMemoryError` errors. Setting `maxEntityLength` to 2 GB or more causes an exception at startup.

Default: 524 288 bytes (512 KB)

`"masks"`: *object, optional*

The configuration to mask the values of headers and attributes in the logs.

For an example, refer to Masking Values of Headers and Attributes.

```
{
  "masks": {
    "headers": [ pattern, ... ],
    "trailers": [ pattern, ... ]
    "attributes": [ pattern, ... ]
    "mask": [ configuration expression<string>, ... ]
  }
}
```

`"headers"`: *array of patterns, optional*

The case-insensitive name of one or more headers whose value to mask in the logs.

The following value masks headers called `X-OpenAM-Username`, `X-OpenAM-Password` and `x-openam-token`:

```
"headers": ["X-OpenAM-.*"]
```

Default: None

`"trailers"`: *array of patterns, optional*

The case-insensitive name of one or more trailers whose value to mask in the logs.

The following value masks trailers called `Expires`:

```
"trailers": ["Expires"]
```

Default: None

`"attributes"`: *array of patterns, optional*

The case-insensitive name of one or more attributes whose value to mask in the logs.

Default: None

`"mask"`: *configuration expression<string>, optional*

Text to replace the masked header value or attribute value in the logs.

Default: `*****`

Decorated Object Usage

```
{
    "name": string,
    "type": string,
    "config": object,
    decorator name: capture point(s)
}
```

`"name"`: *string*, *required except for inline objects*

The unique name of the decorated object.

`"type"`: *string*, *required except for inline objects, required_*

The class name of the decorated object, which must be either a Filter or a Handler. See also Filters and Handlers.

`"config"`: *object* *required unless empty*

The configuration of the decorated object, as documented in the object reference page.

decorator name: capture point(s), optional

The decorator name must match the name of the CaptureDecorator. For example, if the CaptureDecorator has `"name": "capture"`, then decorator name is `capture`.

The capture point(s) are either a single string, or an array of strings. The strings are documented here in lowercase, but are not case-sensitive:

`"all"`

Capture at all available capture points.

`"none"`

Disable capture. If `none` is configured with other capture points, `none` takes precedence.

`"request"`

Capture the request as it enters the Filter or Handler.

`"filtered_request"`

Capture the request as it leaves the Filter. Only applies to Filters.

`"response"`

Capture the response as it enters the Filter or leaves the Handler.

`"filtered_response"`

Capture the response as it leaves the Filter. Only applies to Filters.

Examples

Log the entity

The following example decorator is configured to log the entity:

```
{
    "name": "capture",
    "type": "CaptureDecorator",
    "config": {
        "captureEntity": true
    }
}
```

Do not log the entity

The following example decorator is configured not to log the entity:

```
{
  "name": "capture",
  "type": "CaptureDecorator"
}
```

Log the context

The following example decorator is configured to log the context in JSON format, excluding the request and the response:

```
{
  "name": "capture",
  "type": "CaptureDecorator",
  "config": {
    "captureContext": true
  }
}
```

Log requests and responses with the entity

The following example decorator is configured to log requests and responses with the entity, before sending the request and before returning the response:

```
{
  "heap": [
    {
      "name": "capture",
      "type": "CaptureDecorator",
      "config": {
        "captureEntity": true
      }
    },
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "capture": [
        "request",
        "response"
      ]
    }
  ],
  "handler": "ReverseProxyHandler"
}
```

Capture transformed requests and responses

The following example uses the default CaptureDecorator to capture transformed requests and responses, as they leave filters:

```
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "type": "HeaderFilter",
        "config": {
          "messageType": "REQUEST",
          "add": {
            "X-RequestHeader": [
              "Capture at filtered_request point",
              "And at filtered_response point"
            ]
          }
        }
      },
        {
          "type": "HeaderFilter",
          "config": {
            "messageType": "RESPONSE",
            "add": {
              "X-ResponseHeader": [
                "Capture at filtered_response point"
              ]
            }
          }
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "headers": {
            "Content-Type": [ "text/html; charset=UTF-8" ]
          },
          "entity": "<html><body><p>Hello world!</p></body></html>"
        }
      }
    }
  },
  "capture": [
    "filtered_request",
    "filtered_response"
  ]
}
```

Capture the context as JSON

The following example captures the context as JSON, excluding the request and response, before sending the request and before returning the response:

```
{
  "heap": [
    {
      "name": "capture",
      "type": "CaptureDecorator",
      "config": {
        "captureContext": true
      }
    },
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "capture": [
        "request",
        "response"
      ]
    }
  ],
  "handler": "ReverseProxyHandler"
}
```

Mask values of headers and attributes

This example captures the context, and then masks the value of the cookies and credentials in the logs. To try it, set up the example in Password replay from a file, replace that route with the following route, and search the route log file for the text `MASKED`:

```
{
  "heap": [{
    "name": "maskedCapture",
    "type": "CaptureDecorator",
    "config": {
      "captureContext": true,
      "masks": {
        "headers": [ "cookie*", "set-cookie*" ],
        "attributes": [ "credentials" ],
        "mask": "MASKED"
      }
    }
  }],
  "name": "02-file-masked",
  "condition": "${find(request.uri.path, '^/profile')}",
  "maskedCapture": "all",
  "handler": {
    "type": "Chain",
    "baseURI": "http://app.example.com:8081",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${find(request.uri.path, '^/profile/george') and (request.method == 'GET')}",
            "credentials": {
              "type": "FileAttributesFilter",
              "config": {
                "file": "/tmp/userfile.txt",
                "key": "email",
                "value": "george@example.com",
                "target": "${attributes.credentials}"
              }
            },
            "request": {
              "method": "POST",
              "uri": "http://app.example.com:8081/login",
              "form": {
                "username": [
                  "${attributes.credentials.username}"
                ],
                "password": [
                  "${attributes.credentials.password}"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

More information

org.forgerock.openig.decoration.capture.CaptureDecorator

TimerDecorator

Records time to process filters, handlers, and access token resolvers.

Decorator usage

```
{
    "name": string,
    "type": "TimerDecorator",
    "config": {
        "timeUnit": configuration expression<string>
    }
}
```

IG configures a default TimerDecorator named `timer`. Use `timer` as the decorator name without explicitly declaring a decorator named timer.

`"timeUnit"`: *configuration expression<string>, optional*

Unit of time used in the decorator output. The unit of time can be any unit allowed in the <duration> field.

Default: `ms`

Decorated object usage

```
{
    "name": string,
    "type": string,
    "config": object,
    decorator name: boolean
}
```

`"name"`: *string, required except for inline objects*

The unique name of the object to decorate.

`"type"`: *string, required*

The class name of the object to decorate, which must be a Filter, Handler, or the `accessTokenResolver` property of OAuth2ResourceServerFilter.

`"config"`: *object, optional*

The configuration of the object, just like an object that is not decorated.

Default: Empty

`decorator name`*: configuration expression<boolean>, required*

IG looks for the presence of the decorator name field for the TimerDecorator:

- `true`: Activate the timer
- `false`: Deactivate the TimerDecorator

Timer metrics at the Prometheus Scrape Endpoint

This section describes the timer metrics recorded at the Prometheus Scrape Endpoint. For more information about metrics, refer to Monitoring Endpoints.

When IG is set up as described in the documentation, the endpoint is http://ig.example.com:8080/openig/metrics/prometheus.

Each timer metric is labelled with the following fully qualified names:

- `decorated_object`
- `heap`
- `name` (decorator name)
- `route`
- `router`

Timer metrics at the Prometheus Scrape Endpoint

Name	Monitoring type	Description
`ig_timerdecorator_handler_elapsed_seconds`	`Summary`	Time to process the request and response in the decorated handler.
`ig_timerdecorator_filter_elapsed_seconds`	`Summary`	Time to process the request and response in the decorated filter **and** its downstream filters and handler.
`ig_timerdecorator_filter_internal_seconds`	`Summary`	Time to process the request and response in the decorated filter.
`ig_timerdecorator_filter_downstream_seconds`	`Summary`	Time to process the request and response in filters and handlers that are downstream of the decorated filter.

Timer metrics at the Common REST Monitoring Endpoint (deprecated)

This section describes the metrics recorded at the the ForgeRock Common REST Monitoring Endpoint. For more information about metrics, refer to Monitoring Endpoints.

When IG is set up as described in the documentation, the endpoint is http://ig.example.com:8080/openig/metrics/api?_queryFilter=true.

Metrics are published with an `_id` in the following pattern:

`heap.router-name.route-name.decorator-name.object`

Timer metrics at the Common REST Monitoring Endpoint

Name	Monitoring type	Description
`elapsed`	`Timer`	Time to process the request and response in the decorated handler, or in the decorated filter **and** its downstream filters and handler.
`internal`	`Timer`	Time to process the request and response in the decorated filter.
`downstream`	`Timer`	Time to process the request and response in filters and handlers that are downstream of the decorated filter.

Timer metrics in SLF4J logs

SLF4J logs are named in this format:

```
<className>.<decoratorName>.<decoratedObjectName>
```

If the decorated object is not named, the object path is used in the log.

When a route's top-level handler is decorated, the timer decorator records the elapsed time for operations traversing the whole route:

```
2018-09-04T12:16:08,994Z | INFO  | I/O dispatcher 17 | o.f.o.d.t.T.t.top-level-handler | @myroute | Elapsed time: 13
ms
```

When an individual handler in the route is decorated, the timer decorator records the elapsed time for operations traversing the handler:

```
2018-09-04T12:44:02,161Z | INFO  | http-nio-8080-exec-8 | o.f.o.d.t.T.t.StaticResponseHandler-1 | @myroute | Elapsed
time: 1 ms
```

Examples

The following example uses the default timer decorator to record the time that TokenIntrospectionAccessTokenResolver takes to process a request:

```
{
  "accessTokenResolver": {
    "name": "TokenIntrospectionAccessTokenResolver-1",
    "type": "TokenIntrospectionAccessTokenResolver",
    "config": {
      "amService": "AmService-1",
      ...
    },
    "timer": true
  }
}
```

The following example defines a customized timer decorator in the heap, and uses it to record the time that the SingleSignOnFilter takes to process a request:

```
{
  "heap": [
    {
      "name": "mytimerdecorator",
      "type": "TimerDecorator",
      "config": {
        "timeUnit": "nano"
      }
    },
    ...
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "SingleSignOnFilter",
          "config": {
            ...
          },
          "mytimerdecorator": true
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

More information

org.forgerock.openig.decoration.timer.TimerDecorator

Audit framework

IG uses the ForgeRock common audit framework to record audit events, using an implementation that is common across the ForgeRock platform.

Audit logs use timestamps in UTC format (for example, 2018-07-18T08:48:00.160Z), a unified standard that is not affected by time changes for daylight savings. The timestamps format is not configurable.

The following objects are available for auditing:

AuditService

The audit service is based on the ForgeRock common audit event framework to record `access` audit events. For information about how to record other types of audit event, refer to Record custom audit events.

By default, no routes in a configuration are audited; the NoOpAuditService object type provides an empty audit service to the top-level heap and its child routes. IG provides a default empty service based on the NoOpAuditService type. The top-level heap and child routes inherit from the setting and use a service equivalent to the following declaration:

```
{
  "name": "AuditService",
  "type": "NoOpAuditService"
}
```

Configure auditing in the following ways:

Override the NoOpAuditService for all routes in the configuration

Define an AuditService object named `AuditService` in `config.json`. No other configuration is required; all routes use the same AuditService.

Configure an audit service that can be optionally used by all routes in the configuration

Do both of the following:

- In `config.json` in the top-level heap, define an AuditService object that is **not** named `AuditService`.
- In a route, configure the Route property `auditService` to refer to the name of the declared `AuditService` heaplet.

Configure an audit service specifically for a route

Do one of the following:

- Define an AuditService object named `AuditService` in the route heap.
- In the route heap or a parent heap, define an AuditService object that is **not** named `AuditService`; configure the Route property `auditService` to refer to the name of the declared `AuditService` heaplet.
- Configure the Route property `auditService` with an inline AuditService object.

One configuration can contain multiple AuditServices.

When you define multiple AuditServices that use JsonAuditEventHandler or CsvAuditEventHandler, configure each of the event handlers with a different `logDirectory`. This prevents the AuditServices from logging to the same audit logging file.

Usage

```
{
  "name": string,
  "type": "AuditService",
  "config": {
    "config": object,
    "eventHandlers": [ object, ...],
    "topicsSchemasDirectory": configuration expression<string>
  }
}
```

Properties

`"config"`: *object, required*

Configures the audit service itself, rather than event handlers. If the configuration uses only default settings, you can omit the field instead of including an empty object as the field value.

```
{
  "config": {
    "handlerForQueries": configuration_expression<string>,
    "availableAuditEventHandlers": [configuration_expression<string>, ...],
    "caseInsensitiveFields": [configuration_expression<string>, ...],
    "filterPolicies": {
      "field": {
        "includeIf": [configuration_expression<string>, ...],
        "excludeIf": [configuration_expression<string>, ...]
      }
    }
  }
}
```

`"handlerForQueries"`: *configuration expression<string>, optional*

The name of the event handler to use when querying audit event messages over REST.

`"availableAuditEventHandlers"`: *array of configuration expression<strings>, optional*

A list of fully qualified event handler class names for event handlers available to the audit service.

`"caseInsensitiveFields"`: *array of configuration expression<strings>, optional*

A list of audit event fields to be considered as case-insensitive for filtering. The fields are referenced using JSON pointer syntax. The list can be `null` or empty.

Default: `/access/http/request/headers` and `/access/http/response/headers` fields are considered case-insensitive for filtering. All other fields are considered case-sensitive.

`"filterPolicies"`: *object, optional*

To prevent logging of sensitive data for an event, the Common Audit implementation uses a safelist to specify which event fields appear in the logs. By default, only event fields that are safelisted are included in the audit event logs. For more information about safelisting, refer to Safelisting audit event fields for the logs.

`"field"`: *object*, *optional*

This property specifies non-safelisted event fields to include in the logs, and safelisted event fields to exclude from the logs.

If `includeIf` and `excludeIf` are specified for the same field, `excludeIf` takes precedence.

Audit event fields use JSON pointer notation, and are taken from the JSON schema for the audit event content.

Default: Include only safelisted event fields in the logs.

`"includeIf"`: *array of configuration expression<strings>, optional:*

A list of non-safelisted audit event fields to include in the logs. Specify the topic and the hierarchy to the field. Any child fields of the specified field are encompassed.

> **Important**
>
> Before you include non-safelisted event fields in the logs, consider the impact on security. Including some headers, query parameters, or cookies in the logs could cause credentials or tokens to be logged, and allow anyone with access to the logs to impersonate the holder of these credentials or tokens.

`"excludeIf"`: *array of configuration expression<strings>, optional:*

A list of safelisted audit event fields to exclude from the logs. Specify the topic and the hierarchy to the field. Any child fields of the specified field are encompassed.

The following example excludes fields for the `access` topic:

```
{
  "field": {
    "excludeIf": [
      "/access/http/request/headers/host",
      "/access/http/request/path",
      "/access/server",
      "/access/response"
    ]
  }
}
```

For an example route that excludes fields, see Exclude safelisted audit event fields from logs.

`"eventHandlers"`: *array of Event Handler objects, required*

An array of one or more audit event handler configuration objects to deal with audit events.

The configuration of the event handler depends on type of event handler. IG supports the event handlers listed in AuditFramework.

`"topicsSchemasDirectory"`: *configuration expression<string>, optional*

Directory containing the JSON schema for the topic of a custom audit event. The schema defines which fields are included in the topic. For information about the syntax, see JSON Schema.

Default: `$HOME/.openig/audit-schemas` (Windows, `%appdata%\OpenIG\audit-schemas`)

For an example of how to configure custom audit events, see Record custom audit events.

The following example schema includes the mandatory fields, `_id`, `timestamp`, `transactionId`, and `eventName`, and an optional `customField`:

```
{
  "schema": {
    "$schema": "http://json-schema.org/draft-04/schema#",
    "id": "/",
    "type": "object",
    "properties": {
      "_id": {
        "type": "string"
      },
      "timestamp": {
        "type": "string"
      },
      "transactionId": {
        "type": "string"
      },
      "eventName": {
        "type": "string"
      },
      "customField": {
        "type": "string"
      }
    }
  }
}
```

Example

The following example audit service logs access event messages in a comma-separated variable file, named `/path/to/audit/logs/access.csv`:

```
{
  "name": "AuditService",
  "type": "AuditService",
  "config": {
    "config": {},
    "eventHandlers": [
      {
        "class": "org.forgerock.audit.handlers.csv.CsvAuditEventHandler",
        "config": {
          "name": "csv",
          "logDirectory": "/path/to/audit/logs",
          "topics": [
            "access"
          ]
        }
      }
    ]
  }
}
```

The following example route uses the audit service:

```
{
  "handler": "ReverseProxyHandler",
  "auditService": "AuditService"
}
```

More information

NoOpAuditService

org.forgerock.audit.AuditService

CsvAuditEventHandler

An audit event handler that responds to events by logging messages to files in comma-separated variable (CSV) format.

Declare the configuration in an audit service, as described in AuditService.

> **Important**
>
> The CSV handler does not sanitize messages when writing to CSV log files.
> Do not open CSV logs in spreadsheets or other applications that treat data as code.

Usage

```
{
  "class": "org.forgerock.audit.handlers.csv.CsvAuditEventHandler",
  "config": {
    "name": configuration expression<string>,
    "logDirectory": configuration expression<string>,
    "topics": [ configuration expression<string>, ... ],
    "enabled": configuration expression<boolean>,
    "formatting": {
      "quoteChar": configuration expression<string>,
      "delimiterChar": configuration expression<string>,
      "endOfLineSymbols": configuration expression<string>
    },
    "buffering": {
      "enabled": configuration expression<boolean>,
      "autoFlush": configuration expression<boolean>
    },
    "security": {
      "enabled": configuration expression<boolean>,
      "filename": configuration expression<string>,
      "password": configuration expression<string>,
      "signatureInterval": configuration expression<duration>
    },
    "fileRotation": {
      "rotationEnabled": configuration expression<boolean>,
      "maxFileSize": configuration expression<number>,
      "rotationFilePrefix": configuration expression<string>,
      "rotationFileSuffix": configuration expression<string>,
      "rotationInterval": configuration expression<duration>,
      "rotationTimes": [ configuration expression<duration>, ... ]
    },
    "fileRetention": {
      "maxDiskSpaceToUse": configuration expression<number>,
      "maxNumberOfHistoryFiles": configuration expression<number>,
      "minFreeSpaceRequired": configuration expression<number>
    },
    "rotationRetentionCheckInterval": configuration expression<duration>
  }
}
```

The values in this configuration object can use expressions as long as they resolve to the correct types for each field. For details about expressions, see Expressions.

Configuration

`"name"`: *configuration expression<string>, required*

The name of the event handler.

`"logDirectory"`: *configuration expression<string>, required*

The file system directory where this event handler writes log files.

When multiple AuditServices are defined in the deployment, prevent them from logging to the same audit logging file by setting different values for `logDirectory`.

`"topics"`: *array of configuration expression<strings>, required*

One or more topics that this event handler intercepts. IG can record the following audit event topics:

- `access`: Log access audit events. Access audit events occur at the system boundary, and include the arrival of the initial request and departure of the final response.

  To record `access` audit events, configure AuditService inline in a route, or in the heap.

- customTopic: Log custom audit events. To create a topic for a custom audit event, include a JSON schema for the topic in your IG configuration.

  To record custom audit events, configure AuditService in the heap, and refer to it from the route or subroutes. For an example of how to set up custom audit events, refer to Record custom audit events.

`"enabled"`: *configuration expression<boolean>, optional*

Whether this event handler is active.

Default: true

`"formatting"`: *object, optional*

Formatting settings for CSV log files.

The formatting object has the following fields:

`"quoteChar"`: *configuration expression<string>, optional*

A single character to quote CSV entries.

Default: `"`

`"delimiterChar"`: *configuration expression<string>, optional*

A single character to delimit CSV entries.

Default: `,`

`"endOfLineSymbols"`: *configuration expression<string>, optional*

A character or characters to separate a line.

Default: System-dependent line separator defined for the JVM

`"buffering"`: *object, optional*

Do not enable `buffering` when `security` is configured for tamper-evident logging.

Buffering settings for writing CSV log files. The default is for messages to be written to the log file for each event.

The buffering object has the following fields:

`"enabled"`: *configuration expression<boolean>, optional*

Whether log buffering is enabled.

Default: false

`"autoFlush"`: *configuration expression<boolean>, optional*

Whether events are automatically flushed after being written.

Default: true

`"security"`: *object, optional*

When `security` is configured for tamper-evident logging, do not enable `buffering`.

Security settings for CSV log files. These settings govern tamper-evident logging, whereby messages are signed. By default tamper-evident logging is not enabled.

The security object has the following fields:

`"enabled"`: *configuration expression<boolean>, optional*

Whether tamper-evident logging is enabled.

Default: false

Tamper-evident logging depends on a specially prepared keystore. For an example, see Recording Access Audit Events in CSV.

`"filename"`: *configuration expression<string>, required*

File system path to the keystore containing the private key for tamper-evident logging.

The keystore must be a keystore of type `JCEKS`. For an example, see Recording access audit events in CSV.

`"password"`: *configuration expression<string>, required*

The password for the keystore for tamper-evident logging.

This password is used for the keystore and for private keys. For an example, see Recording access audit events in CSV.

`"signatureInterval"`: *configuration expression<duration>, required*

The time interval after which to insert a signature in the CSV file. This duration must not be zero, and must not be unlimited.

`"fileRotation"`: *object, optional*

File rotation settings for log files.

`"rotationEnabled"`: *configuration expression<boolean>, optional*

A flag to enable rotation of log files.

Default: false.

`"maxFileSize"`: *configuration expression<number>, optional*

The maximum file size of an audit log file in bytes. A setting of 0 or less indicates that the policy is disabled.

Default: 0.

`"rotationFilePrefix"`: *configuration expression<string>, optional*

The prefix to add to a log file on rotation. This has an effect when time-based file rotation is enabled.

`"rotationFileSuffix"`: *configuration expression<string>, optional*

The suffix to add to a log file on rotation, possibly expressed in SimpleDateFormat.

This has an effect when time-based file rotation is enabled.

Default: `-yyyy.MM.dd-HH.mm.ss`, where yyyy characters are replaced with the year, MM characters are replaced with the month, dd characters are replaced with the day, HH characters are replaced with the hour (00-23), mm characters are replaced with the minute (00-60), and ss characters are replaced with the second (00-60).

`"rotationInterval"`: *configuration expression<duration>, optional*

The time interval after which to rotate log files. This duration must not be zero. This has the effect of enabling time-based file rotation.

`"rotationTimes"`: *array of configuration expression<durations>, optional*

The durations, counting from midnight, after which to rotate files.

The following example schedules rotation six and twelve hours after midnight:

```
"rotationTimes": [ "6 hours", "12 hours" ]
```

This has the effect of enabling time-based file rotation.

`"fileRetention"`: *object, optional*

File retention settings for log files.

`"maxNumberOfHistoryFiles"`: *configuration expression<number>, optional*

The maximum number of historical audit files that can be stored. If the number exceeds this maximum, older files are deleted. A value of `-1` disables purging of old log files.

Default: 0.

`"maxDiskSpaceToUse"`: *configuration expression<number>, optional*

The maximum disk space in bytes that can be used for audit files. If the audit files use more than this space, older files are deleted. A negative or zero value indicates that this policy is disabled, and historical audit files can use unlimited disk space.

Default: 0

`"minFreeSpaceRequired"`: *configuration expression<string>, optional*

The minimum free disk space in bytes required on the system that houses the audit files. If the free space drops below this minimum, older files are deleted. A negative or zero value indicates that this policy is disabled, and no minimum space requirements apply.

Default: 0

`"rotationRetentionCheckInterval"`: *configuration expression<string>, optional*

Interval at which to periodically check file rotation and retention policies. The interval must be a duration, for example, 5 seconds, 5 minutes, or 5 hours.

Default: 5 seconds

Example

For information about how to record audit events in a CSV file, see Recording Access Audit Events in CSV.

The following example configures a CSV audit event handler to write a log file, `/path/to/audit/logs/access.csv`, that is signed every 10 seconds to make it tamper-evident:

```
{
  "name": "csv",
  "topics": [
    "access"
  ],
  "logDirectory": "/path/to/audit/logs/",
  "security": {
    "enabled": "true",
    "filename": "/path/to/secrets/audit-keystore",
    "password": "password",
    "signatureInterval": "10 seconds"
  }
}
```

More information

org.forgerock.audit.handlers.csv.CsvAuditEventHandler

ElasticsearchAuditEventHandler (deprecated)

Important

This object is deprecated; use one of the following objects instead:

- SyslogAuditEventHandler
- JsonAuditEventHandler, with `elasticsearchCompatible` set to `true`

For more information, refer to the Deprecated section of the *Release Notes*.

An audit event handler that responds to events by logging messages in the Elasticsearch search and analytics engine. For information about downloading and installing Elasticsearch, refer to the Elasticsearch Getting started document.

Usage

Configure the ElasticsearchAuditEventHandler within an AuditService:

```
{
  "type": "AuditService",
  "config": {
    "config": {},
    "eventHandlers": [{
      "class": "org.forgerock.audit.handlers.elasticsearch.ElasticsearchAuditEventHandler",
      "config": {
        "name": configuration expression<string>,
        "topics": [ configuration expression<string>, ... ],
        "connection": {
          "host": configuration expression<string>,
          "port": configuration expression<number>,
          "useSSL": configuration expression<boolean>,
          "username": configuration expression<string>,
          "password": configuration expression<string>
        },
        "indexMapping": {
          "indexName": configuration expression<string>
        },
        "buffering": {
          "enabled": configuration expression<boolean>,
          "writeInterval": configuration expression<duration>,
          "maxSize": configuration expression<number>,
          "maxBatchedEvents": configuration expression<number>
        }
      }
    }
  }
}
```

The ElasticsearchAuditEventHandler relays audit events to Elasticsearch through the HTTP protocol, using a handler defined in a heap. The handler can be of any kind of handler, from a simple ClientHandler to a complex Chain, composed of multiple filters and a final handler or ScriptableHandler.

IG searches first for a handler named `ElasticsearchClientHandler`. If not found, IG searches for a client handler named `AuditClientHandler`. If not found, IG uses the route's default client handler, named `ClientHandler`.

The following example configures a ClientHandler named `ElasticsearchClientHandler`:

```
{
  "name": "ElasticsearchClientHandler",
  "type": "ClientHandler",
  "config": {}
}
```

The following example configures a ScriptableHandler named `AuditClientHandler`:

```
{
  "name": "AuditClientHandler",
  "type": "ScriptableHandler",
  "config": {}
}
```

Properties

"name": *configuration expression<string>, required*

The name of the event handler.

"topics": *array of configuration expression<strings>, required*

One or more topics that this event handler intercepts. IG can record the following audit event topics:

- `access` : Log access audit events. Access audit events occur at the system boundary, and include the arrival of the initial request and departure of the final response.

  To record `access` audit events, configure AuditService inline in a route, or in the heap.

- customTopic: Log custom audit events. To create a topic for a custom audit event, include a JSON schema for the topic in your IG configuration.

  To record custom audit events, configure AuditService in the heap, and refer to it from the route or subroutes. For an example of how to set up custom audit events, refer to Record custom audit events.

"connection": *object, optional*

Connection settings for sending messages to Elasticsearch. If this object is not configured, it takes default values for its fields. This object has the following fields:

"host": *configuration expression<string>, optional*

Hostname or IP address of Elasticsearch.

Default: `localhost`

"port": *configuration expression<number>, optional*

The port used by Elasticsearch. The value must be between 0 and 65535.

Default: `9200`

"useSSL": *configuration expression<boolean>, optional*

Setting to use or not use SSL/TLS to connect to Elasticsearch.

Default: `false`

"username": *configuration expression<string>, optional*

Username when basic authentication is enabled through Elasticsearch Shield.

`"password"`: *configuration expression<string>, optional*

Password when basic authentication is enabled through Elasticsearch Shield.

`"indexMapping"`: *object, optional*

Defines how an audit event and its fields are stored and indexed.

`"indexName"`: *configuration expression<string>, optional*

The index name. Set this parameter if the default name `audit` conflicts with an existing Elasticsearch index.

Default: `audit`.

`"buffering"`: *object, optional*

Settings for buffering events and batch writes.

`"enabled"`: *configuration expression<boolean>, optional*

Setting to use or not use log buffering.

Default: false.

`"writeInterval"`: *configuration expression<duration>*

The interval at which to send buffered event messages to Elasticsearch. If buffering is enabled, this interval must be greater than 0.

Default: 1 second

`"maxBatchedEvents"`: *configuration expression<number>, optional*

The maximum number of event messages in a batch write to Elasticsearch for each `writeInterval`.

Default: 500

`"maxSize"`: *configuration expression<number>, optional*

The maximum number of event messages in the queue of buffered event messages.

Default: 10000

Example

In the following example, an Elasticsearch audit event handler logs audit events for access. For an example of setting up and testing this configuration, refer to [maintenance-guide:].

```
{
  "name": "30-elasticsearch",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/elasticsearch-audit')}",
  "heap": [
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.elasticsearch.ElasticsearchAuditEventHandler",
            "config": {
              "name": "elasticsearch",
              "indexMapping": {
                "indexName": "audit"
              },
              "connection": {
                "host": "localhost",
                "port": 9200,
                "useSSL": false
              },
              "topics": [
                "access"
              ]
            }
          }
        ]
      }
    }
  ],
  "auditService": "AuditService",
  "handler": "ReverseProxyHandler"
}
```

More information

org.forgerock.audit.handlers.elasticsearch.ElasticsearchAuditEventHandler

JdbcAuditEventHandler

An audit event handler that responds to events by logging messages to an appropriately configured relational database table.

Declare the configuration in an audit service, as described in AuditService.

To configure IG to use the database, add the database .jar file containing the Driver as follows:

- Create the directory `$HOME/.openig/extra`, where `$HOME/.openig` is the instance directory, and add .jar files to the directory.

The JDBC handler library is in the `lib` directory.

Unpack the library, then find the examples under the `db/` folder.

Usage

```
{
  "class": "org.forgerock.audit.handlers.jdbc.JdbcAuditEventHandler",
  "config": {
    "name": configuration expression<string>,
    "topics": [ configuration expression<string>, ... ],
    "databaseType": configuration expression<string>,
    "enabled": configuration expression<boolean>,
    "buffering": {
      "enabled": configuration expression<boolean>,
      "writeInterval": configuration expression<duration>,
      "autoFlush": configuration expression<boolean>,
      "maxBatchedEvents": configuration expression<number>,
      "maxSize": configuration expression<number>,
      "writerThreads": configuration expression<number>
    },
    "connectionPool": {
      "driverClassName": configuration expression<string>,
      "dataSourceClassName": configuration expression<string>,
      "jdbcUrl": configuration expression<string>,
      "username": configuration expression<string>,
      "password": configuration expression<string>,
      "autoCommit": configuration expression<boolean>,
      "connectionTimeout": configuration expression<number>,
      "idleTimeout": configuration expression<number>,
      "maxLifetime": configuration expression<number>,
      "minIdle": configuration expression<number>,
      "maxPoolSize": configuration expression<number>,
      "poolName": configuration expression<string>
    },
    "tableMappings": [
      {
        "event": configuration expression<string>,
        "table": configuration expression<string>,
        "fieldToColumn": map or configuration expression<map>
      }
    ]
  }
}
```

Configuration

`"name"`: *configuration expression<string>, required*

The name of the event handler.

`"topics"`: *array of configuration expression<strings>, required*

One or more topics that this event handler intercepts. IG can record the following audit event topics:

- `access`: Log access audit events. Access audit events occur at the system boundary, and include the arrival of the initial request and departure of the final response.

  To record `access` audit events, configure AuditService inline in a route, or in the heap.

- customTopic: Log custom audit events. To create a topic for a custom audit event, include a JSON schema for the topic in your IG configuration.

  To record custom audit events, configure AuditService in the heap, and refer to it from the route or subroutes. For an example of how to set up custom audit events, refer to Record custom audit events.

`"databaseType"`: *configuration expression<string>, required*

The database type name.

Built-in support is provided for `oracle`, `mysql`, and `h2`.

`"enabled"`: *configuration expression<boolean>, optional*

Whether this event handler is active.

Default: true.

`"buffering"`: *object, optional*

Buffering settings for sending messages to the database. The default is for messages to be written to the log file for each event.

The buffering object has the following fields:

`"enabled"`: *configuration expression<boolean>, optional*

Whether log buffering is enabled.

Default: false.

`"writeInterval"`: *configuration expression<duration>, required*

The interval at which to send buffered event messages to the database.

This interval must be greater than 0 if buffering is enabled.

`"autoFlush"`: *configuration expression<boolean>, optional*

Whether the events are automatically flushed after being written.

Default: true.

`"maxBatchedEvents"`: *configuration expression<number>, optional*

The maximum number of event messages batched into a PreparedStatement.

Default: 100.

`"maxSize"`: : *configuration expression<number>, optional*

The maximum size of the queue of buffered event messages.

Default: 5000.

`"writerThreads"`: *configuration expression<number>, optional*

The number of threads to write buffered event messages to the database.

Default: 1.

`"connectionPool"`: *object, required*

When a JdbcDataSource object named `AuditService` is defined in the route heap. This configuration is not required.

Connection pool settings for sending messages to the database.

`"driverClassName"`: *configuration expression<string>, optional*

The class name of the driver to use for the JDBC connection. For example, with MySQL Connector/J, the class name is `com.mysql.jdbc.Driver`.

`"dataSourceClassName"`: *configuration expression<string>, optional*

The class name of the data source for the database.

`"jdbcUrl"`: *configuration expression<string>, required*

The JDBC URL to connect to the database.

`"username"`: *configuration expression<string>, required*

The username identifier for the database user with access to write the messages.

`"password"`: *configuration expression<number>, optional*

The password for the database user with access to write the messages.

`"autoCommit"`: *configuration expression<boolean>, optional*

Whether to commit transactions automatically when writing messages.

Default: true.

`"connectionTimeout"`: *configuration expression<number>, optional*

The number of milliseconds to wait for a connection from the pool before timing out.

Default: 30000.

`"idleTimeout"`: *configuration expression<number>, optional*

The number of milliseconds to allow a database connection to remain idle before timing out.

Default: 600000.

`"maxLifetime"`: *configuration expression<number>, optional*

The number of milliseconds to allow a database connection to remain in the pool.

Default: 1800000.

`"minIdle"`: *configuration expression<number>, optional*

The minimum number of idle connections in the pool.

Default: 10.

`"maxPoolSize"`: *configuration expression<number>, optional*

The maximum number of connections in the pool.

Default: 10.

`"poolName"`: *configuration expression<string>, optional*

The name of the connection pool.

`"tableMappings"`: *array of objects, required*

Table mappings for directing event content to database table columns.

A table mappings object has the following fields:

`"event"`: *configuration expression<string>, required*

The audit event that the table mapping is for.

Set this to `access`.

`"table"`: *configuration expression<string>, required*

The name of the database table that corresponds to the mapping.

`"fieldToColumn"`: *map or configuration expression<map>, required*

A map of one or more data pairs with the format `Map<String, String>`, where:

- The key is the name of an audit event field
- The value is the name of a database column, or a configuration expression that evaluates to the name of a database column

The following formats are allowed:

```
{
  "fieldToColumn": {
    "string": "configuration expression<string>",
    ...
  }
}
```

```
{
  "fieldToColumn": "configuration expression<map>"
}
```

Audit event fields use JSON pointer notation, and are taken from the JSON schema for the audit event content.

In the following example, the property is a map whose keys and values are strings representing the names of audit event fields and database columns:

```
{
  "fieldToColumn": {
    "_id": "id",
    "timestamp": "timestamp_",
    ...
  }
```

Example

Examples including statements to create tables are provided in the JDBC handler library, `forgerock-audit-handler-jdbc-version.jar`.

For an example of using JdbcAuditEventHandler, refer to Recording access audit events in a database.

In the following example, IG events are logged to an h2 database:

```
{
  "name": "audit-jdbc",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/audit-jdbc')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AuditDataSource",
      "type": "JdbcDataSource",
      "config": {
        "dataSourceClassName" : "org.h2.jdbcx.JdbcDataSource",
        "username"            : "sa",
        "passwordSecretId"    : "database.password",
        "secretsProvider"     : "SystemAndEnvSecretStore-1",
        "properties" : {
          "url"               : "jdbc:h2:tcp://localhost/~/test"
        }
      }
    },
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.jdbc.JdbcAuditEventHandler",
            "config": {
              "databaseType": "h2",
              "name": "jdbc",
              "topics": [
                "access"
              ],
              "tableMappings": [
                {
                  "event": "access",
                  "table": "audit.auditaccess",
                  "fieldToColumn": {
                    "_id": "id",
                    "timestamp": "timestamp_",
                    "eventName": "eventname",
                    "transactionId": "transactionid",
                    "userId": "userid",
                    "trackingIds": "trackingids",
                    "server/ip": "server_ip",
                    "server/port": "server_port",
                    "client/ip": "client_ip",
                    "client/port": "client_port",
                    "request/protocol": "request_protocol",
                    "request/operation": "request_operation",
                    "request/detail": "request_detail",
                    "http/request/secure": "http_request_secure",
                    "http/request/method": "http_request_method",
                    "http/request/path": "http_request_path",
                    "http/request/queryParameters": "http_request_queryparameters",
                    "http/request/headers": "http_request_headers",
                    "http/request/cookies": "http_request_cookies",
                    "http/response/headers": "http_response_headers",
```

```
                    "response/status": "response_status",
                    "response/statusCode": "response_statuscode",
                    "response/elapsedTime": "response_elapsedtime",
                    "response/elapsedTimeUnits": "response_elapsedtimeunits"
                  }
                }
              ]
            }
          }
        ]
      }
    },
    "auditService": "AuditService",
    "handler": "ReverseProxyHandler"
  }
```

More information

org.forgerock.audit.handlers.jdbc.JdbcAuditEventHandler

JmsAuditEventHandler

The Java Message Service (JMS) is a Java API for sending asynchronous messages between clients. It wraps audit events in JMS messages and publishes them in a JMS broker, which then delivers the messages to the appropriate destination.

The JMS API architecture includes a *JMS provider* and *JMS clients*, and supports the *publish/subscribe* messaging pattern. For more information, refer to Basic JMS API Concepts

The JMS audit event handler does not support queries. To support queries, also enable a second handler that supports queries.

The ForgeRock JMS audit event handler supports JMS communication, based on the following components:

- JMS message broker .jar files, to provide clients with connectivity, message storage, and message delivery functionality.

  Add the .jar files to the configuration as follows:

- Create the directory `$HOME/.openig/extra`, where `$HOME/.openig` is the instance directory, and add .jar files to the directory.

- JMS messages.

- Destinations, maintained by a message broker. A destination can be a JMS topic, using publish/subscribe to take the ForgeRock JSON for an audit event, wrap it into a JMS TextMessage, and send it to the broker.

- JMS clients, to produce and/or receive JMS messages.

Depending on the configuration, some or all of these components are included in JMS audit log messages.

> **Important**
>
> The example in this section is based on Apache ActiveMQ, but you can choose a different JMS message broker.

Declare the configuration in an audit service, as described in AuditService.

Usage

```
{
  "name": string,
  "type": "AuditService",
  "config": {
    "config": {},
    "eventHandlers": [
    {
      "class": "org.forgerock.audit.handlers.jms.JmsAuditEventHandler",
      "config": {
        "name": configuration expression<string>,
        "topics": [ configuration expression<string>, ... ],
        "deliveryMode": configuration expression<string>,
        "sessionMode": configuration expression<string>,
        "jndi": {
          "contextProperties": map,
          "topicName": configuration expression<string>,
          "connectionFactoryName": configuration expression<string>
        }
      }
    }]
  }
}
```

The values in this configuration object can use configuration expressions, as described in Configuration and Runtime Expressions.

Configuration

For a list of properties in the `"config"` object, refer to JMS Audit Event Handler in IDM's *Integrator's guide*.

`"name"`: *configuration expression<string>, required*

The name of the event handler.

`"topics"`: *array of configuration expression<strings>, required*

One or more topics that this event handler intercepts. IG can record the following audit event topics:

- `access`: Log access audit events. Access audit events occur at the system boundary, and include the arrival of the initial request and departure of the final response.

  To record `access` audit events, configure AuditService inline in a route, or in the heap.

- customTopic: Log custom audit events. To create a topic for a custom audit event, include a JSON schema for the topic in your IG configuration.

  To record custom audit events, configure AuditService in the heap, and refer to it from the route or subroutes. For an example of how to set up custom audit events, refer to Record custom audit events.

`"deliveryMode"`: *configuration expression<string>, required*

Delivery mode for messages from a JMS provider. Set to `PERSISTENT` or `NON_PERSISTENT`.

`"sessionMode"`: *configuration expression<string>, required*

Acknowledgement mode in sessions without transactions. Set to `AUTO`, `CLIENT`, or `DUPS_OK`.

`"contextProperties"`: *map, optional*

Settings with which to populate the initial context.

The map values are evaluated as configuration expression<strings>.

The following properties are required when ActiveMQ is used as the message broker:

- `java.naming.factory.initial`

  For example, `"org.apache.activemq.jndi.ActiveMQInitialContextFactory"`.

  To substitute a different JNDI message broker, change the JNDI context properties.

- `java.naming.provider.url`

  For example, `"tcp://127.0.0.1:61616"`.

  To configure the message broker on a remote system, substitute the associated IP address.

  To set up SSL, set up keystores and truststores, and change the value of the `java.naming.provider.url` to:

  ```
  ssl://127.0.0.1:61617?
  daemon=true&socket.enabledCipherSuites=SSL_RSA_WITH_RC4_128_SHA,SSL_DH_anon_WITH_3DES_EDE_CBC_SHA
  ```

- `topic.audit`

  For example, `"audit"`.

To use the JMS resources provided by your application server, leave this field empty. The values for `topicName` and `connectionFactoryName` are then JNDI names that depend on the configuration of your application server.

`"topicName"`: *configuration expression<string>, required*

JNDI lookup name for the JMS topic.

For ActiveMQ, this property must be consistent with the value of `topic.audit` in `contextProperties`.

`"connectionFactoryName"`: *configuration expression<string>, required*

JNDI lookup name for the JMS connection factory.

Example

In the following example, a JMS audit event handler delivers audit events in batches. The handler is configured to use the ActiveMQ JNDI message broker, on port 61616. For an example of setting up and testing this configuration, refer to Recording Access Audit Events in JMS.

```
{
  "name": "30-jms",
  "MyCapture" : "all",
  "baseURI": "http://app.example.com:8081",
  "condition" : "${request.uri.path == '/activemq_event_handler'}",
  "heap": [
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers" : [
          {
            "class" : "org.forgerock.audit.handlers.jms.JmsAuditEventHandler",
            "config" : {
              "name" : "jms",
              "topics": [ "access" ],
              "deliveryMode" : "NON_PERSISTENT",
              "sessionMode" : "AUTO",
              "jndi" : {
                "contextProperties" : {
                  "java.naming.factory.initial" : "org.apache.activemq.jndi.ActiveMQInitialContextFactory",
                  "java.naming.provider.url" : "tcp://am.example.com:61616",
                  "topic.audit" : "audit"
                },
                "topicName" : "audit",
                "connectionFactoryName" : "ConnectionFactory"
              }
            }
          }
        ],
        "config" : { }
      }
    }
  ],
  "auditService": "AuditService",
  "handler" : {
    "type" : "StaticResponseHandler",
    "config" : {
      "status" : 200,
      "headers" : {
        "Content-Type" : [ "text/plain; charset=UTF-8" ]
      },
      "entity" : "Message from audited route"
    }
  }
}
```

More information

org.forgerock.audit.handlers.jms.JmsAuditEventHandler

JsonAuditEventHandler

Logs events as JSON objects to a set of JSON files. There is one file for each topic defined in `topics`, named with the format `topic.audit.json`.

The JsonAuditEventHandler is the preferred file-based audit event handler.

Declare the configuration in an audit service, as described in AuditService.

Usage

```
{
  "name": string,
  "type": "AuditService",
  "config": {
    "config": {},
    "eventHandlers": [
    {
      "class": "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
      "config": {
        "name": configuration expression<string>,
        "topics": [ configuration expression<string>, ... ],
        "logDirectory": configuration expression<string>,
        "elasticsearchCompatible": configuration expression<boolean>,
        "fileRotation": {
          "rotationEnabled": configuration expression<boolean>,
          "maxFileSize": configuration expression<number>,
          "rotationFilePrefix": configuration expression<string>,
          "rotationFileSuffix": configuration expression<string>,
          "rotationInterval": configuration expression<duration>,
          "rotationTimes": [ configuration expression<duration>, ... ]
        },
        "fileRetention": {
          "maxNumberOfHistoryFiles": configuration expression<number>,
          "maxDiskSpaceToUse": configuration expression<number>,
          "minFreeSpaceRequired": configuration expression<number>
        },
        "rotationRetentionCheckInterval": configuration expression<duration>,
        "buffering": {
          "writeInterval": configuration expression<duration>,
          "maxSize": configuration expression<number>
        }
      }
    }]
  }
}
```

Configuration

`"name"`: *configuration expression<string>, required*

The event handler name. This property is used only to refer to the event handler, but is not used to name the generated log file.

`"topics"`: *array of configuration expression<strings>, required*

One or more topics that this event handler intercepts. IG can record the following audit event topics:

- `access`: Log access audit events. Access audit events occur at the system boundary, and include the arrival of the initial request and departure of the final response.

  To record `access` audit events, configure AuditService inline in a route, or in the heap.

- customTopic: Log custom audit events. To create a topic for a custom audit event, include a JSON schema for the topic in your IG configuration.

  To record custom audit events, configure AuditService in the heap, and refer to it from the route or subroutes. For an example of how to set up custom audit events, refer to Record custom audit events.

`"logDirectory"`: *configuration expression<string>, required*

The file system directory where this event handler writes log files.

When multiple AuditServices are defined in the deployment, prevent them from logging to the same audit logging file by setting different values for `logDirectory`.

`elasticsearchCompatible`: *configuration expression<boolean>, optional*

Set to `true` to enable compatibility with ElasticSearch JSON format. For more information, refer to the ElasticSearch documentation.

Default: `false`

`"fileRotation"`: *object, optional*

File rotation settings for log files.

`"rotationEnabled"`: *configuration expression<boolean>, optional*

A flag to enable rotation of log files.

Default: false.

`"maxFileSize"`: *configuration expression<number>, optional*

The maximum file size of an audit log file in bytes. A setting of 0 or less indicates that the policy is disabled.

Default: 0.

`"rotationFilePrefix"`: *configuration expression<string>, optional*

The prefix to add to a log file on rotation. This has an effect when time-based file rotation is enabled.

`"rotationFileSuffix"`: *configuration expression<string>, optional*

The suffix to add to a log file on rotation, possibly expressed in SimpleDateFormat.

This has an effect when time-based file rotation is enabled.

Default: `-yyyy.MM.dd-HH.mm.ss`, where yyyy characters are replaced with the year, MM characters are replaced with the month, dd characters are replaced with the day, HH characters are replaced with the hour (00-23), mm characters are replaced with the minute (00-60), and ss characters are replaced with the second (00-60).

`"rotationInterval"`: *configuration expression<duration>, optional*

The time interval after which to rotate log files. This duration must not be zero. This has the effect of enabling time-based file rotation.

`"rotationTimes"`: *array of configuration expression<durations>, optional*

The durations, counting from midnight, after which to rotate files.

The following example schedules rotation six and twelve hours after midnight:

```
"rotationTimes": [ "6 hours", "12 hours" ]
```

This has the effect of enabling time-based file rotation.

`"fileRetention"`: *object, optional*

File retention settings for log files.

`"maxNumberOfHistoryFiles"`: *configuration expression<number>, optional*

The maximum number of historical audit files that can be stored. If the number exceeds this maximum, older files are deleted. A value of `-1` disables purging of old log files.

Default: 0.

`"maxDiskSpaceToUse"`: *configuration expression<number>, optional*

The maximum disk space in bytes that can be used for audit files. If the audit files use more than this space, older files are deleted. A negative or zero value indicates that this policy is disabled, and historical audit files can use unlimited disk space.

Default: 0

`"minFreeSpaceRequired"`: *configuration expression<string>, optional*

The minimum free disk space in bytes required on the system that houses the audit files. If the free space drops below this minimum, older files are deleted. A negative or zero value indicates that this policy is disabled, and no minimum space requirements apply.

Default: 0

`"rotationRetentionCheckInterval"`: *configuration expression<string>, optional*

Interval at which to periodically check file rotation and retention policies. The interval must be a duration, for example, 5 seconds, 5 minutes, or 5 hours.

Default: 5 seconds

`"buffering"`: *object, optional*

Settings for buffering events and batch writes.

`"writeInterval"`: *configuration expression<duration>, optional*

The interval at which to send buffered event messages. If buffering is enabled, this interval must be greater than 0.

Default: 1 second

`"maxSize"`: *configuration expression<number>, optional*

The maximum number of event messages in the queue of buffered event messages.

Default: 10000

Example

For an example of setting up and testing this configuration, see Recording Access Audit Events in JSON.

More information

org.forgerock.audit.handlers.json.JsonAuditEventHandler

JsonStdoutAuditEventHandler

Logs events to JSON standard output (stdout).

Declare the configuration in an audit service, as described in AuditService.

Usage

```
{
  "name": string,
  "type": "AuditService",
  "config": {
    "config": {},
    "eventHandlers": [
    {
      "class": "org.forgerock.audit.handlers.json.stdout.JsonStdoutAuditEventHandler",
      "config": {
        "name": configuration expression<string>,
        "topics": [ configuration expression<string>, ... ],
        "elasticsearchCompatible": configuration expression<boolean>
      }
    }
  }
}
```

Configuration

`"name"`: *configuration expression<string>, required*

The name of the event handler.

`"topics"`: *array of configuration expression<strings>, required*

One or more topics that this event handler intercepts. IG can record the following audit event topics:

- `access`: Log access audit events. Access audit events occur at the system boundary, and include the arrival of the initial request and departure of the final response.

  To record `access` audit events, configure AuditService inline in a route, or in the heap.

- customTopic: Log custom audit events. To create a topic for a custom audit event, include a JSON schema for the topic in your IG configuration.

  To record custom audit events, configure AuditService in the heap, and refer to it from the route or subroutes. For an example of how to set up custom audit events, refer to Record custom audit events.

`elasticsearchCompatible`: *configuration expression<boolean>, optional*

Set to `true` to enable compatibility with ElasticSearch JSON format. For more information, refer to the ElasticSearch documentation.

Default: `false`

Example

In the following example, a JsonStdoutAuditEventHandler logs audit events. For an example of setting up and testing this configuration, refer to Recording access audit events to standard output.

```
{
  "name": "30-jsonstdout",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/jsonstdout-audit')}",
  "heap": [
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.json.stdout.JsonStdoutAuditEventHandler",
            "config": {
              "name": "jsonstdout",
              "elasticsearchCompatible": false,
              "topics": [
                "access"
              ]
            }
          }
        ],
        "config": {}
      }
    }
  ],
  "auditService": "AuditService",
  "handler": "ReverseProxyHandler"
}
```

More information

org.forgerock.audit.handlers.json.stdout.JsonStdoutAuditEventHandler

NoOpAuditService

Provides an empty audit service to the top-level heap and its child routes. Use NoOpAuditService to prevent routes from using the parent audit service, when an AuditService is not explicitly defined.

For information about how to override the default audit service, refer to Audit framework.

Usage

```
{
  "name": "AuditService",
  "type": "NoOpAuditService"
 }
```

```
"auditService": "NoOpAuditService"
```

More information

AuditService

org.forgerock.openig.audit.NoOpAuditService

SyslogAuditEventHandler

An audit event handler that responds to events by logging messages to the UNIX system log as governed by RFC 5424, The Syslog Protocol.

Declare the configuration in an audit service, as described in AuditService.

Usage

```
{
  "class": "org.forgerock.audit.handlers.syslog.SyslogAuditEventHandler",
  "config": {
    "name": configuration expression<string>,
    "topics": [ configuration expression<string>, ... ],
    "protocol": configuration expression<string>,
    "host": configuration expression<string>,
    "port": configuration expression<number>,
    "connectTimeout": configuration expression<number>,
    "facility": configuration expression<string>,
    "buffering": {
        "enabled": configuration expression<boolean>,
        "maxSize": configuration expression<number>
    },
    "severityFieldMappings": [
      {
        "topic": configuration expression<string>,
        "field": configuration expression<string>,
        "valueMappings": {
          "field-value": object
        }
      }
    ]
  }
}
```

The values in this configuration object can use expressions as long as they resolve to the correct types for each field. For details about expressions, refer to Expressions.

Configuration

`"name"`: *configuration expression<string>, required*

The name of the event handler.

`"topics"`: *array of configuration expression<strings>, required*

One or more topics that this event handler intercepts. IG can record the following audit event topics:

- `access`: Log access audit events. Access audit events occur at the system boundary, and include the arrival of the initial request and departure of the final response.

  To record `access` audit events, configure AuditService inline in a route, or in the heap.

- customTopic: Log custom audit events. To create a topic for a custom audit event, include a JSON schema for the topic in your IG configuration.

  To record custom audit events, configure AuditService in the heap, and refer to it from the route or subroutes. For an example of how to set up custom audit events, refer to Record custom audit events.

`"protocol"`: *configuration expression<string>, required*

The transport protocol used to send event messages to the Syslog daemon.

Set this to `TCP` for Transmission Control Protocol, or to `UDP` for User Datagram Protocol.

`"host"`: *configuration expression<string>, required*

The hostname of the Syslog daemon to which to send event messages. The hostname must resolve to an IP address.

`"port"`: *configuration expression<number>, required*

The port of the Syslog daemon to which to send event messages.

The value must be between 0 and 65535.

`"connectTimeout"`: *configuration expression<number>, required when using TCP*

The number of milliseconds to wait for a connection before timing out.

`"facility"`: *configuration expression<enumeration>, required*

The Syslog facility to use for event messages. Set to one of the following values:

- `kern`: Kernel messages
- `user`: User-level messages
- `mail`: Mail system
- `daemon`: System daemons
- `auth`: Security/authorization messages

- `syslog` : Messages generated internally by `syslogd`
- `lpr` : Line printer subsystem
- `news` : Network news subsystem
- `uucp` : UUCP subsystem
- `cron` : Clock daemon
- `authpriv` : Security/authorization messages
- `ftp` : FTP daemon
- `ntp` : NTP subsystem
- `logaudit` : Log audit
- `logalert` : Log alert
- `clockd` : Clock daemon
- `local0` : Local use 0
- `local1` : Local use 1
- `local2` : Local use 2
- `local3` : Local use 3
- `local4` : Local use 4
- `local5` : Local use 5
- `local6` : Local use 6
- `local7` : Local use 7

`"buffering"`: *object*, *optional*

Buffering settings for writing to the system log facility. The default is for messages to be written to the log for each event.

`"enabled"`: *configuration expression*<*boolean*>, *optional*

Whether log buffering is enabled.

Default: false.

`"maxSize"`: *configuration expression*<*number*>, *optional*

The maximum number of buffered event messages.

Default: 5000.

`"severityFieldMappings"`: *object*, *optional*

Severity field mappings set the correspondence between audit event fields and Syslog severity values.

The severity field mappings object has the following fields:

`"topic"`: *configuration expression<string>, required*

The audit event topic to which the mapping applies.

Set this to a value configured in `topics`.

`"field"`: *configuration expression<string>, required*

The audit event field to which the mapping applies.

Audit event fields use JSON pointer notation, and are taken from the JSON schema for the audit event content.

`"valueMappings"`: *object, required*

The map of audit event values to Syslog severities, where both the keys and the values are strings.

Syslog severities are one of the following values:

- `emergency`: System is unusable.
- `alert`: Action must be taken immediately.
- `critical`: Critical conditions.
- `error`: Error conditions.
- `warning`: Warning conditions.
- `notice`: Normal but significant condition.
- `informational`: Informational messages.
- `debug`: Debug-level messages.

Example

The following example configures a Syslog audit event handler that writes to the system log daemon on `syslogd.example.com`, port `6514` over TCP with a timeout of 30 seconds. The facility is the first one for local use, and response status is mapped to Syslog informational messages:

```
{
  "class": "org.forgerock.audit.handlers.syslog.SyslogAuditEventHandler",
  "config": {
    "name": "MySyslogAuditEventHandler",
    "topics": ["access"],
    "protocol": "TCP",
    "host": "https://syslogd.example.com",
    "port": 6514,
    "connectTimeout": 30000,
    "facility": "local0",
    "severityFieldMappings": [
      {
        "topic": "access",
        "field": "response/status",
        "valueMappings": {
          "FAILED": "INFORMATIONAL",
          "SUCCESSFUL": "INFORMATIONAL"
        }
      }
    ]
  }
}
```

More information

org.forgerock.audit.handlers.syslog.SyslogAuditEventHandler

SplunkAuditEventHandler (deprecated)

> **Important**
>
> This object is deprecated; use SyslogAuditEventHandler or JsonAuditEventHandler instead. For more information, refer to the Deprecated section of the *Release Notes*.

The Splunk audit event handler logs IG events to a Splunk system.

For an example of setting up and testing Splunk, see Recording access audit events in Splunk.

Usage

Configure the SplunkAuditEventHandler within an AuditService:

```
{
  "type": "AuditService",
  "config": {
    "config": {},
    "eventHandlers": [{
      "class": "org.forgerock.audit.handlers.splunk.SplunkAuditEventHandler",
      "config": {
        "name": configuration expression<string>,
        "topics": [ configuration expression<string>, ... ],
        "enabled": configuration expression<boolean>,
        "connection": {
          "useSSL": configuration expression<boolean>,
          "host": configuration expression<string>,
          "port": configuration expression<number>
        },
        "buffering": {
          "maxSize": configuration expression<number>,
          "writeInterval": configuration expression<duration>,
          "maxBatchedEvents": configuration expression<number>
        },
        "authzToken": configuration expression<string>
      }
    }]
  }
}
```

The SplunkAuditEventHandler relays audit events to Splunk through the HTTP protocol, using a handler defined in a heap. The handler can be of any kind of handler, from a simple ClientHandler to a complex Chain, composed of multiple filters and a final handler or ScriptableHandler.

IG searches first for a handler named `SplunkAuditEventHandler`. If not found, IG searches for a client handler named `AuditClientHandler`. If not found, IG uses the route's default client handler, named `ClientHandler`.

The following example configures a ClientHandler named `SplunkClientHandler`:

```
{
  "name": "SplunkClientHandler",
  "type": "ClientHandler",
  "config": {}
}
```

The following example configures a ScriptableHandler named `AuditClientHandler`:

```
{
  "name": "AuditClientHandler",
  "type": "ScriptableHandler",
  "config": {}
}
```

Configuration

`"name"`: *configuration expression<string>, required*

The name of the event handler.

`"topics"`: *array of configuration expression<strings>, required*

One or more topics that this event handler intercepts. IG can record the following audit event topics:

- `access`: Log access audit events. Access audit events occur at the system boundary, and include the arrival of the initial request and departure of the final response.

  To record `access` audit events, configure AuditService inline in a route, or in the heap.

- customTopic: Log custom audit events. To create a topic for a custom audit event, include a JSON schema for the topic in your IG configuration.

  To record custom audit events, configure AuditService in the heap, and refer to it from the route or subroutes. For an example of how to set up custom audit events, refer to Record custom audit events.

`"enabled"`: *configuration expression<boolean>, required*

Specifies whether this audit event handler is enabled.

`"connection"`: *object, optional*

Connection settings for sending messages to the Splunk system. If this object is not configured, it takes default values for its fields. This object has the following fields:

`"useSSL"`: *configuration expression<boolean>, optional*

Specifies whether IG should connect to the audit event handler instance over SSL.

Default: `false`

`"host"`: *configuration expression<string>, optional*

Hostname or IP address of the Splunk system.

Default: `localhost`

`"port"`: *configuration expression<number>, optional*

The dedicated Splunk port for HTTP input.

Before you install Splunk, make sure this port is free. Otherwise, change the port number in Splunk and in the IG routes that use Splunk.

Default: `8088`

`"buffering"`: *object, optional*

Settings for buffering events and batch writes. If this object is not configured, it takes default values for its fields. This object has the following fields:

`"maxSize"`: *configuration expression<number>, optional*

The maximum number of event messages in the queue of buffered event messages.

Default: 10000

`"maxBatchedEvents"`: *configuration expression<number>, optional*

The maximum number of event messages in a batch write to this event handler for each `writeInterval`.

Default: 500

`"writeInterval"`: *configuration expression<duration>, optional*

The delay after which the writer thread is scheduled to run after encountering an empty event buffer.

Default: 100 ms (units of 'ms' or 's' are recommended)

`"authzToken"`: *configuration expression<string>, required*

The authorization token associated with the configured HTTP event collector.

Example

In the following example, IG events are logged to a Splunk system.

```
{
  "name": "30-splunk",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/splunk-audit')}",
  "heap": [
    {
      "name": "AuditService",
      "type": "AuditService",
      "config": {
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.splunk.SplunkAuditEventHandler",
            "config": {
              "name": "splunk",
              "enabled": true,
              "authzToken": "<splunk-authorization-token>",
              "connection": {
                "host": "localhost",
                "port": 8088,
                "useSSL": false
              },
              "topics": [
                "access"
              ],
              "buffering": {
                "maxSize": 10000,
                "maxBatchedEvents": 500,
                "writeInterval": "100 ms"
              }
            }
          }
        ]
      }
    }
  ],
  "auditService": "AuditService",
  "handler": "ReverseProxyHandler"
}
```

For an example of setting up and testing this configuration, see Recording Access Audit Events in Splunk.

More information

org.forgerock.audit.handlers.splunk.SplunkAuditEventHandler

Monitoring

The following sections describe monitoring endpoints exposed by IG, and the metrics available at the endpoints.

For information about how to set up and maintain monitoring, refer to Monitor services.

Vert.x Metrics

Vert.x metrics for HTTP clients, TCP clients, and servers are available by default at the Prometheus Scrape Endpoint and Common REST Monitoring Endpoint (deprecated) endpoints. Vert.x metrics provide low-level information about requests and responses, such as the number of bytes, duration, the number of concurrent requests. The available metrics are based on those described in Vert.x core tools metrics.

For more information about Vert.x and IG, refer to the `vertx` object in AdminHttpApplication (`admin.json`), and Monitoring Vert.x Metrics.

Monitoring types

This section describes the data types used in monitoring:

Counter

Cumulative metric for a numerical value that only increases.

Gauge

Metric for a numerical value that can increase or decrease.

Summary

Metric that samples observations, providing a count of observations, sum total of observed amounts, average rate of events, and moving average rates across a sliding time window.

The Prometheus view does not provide time-based statistics, as rates can be calculated from the time-series data. Instead, the Prometheus view includes summary metrics whose names have the following suffixes or labels:

- `_count` : number of events recorded
- `_total` : sum of the amounts of events recorded
- `{quantile="0.5"}` : 50% at or below this value
- `{quantile="0.75"}` : 75% at or below this value
- `{quantile="0.95"}` : 95% at or below this value
- `{quantile="0.98"}` : 98% at or below this value
- `{quantile="0.99"}` : 99% at or below this value
- `{quantile="0.999"}` : 99.9% at or below this value

Timer

Metric combining time-series summary statistics.

Common REST views show summaries as JSON objects. JSON summaries have the following fields:

```
{
 "count": number,           // events recorded for this metric
 "max": number,             // maximum duration recorded
 "mean": number,            // total/count, or 0 if count is 0
 "min": number,             // minimum duration recorded for this metric
 "mean_rate": number,       // average rate
 "p50": number,             // 50% at or below this value
 "p75": number,             // 75% at or below this value
 "p95": number,             // 95% at or below this value
 "p98": number,             // 98% at or below this value
 "p99": number,             // 99% at or below this value
 "p999": number,            // 99.9% at or below this value
 "stddev": number,          // standard deviation of recorded durations
 "m15_rate": number,        // fifteen-minute average rate
 "m5_rate": number,         // five-minute average rate
 "m1_rate": number,         // one-minute average rate
 "duration_units": string,  // time unit used in durations
 "rate_units": string,      // event count unit and time unit used in rate
 "total": number            // sum of the durations of events recorded
}
```

Monitoring Endpoints

This section describes the monitoring endpoints exposed in IG.

Prometheus Scrape Endpoint

All ForgeRock products automatically expose a monitoring endpoint where Prometheus can scrape metrics, in a standard Prometheus format. For information about configuring Prometheus to scrape metrics, refer to the Prometheus website.

When IG is set up as described in the documentation, the endpoint is http://ig.example.com:8080/openig/metrics/prometheus.

For information about available metrics, refer to:

- Route metrics at the Prometheus Scrape Endpoint
- Router metrics at the Prometheus Scrape Endpoint
- Cache metrics at the Prometheus Scrape Endpoint
- Timer metrics At the Prometheus Scrape Endpoint

For an example that queries the Prometheus Scrape Endpoint, refer to Monitor the Prometheus Scrape Endpoint.

Common REST Monitoring Endpoint (deprecated)

{forgerock_name} products expose a monitoring endpoint where metrics are exposed as a JSON format monitoring resource.

When IG is set up as described in the documentation, the endpoint is `http://ig.example.com:8080/openig/metrics/api?_queryFilter=true`.

For information about available metrics, refer to:

- Route metrics at the Common REST Monitoring Endpoint

- Router metrics at the Common REST Monitoring Endpoint
- Cache metrics at the Common REST Monitoring Endpoint
- Timer metrics at the Common REST Monitoring Endpoint

For an example that queries the Common REST Monitoring Endpoint, refer to Monitor the Common REST Monitoring Endpoint.

Throttling policies

To protect applications from being overused by clients, use a ThrottlingFilter with one of the following policies to limit how many requests clients can make in a defined time:

MappedThrottlingPolicy

Maps different throttling rates to different groups of requests, according to the evaluation of `throttlingRateMapper`.

Usage

```
{
    "name": string,
    "type": "ThrottlingFilter",
    "config": {
        "requestGroupingPolicy": runtime expression<string>,
        "throttlingRatePolicy": {
            "type": "MappedThrottlingPolicy",
            "config": {
                "throttlingRateMapper": runtime expression<string>,
                "throttlingRatesMapping": {
                    "mapping1": {
                        "numberOfRequests": configuration expression<number>,
                        "duration": configuration expression<duration>
                    },
                    "mapping2": {
                        "numberOfRequests": configuration expression<number>,
                        "duration": configuration expression<duration>
                    }
                },
                "defaultRate": {
                    "numberOfRequests": configuration expression<number>,
                    "duration": configuration expression<duration>
                }
            }
        }
    }
}
```

Properties

`"throttlingRateMapper"`*: runtime expression<string>, required*

An expression to categorize requests for mapping to a throttling rate in the `throttlingRatesMapping`.

If this parameter is null or does not match any specified mappings, the default throttling rate is applied.

`"throttlingRatesMapping"`: *object*, *required*

A map of throttling rate by request group. Requests are categorized into groups by the evaluation of the expression `"throttlingRateMapper"`.

`"mapping1"` *and* `"mapping2"`: *string, required*

The evaluation of the expression `"throttlingRateMapper"`.

The number of mappings is not limited to two.

`"numberOfRequests"`: *configuration expression<integer>, required*

The number of requests allowed through the filter in the time specified by `"duration"`.

`"duration"`: *configuration expression<duration>, required*

A time interval during which the number of requests passing through the filter is counted.

`"defaultRate"`: *object*, *required*

The default throttling rate to apply if the evaluation of the expression `"throttlingRateMapper"` is null or is not mapped to a throttling rate.

`"numberOfRequests"`: *configuration expression<integer>, required*

The number of requests allowed through the filter in the time specified by `"duration"`.

`"duration"`: *configuration expression<duration>, required*

A time interval during which the number of requests passing through the filter is counted.

Example of a Mapped Throttling Policy

In the following example, requests from users with different statuses are mapped to different throttling rates. For information about how to set up and test this example, see Configure Mapped Throttling.

```
{
  "name": "00-throttle-mapped",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/throttle-mapped')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "token-resolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
```

```
          },
          {
            "name": "ThrottlingFilter-1",
            "type": "ThrottlingFilter",
            "config": {
              "requestGroupingPolicy": "${contexts.oauth2.accessToken.info.mail}",
              "throttlingRatePolicy": {
                "name": "MappedPolicy",
                "type": "MappedThrottlingPolicy",
                "config": {
                  "throttlingRateMapper": "${contexts.oauth2.accessToken.info.status}",
                  "throttlingRatesMapping": {
                    "gold": {
                      "numberOfRequests": 6,
                      "duration": "10 s"
                    },
                    "silver": {
                      "numberOfRequests": 3,
                      "duration": "10 s"
                    },
                    "bronze": {
                      "numberOfRequests": 1,
                      "duration": "10 s"
                    }
                  },
                  "defaultRate": {
                    "numberOfRequests": 1,
                    "duration": "10 s"
                  }
                }
              }
            }
          }
        ],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

More information

org.forgerock.openig.filter.throttling.MappedThrottlingPolicyHeaplet

ScriptableThrottlingPolicy

Uses a script to look up the throttling rates to apply to groups of requests.

The script can store the mapping for the throttling rate in memory, and can use a more complex mapping mechanism than that used in the `MappedThrottlingPolicy`. For example, the script can map the throttling rate for a range of IP addresses. The script can also query an external database or read the mapping from a file.

Scripts must return a Promise<ThrottlingRate, Exception> or a ThrottlingRate.

For information about script properties, available global objects, and automatically imported classes, refer to Scripts.

- For an example of how to create a ScriptableThrottlingPolicy in Studio, refer to Configure scriptable throttling.

Usage

```
{
  "type": "ThrottlingFilter",
  "config": {
    "requestGroupingPolicy": runtime expression<string>,
    "throttlingRatePolicy": {
      "name": string,
      "type": "ScriptableThrottlingPolicy",
      "config": {
        "type": configuration expression<string>,
        "file": configuration expression<string>, // Use either "file"
        "source": [ string, ... ],                // or "source", but not both
        "args": map,
        "clientHandler": Handler reference
        }
    }
  }
}
```

Properties

For information about properties for ScriptableThrottlingPolicy, refer to Scripts.

Example of a scriptable throttling policy

In the following example, the `DefaultRateThrottlingPolicy` delegates the management of throttling to the scriptable throttling policy. For information about how to set up and test this example, refer to Configure scriptable throttling.

```
{
  "name": "00-throttle-scriptable",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/throttle-scriptable')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam/"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "token-resolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
              }
            }
          }
```

```
        },
        {
          "name": "ThrottlingFilter-1",
          "type": "ThrottlingFilter",
          "config": {
            "requestGroupingPolicy": "${contexts.oauth2.accessToken.info.mail}",
            "throttlingRatePolicy": {
              "type": "DefaultRateThrottlingPolicy",
              "config": {
                "delegateThrottlingRatePolicy": {
                  "name": "ScriptedPolicy",
                  "type": "ScriptableThrottlingPolicy",
                  "config": {
                    "type": "application/x-groovy",
                    "source": [
                      "if (contexts.oauth2.accessToken.info.status == status) {",
                      "  return new ThrottlingRate(rate, duration)",
                      "} else {",
                      "  return null",
                      "}"
                    ],
                    "args": {
                      "status": "gold",
                      "rate": 6,
                      "duration": "10 seconds"
                    }
                  }
                },
                "defaultRate": {
                  "numberOfRequests": 1,
                  "duration": "10 s"
                }
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

More information

org.forgerock.openig.filter.throttling.ScriptableThrottlingPolicy.Heaplet

DefaultRateThrottlingPolicy

Provides a default throttling rate if the delegating throttling policy returns `null`.

Usage

```
{
   "name": string,
   "type": "ThrottlingFilter",
   "config": {
       "requestGroupingPolicy": expression,
       "throttlingRatePolicy": {
           "type": "DefaultRateThrottlingPolicy",
           "config": {
               "delegateThrottlingRatePolicy": ThrottlingRatePolicy reference,
               "defaultRate": {
                   "numberOfRequests": configuration expression<number>,
                   "duration": configuration expression<duration>
               }
           }
       }
   }
}
```

Properties

`"delegateThrottlingRatePolicy"`: *ThrottlingRatePolicy reference, required*

The policy to which the default policy delegates the throttling rate. The `DefaultRateThrottlingPolicy` delegates management of throttling to the policy specified by `delegateThrottlingRatePolicy`.

If `delegateThrottlingRatePolicy` returns `null`, the `defaultRate` is used.

For information about policies to use, refer to MappedThrottlingPolicy and ScriptableThrottlingPolicy.

`"defaultRate"`: *object, required*

The default throttling rate to apply if the delegating policy returns `null`.

`"numberOfRequests"`: *configuration expression<integer>, required*

The number of requests allowed through the filter in the time specified by `"duration"`.

`"duration"`: *configuration expression<duration>, required*

A time interval during which the number of requests passing through the filter is counted.

Example

For an example of how this policy is used, refer to Example of a Scriptable Throttling Policy.

More information

org.forgerock.openig.filter.throttling.DefaultRateThrottlingPolicyHeaplet

Miscellaneous configuration objects

The following objects can be defined in the configuration:

AmService

Holds information about the configuration of an instance of AM. The AmService is available to IG filters that communicate with that instance.

When IG uses an AmService, IG is positioned as the client of the service. By default, IG is subscribed to Websocket notifications from AM, and the WebSocket connection can be secured by ClientTlsOptions.

Usage

```
{
  "name": string,
  "type": "AmService",
  "config": {
    "agent": object,
    "secretsProvider": SecretsProvider reference,
    "notifications": object,
    "realm": configuration expression<string>,
    "amHandler": Handler reference,
    "sessionCache": object,
    "sessionIdleRefresh": object,
    "sessionProperties": [ configuration expression<string>, ... ],
    "ssoTokenHeader": configuration expression<string>,
    "url": configuration expression<url>,
    "version": configuration expression<string>
  }
}
```

Properties

`"agent"`: *object*, *required*

An IG agent profile. When the agent is authenticated, the token can be used for tasks such as getting the user's profile, making policy evaluations, and connecting to the AM notification endpoint.

```
{
  "AmService": {
     "username": configuration expression<string>,
     "passwordSecretId": configuration expression<secret-id>
   }
}
```

`"username"`: *configuration expression*<*string*>, *required*

Name of the AM agent profile.

`"passwordSecretId"`: *configuration expression<secret-id>, required*

The secret ID of the AM agent password. This secret ID must point to a GenericSecret.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for the agent password.

`"realm"`: *configuration expression<string>, optional*

The AM realm in which the IG agent is created.

Default: `/` (top level realm).

`"amHandler"`: *Handler reference, optional*

The Handler to use for communicating with AM. In production, use a ClientHandler that is capable of making an HTTPS connection to AM.

AmService does not use `amHandler` to subscribe to WebSocket notifications from AM. To subscribe to WebSocket notifications from AM, configure a ClientTlsOptions object in the heap, and refer to it from the `amHandler` object and the `notifications` subproperty `tls` .

To facilitate auditing, configure this handler with a `ForgeRockClientHandler`, which sends a ForgeRock Common Audit transaction ID when it communicates with protected applications.

Alternatively, configure this handler as a chain containing a `TransactionIdOutboundFilter`, as in the following configuration:

```
"amHandler": {
  "type": "Chain",
  "config": {
    "handler": "MySecureClientHandler",
    "filters": [ "TransactionIdOutboundFilter" ]
  }
}
```

Default: `ForgeRockClientHandler`

See also Handlers and ClientHandler.

`"notifications"`: *object, optional*

Configure a WebSocket notification service to subscribe to Websocket notifications from AM.

To subscribe to WebSocket notifications from AM, configure a ClientTlsOptions object in the heap, and refer to it from the `amHandler` object and the `notifications` subproperty `tls`. Alternatively, use `proxyOptions` to share a proxy configuration between the `amHandler` and the notification service.

For information, refer to WebSocket notifications.

```
{
  "notifications": {
    "enabled": configuration expression<boolean>,
    "initialConnectionAttempts": configuration expression<number>,
    "reconnectDelay": configuration expression<duration>,
    "renewalDelay": configuration expression<duration>,
    "heartbeatInterval": configuration expression<duration>,
    "connectionTimeout": configuration expression<duration>,
    "idleTimeout":  configuration expression<duration>,
    "tls": ClientTlsOptions reference,
    "proxyOptions": ProxyOptions reference,
    "vertx": object
  }
}
```

`enabled`: *configuration expression<boolean>, optional*

A flag to enable WebSocket notifications. Set to `false` to disable WebSocket notifications.

Default: `true`

`initialConnectionAttempts`: *configuration expression<number>, optional*

The maximum number of times IG attempts to open a WebSocket connection before failing to deploy a route. For no limit, set this property to `-1`.

If the WebSocket connection fails **after** it has been opened and the route is deployed, IG attempts to reconnect to it an unlimited number of times.

Default: `5`

`reconnectDelay`: *configuration expression<duration>, optional*

The time between attempts to re-establish a lost WebSocket connection.

When a WebSocket connection is lost, IG waits for this delay and then attempts to re-establish the connection. If subsequent attempts fail, IG waits and tries again an unlimited number of times.

Default: `5 seconds`

`renewalDelay`: *configuration expression<duration>, optional*

The time before automatically renewing a WebSocket connection between IG and AM. IG renews connections transparently.

ForgeRock Identity Cloud closes WebSocket connections every 60 minutes. This property is set by default to prevent connection closure by automatically renewing connections every 50 minutes.

Set to `0` or `unlimited` to never automatically renew connections.

Default: `50 minutes`

`heartbeatInterval`: *configuration expression<duration>, optional*

The interval at which the AmService issues a heartbeat on WebSocket connections. When activity on the connection is low, the heartbeat prevents middleware or policies situated between IG and AM from closing the connection for timeout.

Set to zero or unlimited to disable heartbeats.

Default: `1 minute`

`connectionTimeout`: *configuration expression<duration>, optional*

The time IG waits to establish a Websocket connection to AM before it considers the attempt as failed.

Default: `60 seconds`

`idleTimeout`: *configuration expression<duration>, optional*

The time that a WebSocket connection to AM can be inactive before IG closes it.

Default: `unlimited`

`tls`: *ClientTlsOptions reference, optional*

Configure options for WebSocket connections to TLS-protected endpoints. Define a ClientTlsOptions object inline or in the heap.

Default: Connections to TLS-protected endpoints are not configured.

`proxyOptions`: *ProxyOptions reference>, optional*

A proxy server to which requests can be submitted. Use this property to relay requests to other parts of the network. For example, use it to submit requests from an internal network to the internet.

Provide the name of a ProxyOptions object defined in the heap or an inline configuration.

Default: A heap object named `ProxyOptions`.

`vertx`: *object, optional*

Vert.x-specific configuration for WebSocket connections to AM. Vert.x values are evaluated as configuration expressions.

Use the Vert.x options described in VertxOptions.

`"url"`: *configuration expression<url>, required*

The URL of the AM service. When AM is running locally, this value could be `https://am.example.com/openam`. When AM is running in the ForgeRock Identity Cloud, this value could be `https://myTenant.forgeblocks.com/am`.

`"sessionCache"`: *object, optional*

In AM, if the realm includes a customized session property safelist, include `AMCtxId` in the list of properties. The customized session property safelist overrides the global session property safelist.

Enable and configure caching of session information from AM, based on *Caffeine*. For more information, see the GitHub entry, Caffeine.

When `sessionCache` is enabled, IG can reuse session token information without repeatedly asking AM to verify the token. Each instance of AmService has an independent cache content. The cache is not shared with other AmService instances, either in the same or different routes, and is not distributed among clustered IG instances.

When `sessionCache` is disabled, IG must ask AM to verify the token for each request.

IG evicts session info entries from the cache for the following reasons:

- AM cache timeout, based the whichever of the following events occur first:
  - `maxSessionExpirationTime` from SessionInfo
  - `maxSessionTimeout` from the AmService configuration

    When IG evicts session info entries from the cache, the next time the token is presented, IG must ask AM to verify the token.
- If Websocket notifications are enabled, AM session revocation, for example, when a user logs out of AM.

  When Websocket notifications are enabled, IG evicts a cached token almost as soon as it is revoked on AM, and in this way stays synchronized with AM. Subsequent requests to IG that present the revoked token are rejected.

  When Websocket notifications are disabled, the token remains in the cache after it is revoked on AM. Subsequent requests to IG that present the revoked token are considered as valid, and can cause incorrect authentication and authorization decisions until its natural eviction from the cache.

```
{
  "sessionCache": {
    "enabled": configuration expression<boolean>,
    "executor": Executor service reference,
    "maximumSize": configuration expression<number>,
    "maximumTimeToCache": configuration expression<duration>,
    "onNotificationDisconnection": configuration expression<enumeration>
  }
}
```

`enabled`*: configuration expression<boolean>, optional*

Enable caching.

Default: `false`

`executor`*: Executor service reference, optional*

An executor service to schedule the execution of tasks, such as the eviction of entries in the cache.

Default: `ForkJoinPool.commonPool()`

`"maximumSize"`*: configuration expression<number>, optional*

The maximum number of entries the cache can contain.

Default: Unlimited/unbound.

`maximumTimeToCache`: *configuration expression<duration>, optional*

The maximum duration for which to cache session info. Consider setting this duration to be less than the idle timeout of AM.

If `maximumTimeToCache` is longer than `maxSessionExpirationTime` from SessionInfo, `maxSessionExpirationTime` is used.

Default:

- When `sessionIdleRefresh` is set, idle timeout of AM minus 30 seconds.
- When `sessionIdleRefresh` is not set, `maxSessionExpirationTime`, from SessionInfo.

`onNotificationDisconnection`: *configuration expression<enumeration>, optional*

The strategy to manage the cache when the WebSocket notification service is disconnected, and IG receives no notifications for AM events. If the cache is not cleared it can become outdated, and IG can allow requests on revoked sessions or tokens.

Cached entries that expire naturally while the notification service is disconnected are removed from the cache.

Use one of the following values:

- `NEVER_CLEAR`
  - When the notification service is disconnected:
    - Continue to use the existing cache.
    - Deny access for requests that are not cached, but do not update the cache with these requests.
  - When the notification service is reconnected:
    - Continue to use the existing cache.
    - Query AM for incoming requests that are not found in the cache, and update the cache with these requests.
- `CLEAR_ON_DISCONNECT`
  - When the notification service is disconnected:
    - Clear the cache.
    - Deny access to all requests, but do not update the cache with these requests.
  - When the notification service is reconnected:
    - Query AM for all requests that are not found in the cache. (Because the cache was cleared, the cache is empty after reconnection.)
    - Update the cache with these requests.

- `CLEAR_ON_RECONNECT`
  - When the notification service is disconnected:
    - Continue to use the existing cache.
    - Deny access for requests that are not cached, but do not update the cache with these requests.
  - When the notification service is reconnected:
    - Query AM for all requests that are not found in the cache. (Because the cache was cleared, the cache is empty after reconnection.)
    - Update the cache with these requests.

Default: `CLEAR_ON_DISCONNECT`

`"sessionIdleRefresh"`: *object*, *optional*

Enable and configure periodic refresh of idle sessions. When this property is enabled, IG requests session refresh:

- The first time IG gets an SSO token from AM, irrespective of the age of the token
- When `sessionIdleRefresh.interval` has elapsed

Use this property when AM is using CTS-based sessions. AM does not monitor idle time for client-side sessions, and so refresh requests are ignored.

When the SingleSignOnFilter is used for authentication with AM, AM can view a session as idle even though a user continues to interact with IG. The user session eventually times out and the user must re-authenticate.

When the SingleSignOnFilter filter is used with the PolicyEnforcementFilter, the session is refreshed each time IG requests a policy decision from AM. The session is less likely to become idle, and this property less required.

```
{
  "sessionIdleRefresh": {
    "enabled": configuration expression<boolean>,
    "interval": configuration expression<duration>
  }
}
```

`enabled`: *configuration expression*<*boolean*>, *optional*

Enable refresh of idle sessions.

Default: `false`

`interval`: *configuration expression*<*duration*>, *optional*

Duration to wait after a session becomes idle before requesting a session refresh.

Consider setting the refresh interval in line with the latest access time update frequency of AM. For example, if IG requests a refresh every 60 seconds, but the update frequency of AM is 5 minutes, AM ignores most of the IG requests.

> **Important**
>
> Each session refresh must be reflected in the AM core token service. Setting the interval to a duration lower than one minute can adversely impact AM performance.

Default: `5 minutes`

`"sessionProperties"`: *array of configuration expression<strings>, optional*

The list of user session properties to retrieve from AM by the SessionInfoFilter.

Default: All available session properties are retrieved from AM.

`"ssoTokenHeader"`: *configuration expression<string>, optional*

The header name or cookie name where this AM server expects to find SSO tokens.

If a value for `ssoTokenHeader` is provided, IG uses that value. Otherwise, IG queries the AM `/serverinfo/*` endpoint for the header or cookie name.

Default: Empty. IG queries AM for the cookie name.

`"version"`: *configuration expression<string>, optional*

The version number of the AM server. IG uses the AM version to establish endpoints for its interaction with AM.

The AM version is derived as follows, in order of precedence:

- Discovered value: AmService discovers the AM version. If `version` is configured with a different value, AmService ignores the value of `version` and issues a warning.
- Value in `version`: AmService cannot discover the AM version, and `version` is configured.
- Default value of AM 6: AmService cannot discover the AM version, and `version` is not configured.

If you use a feature that is supported only in a higher AM version than discovered or specifed, a message can be logged or an error thrown.

Default: AM 6.

More information

org.forgerock.openig.tools.am.AmService

ClientRegistration

A ClientRegistration holds information about registration with an OAuth 2.0 Authorization Server or OpenID Provider.

The configuration includes the client credentials that are used to authenticate to the identity provider. The client credentials can be included directly in the configuration, or retrieved in some other way using an expression, described in Expressions.

Usage

```
{
  "name": string,
  "type": "ClientRegistration",
  "config": {
    "clientId": configuration expression<string>,
    "issuer": Issuer reference,
    "scopes": [ configuration expression<string>, ...],
    "registrationHandler": Handler reference,
    "authenticatedRegistrationHandler": Handler reference,
    "clientSecretId": configuration expression<secret-id>, //deprecated
    "jwtExpirationTimeout": duration, //deprecated
    "privateKeyJwtSecretId": configuration expression<secret-id>, //deprecated
    "secretsProvider": SecretsProvider reference, //deprecated
    "tokenEndpointAuthMethod": enumeration, //deprecated
    "tokenEndpointAuthSigningAlg": string //deprecated
  }
}
```

Properties

`"clientId"`: *configuration expression<string>, required*

The `client_id` obtained when registering with the Authorization Server. See also Expressions.

When using a login page with AuthorizationCodeOAuth2ClientFilter, the link to the `/login` endpoint must refer to a valid `clientId` identified by this property.

`"issuer"`: *Issuer reference, required*

The provider configuration to use for this client registration. Provide either the name of a Issuer object defined in the heap or an inline Issuer configuration object. See also Issuer.

`"scopes"`: *array of configuration expression<strings>, optional*

Array of scope strings to present to the user for approval, and include in tokens so that protected resources can make decisions about access.

Default: Empty

`"registrationHandler"`: *Handler reference, optional*

HTTP client handler to invoke during client registration, to access endpoints that do not require client authentication. Provide either the name of a Handler object defined in the heap or an inline Handler configuration object.

Usually set this to the name of a ClientHandler configured in the heap, or a chain that ends in a ClientHandler.

Default: ClientHandler.

`"authenticatedRegistrationHandler"`: *Handler reference, optional*

HTTP client handler to invoke during client registration, to access endpoints that require client authentication. Configure this property as a Chain, using one of the following filters for client authentication:

- ClientSecretBasicAuthenticationFilter
- ClientSecretPostAuthenticationFilter
- EncryptedPrivateKeyJwtClientAuthenticationFilter
- PrivateKeyJwtClientAuthenticationFilter

```
{
  "name": "AuthenticatedRegistrationHandler",
  "type": "Chain",
  "config": {
    "handler": "ForgeRockClientHandler",
    "filters": [
      {
        "type": "ClientSecretBasicAuthenticationFilter",
        "config": {
          "clientId": "service-client",
          "clientSecretId": "client.secret.id",
          "secretsProvider" : "SystemAndEnvSecretStore-1",
        }
      }
    ]
  }
}
```

Default: `registrationHandler` with no authentication filter.

`"clientSecretId"`: *configuration expression<secret-id>, required if* `tokenEndpointAuthMethod` *is* `client_secret_basic` *or* `client_secret_post`

> **Important**
>
> This property is deprecated; use `authenticatedRegistrationHandler` instead. For more information, refer to the Deprecated section of the *Release Notes*.

The secret ID of the client secret required to authenticate the client to the Authorization Server.

This secret ID must point to a GenericSecret.

`"jwtExpirationTimeout"`: *duration, optional*

> **Important**
>
> This property is deprecated; use `authenticatedRegistrationHandler` instead. For more information, refer to the Deprecated section of the *Release Notes*.

When `private_key_jwt` is used for authentication, this property specifies the duration for which the JWT is valid.

Default: 1 minute

`"privateKeyJwtSecretId"`: *configuration expression<secret-id>, required when* `private_key_jwt` *is used for client authentication*

> **Important**
>
> This property is deprecated; use `authenticatedRegistrationHandler` instead. For more information, refer to the Deprecated section of the *Release Notes*.

The secret ID of the key that is used to sign the JWT.

This secret ID must point to a CryptoKey.

`"secretsProvider"`: *SecretsProvider reference, required*

> **Important**
>
> This property is deprecated; use `authenticatedRegistrationHandler` instead. For more information, refer to the Deprecated section of the *Release Notes*.

The SecretsProvider object to query for the client's GenericSecret. For more information, see SecretsProvider.

Default: The route's default secret service. For more information, refer to Default secrets object.

`"tokenEndpointAuthMethod"`: *enumeration, optional*

> **Important**
>
> This property is deprecated; use `authenticatedRegistrationHandler` instead. For more information, refer to the Deprecated section of the *Release Notes*.

The authentication method with which a client authenticates to the authorization server or OpenID provider at the token endpoint. For information about client authentication methods, see OpenID Client Authentication. The following client authentication methods are allowed:

- `client_secret_basic`: Clients that have received a `client_secret` value from the Authorization Server authenticate with the Authorization Server by using the HTTP Basic authentication scheme, as in the following example:

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Authorization: Basic ....
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=...
```

- `client_secret_post`: Clients that have received a `client_secret` value from the Authorization Server authenticate with the Authorization Server by including the client credentials in the request body, as in the following example:

```
POST /oauth2/token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&;
client_id=...&
client_secret=...&
code=...
```

- `private_key_jwt`: Clients send a signed JSON Web Token (JWT) to the Authorization Server. IG builds and signs the JWT, and prepares the request as in the following example:

```
POST /token HTTP/1.1
Host: as.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code&
code=...&
client_id=<clientregistration_id>&
client_assertion_type=urn%3Aietf%3Aparams%3Aoauth%3Aclient-assertion-type%3Ajwt-bearer&
client_assertion=PHNhbWxwOl ... ZT
```

If the Authorization Server doesn't support `private_key_jwt`, a dynamic registration falls back on the method returned by the Authorization Server, for example, `client_secret_basic` or `client_secret_post`.

If `tokenEndpointAuthSigningAlg` is not configured, the `RS256` signing algorithm is used for `private_key_jwt`.

Consider these points for identity providers:

- Some providers accept more than one authentication method.
- If a provider strictly enforces how the client must authenticate, align the authentication method with the provider.
- If a provider doesn't support the authentication method, the provider sends an HTTP 400 Bad Request response with an `invalid_client` error message, according to RFC 6749 The OAuth 2.0 Authorization Framework, section 5.2.
- If the authentication method is invalid, the provider sends an `IllegalArgumentException`.

Default: `client_secret_basic`

"`tokenEndpointAuthSigningAlg`": *string, optional*

> **Important**
>
> This property is deprecated; use `authenticatedRegistrationHandler` instead. For more information, refer to the Deprecated section of the *Release Notes*.

The JSON Web Algorithm (JWA) used to sign the JWT that is used to authenticate the client at the token endpoint. The property is used when `private_key_jwt` is used for authentication.

Use one of the following algorithms:

- `RS256`: RSA using SHA-256
- `ES256`: ECDSA with SHA-256 and NIST standard P-256 elliptic curve
- `ES384`: ECDSA with SHA-384 and NIST standard P-384 elliptic curve
- `ES512`: ECDSA with SHA-512 and NIST standard P-521 elliptic curve

Default: `RS256`

Example

Refer to AM as a single OpenID Connect provider.

More information

org.forgerock.openig.filter.oauth2.client.ClientRegistration

Issuer, AuthorizationCodeOAuth2ClientFilter

The OAuth 2.0 Authorization Framework

The OAuth 2.0 Authorization Framework: Bearer Token Usage

OpenID Connect

ClientTlsOptions

Configures connections to the TLS-protected endpoint of servers, when IG is client-side.

When IG is *client-side*, IG sends requests to a proxied application, or requests services from a third-party application. IG is acting as a client of the application, and the application is acting as a server.

Use ClientTlsOptions in ClientHandler, ReverseProxyHandler, and AmService.

Usage

```
{
  "name": string,
  "type": "ClientTlsOptions",
  "config": {
    "keyManager": [ Key manager reference, ...],
    "trustManager": [ Trust manager reference, ...],
    "sslCipherSuites": [ configuration expression<string>, ...],
    "sslContextAlgorithm": configuration expression<string>,
    "sslEnabledProtocols": [ configuration expression<string>, ...],
    "alpn": object,
    "hostnameVerifier": configuration expression<enumeration>
  }
}
```

Properties

`"keyManager"`: *array of key manager references, optional*

One or more of the following objects to serve the same secret key and certificate pair for TLS connections to all server names in the deployment:

- SecretsKeyManager
- KeyManager (deprecated)

Key managers are used to prove the identity of the local peer during TLS handshake, as follows:

- When ServerTlsOptions is used in an HTTPS connector configuration (server-side), the key managers to which ServerTlsOptions refers are used to prove this IG's identity to the remote peer (client-side). This is the usual TLS configuration setting (without mTLS).
- When ClientTlsOptions is used in a ClientHandler or ReverseProxyHandler configuration (client-side), the key managers to which ClientTlsOptions refers are used to prove this IG's identity to the remote peer (server-side). This configuration is used in mTLS scenarios.

Default: None

`"trustManager"`: *array of trust manager references, optional*

One or more of the following objects to manage IG's public key certificates:

- SecretsTrustManager
- TrustAllManager
- TrustManager (deprecated)

> **Important**
>
> When the TrustManager object is configured, only certificates accessible through that TrustManager are trusted. Default and system certificates are no longer trusted.

Trust managers verify the identity of a peer by using certificates, as follows:

- When ServerTlsOptions is used in an HTTPS connector configuration (server-side), ServerTlsOptions refers to trust managers that verify the remote peer's identity (client-side). This configuration is used in mTLS scenarios.
- When ClientTlsOptions is used in a ClientHandler or a ReverseProxyHandler configuration (client-side), ClientTlsOptions refers to trust managers that verify the remote peer's identity (server-side). This is the usual TLS configuration setting (without mTLS).

If `trustManager` is not configured, IG uses the default Java truststore to verify the remote peer's identity. The default Java truststore depends on the Java environment. For example, `$JAVA_HOME/lib/security/cacerts`.

Default: No `trustManager` is set, and IG uses the default and system certificates

`"sslCipherSuites"`: *array of configuration expression<strings>, optional*

Array of cipher suite names, used to restrict the cipher suites allowed when negotiating transport layer security for an HTTPS connection.

For information about the available cipher suite names, refer to the documentation for the Java virtual machine (JVM) where you run IG. For Oracle Java, refer to the list of JSSE Cipher Suite Names.

Default: Allow any cipher suite supported by the JVM.

`"sslContextAlgorithm"`: *configuration expression<string>, optional*

The `SSLContext` algorithm name, as listed in the table of SSLContext Algorithms for the Java Virtual Machine (JVM).

Default: `TLS`

`"sslEnabledProtocols"`: *array of configuration expression<strings>, optional*

Array of protocol names, used to restrict the protocols allowed when negotiating transport layer security for an HTTPS connection.

For information about the available protocol names, refer to the documentation for the Java Virtual Machine (JVM). For Oracle Java, refer to the list of Additional JSSE Standard Names.

Follow these protocol recommendations:

- Use TLS 1.3 when it is supported by available libraries, otherwise use TLS 1.2.
- If TLS 1.1 or TLS 1.0 is required for backwards compatibility, use it only with express approval from enterprise security.
- Do not use deprecated versions SSL 3 or SSL 2.

Default: TLS 1.3, TLS 1.2

`"alpn"`: *object, optional*

A flag to enable the Application-Layer Protocol Negotiation (ALPN) extension for TLS connections.

```
{
  "alpn": {
    "enabled": configuration expression<boolean>
  }
}
```

`enabled`: *configuration expression<boolean>, optional*

- `true`: Enable ALPN. Required for HTTP/2 connections over TLS
- `false`: Disable ALPN.

Default: `true`

`"hostnameVerifier"`: *configuration expression<enumeration>, optional*

The method to handle hostname verification for outgoing SSL connections.

For backward compatibility, when a ClientHandler or ReverseProxyHandler includes the deprecated `"hostnameVerifier": "ALLOW_ALL"` configuration, it takes precedence over this property. A deprecation warning is written to the logs.

Use one of the following values:

- `ALLOW_ALL`: Allow a certificate issued by a trusted CA for any hostname or domain to be accepted for a connection to any domain.

  If the SSL endpoint uses a raw IP address rather than a fully-qualified hostname, you must configure this property as `ALLOW_ALL`.

  To prevent the compromise of TLS connections, use `ALLOW_ALL` in development mode only. In production, use `STRICT`.

  > **Caution**
  >
  > The `ALLOW_ALL` setting allows a certificate issued for one company to be accepted as a valid certificate for another company.

- `STRICT`: Match the hostname either as the value of the the first CN, or any of the subject-alt names.

  A wildcard can occur in the CN, and in any of the subject-alt names. Wildcards match one domain level, so `*.example.com` matches `www.example.com` but not `some.host.example.com`.

Default: `STRICT`

Example

For an example that uses ClientTlsOptions, refer to Configure IG for HTTPS (client-side).

Delegate

Delegates all method calls to a referenced handler, filter, or any object type.

Use a Delegate to decorate referenced objects differently when they are used multiple times in a configuration.

Usage

```
{
  "filter or handler": {
  "type": "Delegate",
  [decorator reference, ...],
    "config": {
      "delegate": object
    }
  }
}
```

Example

For an example of how to delegate tasks to `ForgeRockClientHandler`, and capture IG's interaction with AM, refer to Decorating IG's interactions with AM.

More information

org.forgerock.openig.decoration.DelegateHeaplet

JwtSession

Configures settings for stateless sessions.

Session information is serialized as a secure JWT, that is encrypted and signed, and optionally compressed. The resulting JWT string is placed in one or more JWT session cookies. The cookies contain session attributes as JSON, and a marker for the session timeout.

Use JwtSession to configure stateless sessions as follows:

- Configure a JwtSession object named `Session` in the heap of `config.json`.

  Stateless sessions are created when a request traverses any route or subroute in the configuration. No routes can create stateful sessions.

- Configure a JwtSession object in the `session` property of a Route object.

  When a request enters the route, IG builds a new session object for the route. Any child routes inherit the session. The session information is saved/persisted when the response exits the route. For more information, refer to Route.

- Configure a JwtSession object in the `session` property of multiple sibling routes in the configuration, using an identical cookie name and cryptographic properties. Sibling routes are in the same configuration, with no ascending hierarchy to each other.

  When a JwtSession object is declared in a route, the session content is available only within that route. With this configuration, sibling routes can read/write in the same session.

Consider the following points when you configure JwtSession:

- Only JSON-compatible types can be serialized into a JWT and included in JWT session cookies. Compatible types include primitive JSON structures, lists, arrays, and maps. For more information, refer to http://json.org.
- The maximum size of the JWT session cookie is 4 KBytes, as defined by the browser. If the cookie exceeds this size, IG automatically splits it into multiple cookies.
- If an empty session is serialized, the supporting cookie is marked as expired and is effectively discarded.

  To prevent IG from cleaning up empty session cookies, consider adding some information to the session context by using an AssignmentFilter. For an example, refer to Adding info to a session.

- When HTTP clients perform multiple requests in a session that modify the content, the session information can become inconsistent.

For information about IG sessions, refer to Sessions.

Usage

```
{
  "name": string,
  "type": "JwtSession",
  "config": {
    "authenticatedEncryptionSecretId": configuration expression<secret-id>,
    "encryptionMethod": configuration expression<string>,
    "cookie": object,
    "sessionTimeout": configuration expression<duration>,
    "persistentCookie": configuration expression<boolean>,
    "secretsProvider": SecretsProvider reference,
    "skewAllowance": configuration expression<duration>,
    "useCompression": configuration expression<boolean>
  }
}
```

Properties

"authenticatedEncryptionSecretId": *configuration expression<secret-id>, optional*

The secret ID of the encryption key used to perform authenticated encryption on a JWT. Authenticated encryption encrypts data and then signs it with HMAC, in a single step.

This secret ID must point to a CryptoKey.

Authenticated encryption is achieved with a symmetric encryption key. Therefore, the secret must refer to a symmetric key.

For more information, refer to RFC 5116.

Default: IG generates a default symmetric key for authenticated encryption. Consequently, IG instances cannot share the JWT session.

"encryptionMethod": *configuration expression<string>, optional*

The algorithm to use for authenticated encryption. For information about allowed encryption algorithms, refer to RFC 7518: "enc" (Encryption Algorithm) Header Parameter Values for JWE.

Default: A256GCM

"cookie": *object, optional*

The configuration of the cookie used to store the encrypted JWT.

The maximum size of the JWT session cookie is 4 KBytes, as defined by the browser. If the cookie exceeds this size, IG automatically splits it into multiple cookies.

Default: The cookie is treated as a host-based cookie.

```
{
  "name": configuration expression<string>,
  "domain": configuration expression<string>,
  "httpOnly": configuration expression<boolean>,
  "path": configuration expression<string>,
  "sameSite": configuration expression<enumeration>,
  "secure": configuration expression<boolean>
}
```

`"name"` *configuration expression<string>, optional*

Name of the JWT cookie stored on the user agent. For security, change the default name of cookies.

Default: `openig-jwt-session`

`"domain"` *configuration expression<string>, optional*

Domain from which the JWT cookie can be accessed. When the domain is specified, a JWT cookie can be accessed from different hosts in that domain.

Set a domain only if the user agent is able to re-emit cookies on that domain on its next hop. For example, to re-emit a cookie on the domain `.example.com`, the user agent must be able to access that domain on its next hop.

Default: The fully qualified hostname of the user agent's next hop.

`"httpOnly"`*: configuration expression<boolean>, optional*

Flag to mitigate the risk of client-side scripts accessing protected cookies.

Default: `true`

`"path"`*: configuration expression<string>, optional*

Path protected by this session.

Set a path only if the user agent is able to re-emit cookies on the path. For example, to re-emit a cookie on the path `/home/cdsso`, the user agent must be able to access that path on its next hop.

Default: The path of the request that got the `Set-Cookie` in its response.

`"sameSite"`*: configuration expression<enumeration>, optional*

Options to manage the circumstances in which a cookie is sent to the server. Use one of the following values to reduce the risk of CSRF attacks:

- `STRICT`: Send the cookie only if the request was initiated from the cookie domain. Not case-sensitive.

  Use this value to reduce the risk of cross-site request forgery (CSRF) attacks.

- `LAX`: Send the cookie only with GET requests in a first-party context, where the URL in the address bar matches the cookie domain. Not case-sensitive.

  Use this value to reduce the risk of cross-site request forgery (CSRF) attacks.

- `NONE`: Send the cookie whenever a request is made to the cookie domain. Not case-sensitive.

With this setting, consider setting `secure` to `true` to prevent browsers from rejecting the cookie. For more information, refer to SameSite cookies.

Default: `LAX`

`"secure"`: *configuration expression<boolean>, optional*

Flag to limit the scope of the cookie to secure channels.

Set this flag only if the user agent is able to re-emit cookies over HTTPS on its next hop. For example, to re-emit a cookie with the `secure` flag, the user agent must be connected to its next hop by HTTPS.

Default: `false`

`"sessionTimeout"`: *configuration expression<duration>, optional*

The duration for which a JWT session is valid. If the supporting cookie is persistent, this property also defines the expiry of the cookie.

The value must be above zero. The maximum value is 3650 days (approximately 10 years). If you set a longer duration, IG truncates the duration to 3650 days.

Default: 30 minutes

`"persistentCookie"`: *configuration expression<boolean>,optional*

Whether or not the supporting cookie is persistent:

- `true`: the supporting cookie is a persistent cookie. Persistent cookies are re-emitted by the user agent until their expiration date or until they are deleted.
- `false`: the supporting cookie is a session cookie. IG does not specify an expiry date for session cookies. The user agent is responsible for deleting them when it considers that the session is finished (for example, when the browser is closed).

Default: `false`

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for the JWT session signing or encryption keys.

`"skewAllowance"`: *configuration expression<duration>, optional*

The duration to add to the validity period of a JWT to allow for clock skew between different servers.

A `skewAllowance` of 2 minutes affects the validity period as follows:

- A JWT with an `iat` of 12:00 is valid from 11:58 on the IG clock.
- A JWT with an `exp` 13:00 is expired after 13:02 on the IG clock.

Default: To support a zero-trust policy, the skew allowance is by default `zero`.

`"useCompression"`: *configuration expression boolean, optional*

A flag to compress the session JWT before it is placed in a cookie.

> **Important**
>
> Compression can undermine the security of encryption. Evaluate this threat according to your use case before you enable compression.

Default: `false`

Example

For information about configuring a JwtSession with authenticated encryption, see Encrypt JWT sessions.

For information about managing multiple instances of IG in the same deployment, refer to the Installation guide.

More information

For information about IG sessions, refer to Sessions.

KeyManager (deprecated)

> **Important**
>
> This object is deprecated; use SecretsKeyManager instead. For more information, refer to the Deprecated section of the *Release Notes*.

The configuration of a Java Secure Socket Extension KeyManager to manage private keys for IG. The configuration references the keystore that holds the keys.

When IG acts as a server, it uses a KeyManager to prove its identity to the client. When IG acts as a client, it uses a KeyManager to prove its identity to the server.

Usage

```
{
  "name": string,
  "type": "KeyManager",
  "config": {
    "keystore": KeyStore reference,
    "passwordSecretId": configuration expression<secret-id>,
    "alg": configuration expression<string>,
    "secretsProvider": SecretsProvider reference
  }
}
```

Properties

`"keystore"`: *KeyStore reference*, *required*

The KeyStore (deprecated) object that references the store for key certificates. When `keystore` is used in a KeyManager, it queries for private keys; when `keystore` is used in a TrustManager, it queries for certificates.

Provide either the name of the keystore object defined in the heap or an inline keystore configuration object.

`"passwordSecretId"`: *configuration expression<secret-id>, required*

The secret ID of the password required to read private keys from the keystore.

This secret ID must point to a GenericSecret.

`"alg"`: *configuration expression<string>, optional*

The certificate algorithm to use.

Default: the default for the platform, such as `SunX509`.

See also Expressions.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for the keystore password.

Example

The following example configures a KeyManager that depends on a KeyStore (deprecated) configuration. The KeyManager and KeyStore passwords are provided by Java system properties or environment variables, and retrieved by the SystemAndEnvSecretStore. By default, the password values must be base64-encoded.

```
{
  "name": "MyKeyManager",
  "type": "KeyManager",
  "config": {
    "keystore": {
      "type": "KeyStore",
      "config": {
        "url": "file://${env['HOME']}/keystore.p12",
        "passwordSecretId": "keymanager.keystore.secret.id",
        "secretsProvider": "SystemAndEnvSecretStore"
      }
    },
    "passwordSecretId": "keymanager.secret.id",
    "secretsProvider": "SystemAndEnvSecretStore"
  }
}
```

More information

org.forgerock.openig.security.KeyManagerHeaplet

JSSE Reference guide, KeyStore, TrustManager

KeyStore (deprecated)

> **Important**
>
> This object is deprecated; use KeyStoreSecretStore instead. For more information, refer to the Deprecated section of the *Release Notes*.

The configuration for a Java KeyStore, which stores cryptographic private keys and public key certificates.

> **Warning**
>
> Legacy keystore types such as JKS and JCEKS are supported but are not secure. Consider using the PKCS#12 keystore type.

Usage

```
{
  "name": name,
  "type": "KeyStore",
  "config": {
    "url": configuration expression<url>,
    "passwordSecretId": configuration expression<secret-id>,
    "type": configuration expression<string>,
    "secretsProvider": SecretsProvider reference
  }
}
```

Properties

`"url"`: *configuration expression<url>, required*

URL to the keystore file.

See also Expressions.

`"passwordSecretId"`: *configuration expression<secret-id>, optional*

The secret ID of the password required to read private keys from the KeyStore.

This secret ID must point to a GenericSecret.

If the KeyStore is used as a truststore to store only public key certificates of peers and no password is required to do so, then you do not have to specify this field.

Default: No password is set.

See also Expressions.

`"type"`: *configuration expression<string>, optional*

The secret store type.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for the keystore password.

Example

The following example configures a KeyStore that references the Java KeyStore file `$HOME/keystore.p12`. The KeyStore password is provided by a Java system property or environment variable, and retrieved by the SystemAndEnvSecretStore. By default, the password value must be base64-encoded.

```
{
  "name": "MyKeyStore",
  "type": "KeyStore",
  "config": {
    "url": "file://${env['HOME']}/keystore.p12",
    "passwordSecretId": "keystore.secret.id",
    "secretsProvider": "SystemAndEnvSecretStore"
  }
}
```

More information

org.forgerock.openig.security.KeyStoreHeaplet

JSSE Reference guide

KeyManager

TrustManager

Issuer

Describes an OAuth 2.0 Authorization Server or an OpenID Provider that IG can use as a OAuth 2.0 client or OpenID Connect relying party.

An Issuer is usually referenced from a ClientRegistration.

Usage

```
{
  "name": string,
  "type": "Issuer",
  "config": {
    "wellKnownEndpoint": configuration expression<url>,
    "authorizeEndpoint": configuration expression<url>,
    "registrationEndpoint": configuration expression<url>,
    "tokenEndpoint": configuration expression<url>,
    "userInfoEndpoint": configuration expression<url>,
    "endSessionEndpoint": configuration expression<url>,
    "revocationEndpoint": configuration expression<url>,
    "issuerHandler": Handler reference,
    "issuerRepository": Issuer repository reference,
    "supportedDomains": [ pattern, ... ]
  }
}
```

Properties

If the provider has a well-known configuration URL as defined for OpenID Connect 1.0 Discovery that returns JSON with at least authorization and token endpoint URLs, then you can specify that URL in the provider configuration. Otherwise, you must specify at least the provider authorization and token endpoint URLs, and optionally the registration endpoint and user info endpoint URLs.

The provider configuration object properties are as follows:

`"name"`: *string*, ***required***

A name for the provider configuration.

`"wellKnownEndpoint"`: ***configuration expression*<*url*>, *required unless authorizeEndpoint and tokenEndpoint are specified***

The URL to the well-known configuration resource as described in OpenID Connect 1.0 Discovery.

`"authorizeEndpoint"`: ***configuration expression*<*url*>, *required unless obtained through wellKnownEndpoint***

The URL to the provider's OAuth 2.0 authorization endpoint.

`"registrationEndpoint"`: ***configuration expression*<*url*>, *optional***

The URL to the provider's OpenID Connect dynamic registration endpoint.

`"tokenEndpoint"`: ***configuration expression*<*url*>, *required unless obtained through wellKnownEndpoint***

The URL to the provider's OAuth 2.0 token endpoint.

`"userInfoEndpoint"`: ***configuration expression*<*url*>, *optional***

The URL to the provider's OpenID Connect UserInfo endpoint.

Default: no UserInfo is obtained from the provider.

`"endSessionEndpoint"`: ***configuration expression*<*url*>, *optional***

The URL to the Authorization Server's `end_session_endpoint`. In OpenID Connect, when a request accesses this endpoint, IG kills the user session in AM.

Consider the following example endpoint: `https://am.example.com:8443/openam/oauth2/realms/root/realms/alpha/connect/endSession`

For more information, refer to OpenID Connect Session Management.

Default: No endpoint

`"revocationEndpoint"`: ***configuration expression*<*url*>, *optional***

The URL to the Authorization Server's `revocation_endpoint`. When a request accesses this endpoint, IG revokes access tokens or refresh tokens associated to the current user session in AM.

Consider the following example endpoint: `https://am.example.com:8443/openam/oauth2/realms/root/realms/alpha/token/revoke`

Default: No endpoint

`"issuerHandler"`: *Handler reference*, ***optional***

Invoke this HTTP client handler to communicate with the Authorization Server.

Provide either the name of a Handler object defined in the heap or an inline Handler configuration object.

Usually set this to the name of a ClientHandler configured in the heap, or a chain that ends in a ClientHandler.

Default: IG uses the default ClientHandler.

See also Handlers, ClientHandler.

`"issuerRepository"`: *Issuer repository reference, optional*

A repository of OAuth 2.0 issuers, built from discovered issuers and the IG configuration.

Provide the name of an IssuerRepository object defined in the heap.

Default: Look up an issuer repository named `IssuerRepository` in the heap. If none is explicitly defined, then a default one named `IssuerRepository` is created in the current route.

See also IssuerRepository.

`"supportedDomains"`: *array of patterns, optional*

One or more domain patterns to match domain names that are handled by this issuer, used as a shortcut for OpenID Connect discovery before performing OpenID Connect dynamic registration.

In summary when the OpenID Provider is not known in advance, it might be possible to discover the OpenID Provider Issuer based on information provided by the user, such as an email address. The OpenID Connect discovery specification explains how to use WebFinger to discover the issuer. IG can discover the issuer in this way. As a shortcut IG can also use supported domains lists to find issuers already described in the IG configuration.

To use this shortcut, IG extracts the domain from the user input, and looks for an issuer whose supported domains list contains a match.

Supported domains patterns match host names with optional port numbers. Do not specify a URI scheme such as HTTP. IG adds the scheme. For instance, `*.example.com` matches any host in the `example.com` domain. You can specify the port number as well as in `host.example.com:8443`. Patterns must be valid regular expression patterns according to the rules for the Java Pattern class.

Examples

The following example shows an AM issuer configuration for AM. AM exposes a well-known endpoint for the provider configuration, but this example demonstrates use of the other fields:

```
{
    "name": "openam",
    "type": "Issuer",
    "config": {
        "authorizeEndpoint":
          "https://am.example.com:8443/openam/oauth2/authorize",
        "registration_endpoint":
          "https://am.example.com:8443/openam/oauth2/connect/register",
        "tokenEndpoint":
          "https://am.example.com:8443/openam/oauth2/access_token",
        "userInfoEndpoint":
          "https://am.example.com:8443/openam/oauth2/userinfo",
        "supportedDomains": [ "mail.example.*", "docs.example.com:8443" ]
    }
}
```

The following example shows an issuer configuration for Google:

```
{
    "name": "google",
    "type": "Issuer",
    "config": {
        "wellKnownEndpoint":
          "https://accounts.google.com/.well-known/openid-configuration",
        "supportedDomains": [ "gmail.*", "googlemail.com:8052" ]
    }
}
```

More information

org.forgerock.openig.filter.oauth2.client.Issuer

IssuerRepository

Stores OAuth 2 issuers that are discovered or built from the configuration.

It is not normally necessary to change this object. Change it only for the following tasks:

- To isolate different repositories in the same route.
- To view the interactions of the well-known endpoint, for example, if the `issuerHandler` is delegating to another handler.

Usage

```
{
   "name": string,
   "type": "IssuerRepository",
   "config": {
     "issuerHandler": Handler reference
   }
}
```

Properties

`"issuerHandler"`: *Handler reference, optional*

The default handler to fetch OAuth2 issuer configurations from the well-known endpoint.

Provide the name of a Handler object defined in the heap or an inline Handler configuration object.

Default: ForgeRockClientHandler

More information

org.forgerock.openig.filter.oauth2.client.IssuerRepository

JdbcDataSource

Manages connections to a JDBC data source.

To configure the connection pool, add a JdbcDataSource object named `AuditService` in the route heap.

Usage

```
{
  "name": string,
  "type": "JdbcDataSource",
  "config": {
    "dataSourceClassName": configuration expression<string>,
    "driverClassName": configuration expression<string>,
    "executor": ScheduledExectutorService reference,
    "jdbcUrl": configuration expression<url>,
    "passwordSecretId": configuration expression<secret-id>,
    "poolName": configuration expression<string>,
    "properties": object,
    "secretsProvider": SecretsProvider reference,
    "username": configuration expression<string>
  }
}
```

Properties

`"dataSourceClassName"`: *configuration expression<string>, optional*

The data source class name to use to connect to the database.

Depending on the underlying data source, use either `jdbcUrl`, or `dataSourceClassName` with `url`. See the Properties.

`"driverClassName"`: *configuration expression<string>, optional*

Class name of the JDBC connection driver. The following examples can be used:

- MySQL Connector/J: `com.mysql.jdbc.Driver`
- H2: `org.h2.Driver`

This property is optional, but required for older JDBC drivers.

`"executor"`: *ScheduledExecutorService reference, optional*

A ScheduledExecutorService for maintenance tasks.

Default: ScheduledExecutorService.

`"jdbcUrl"`: *configuration expression<url>, optional*

The JDBC URL to use to connect to the database.

Depending on the underlying data source, use either `jdbcUrl`, or `dataSourceClassName` with `url`. See the Properties.

`"passwordSecretId"`: *configuration expression<secret-id>, required if the database is password-protected*

The secret ID of the password to access the database.

This secret ID must point to a GenericSecret.

`"poolName"`: *configuration expression<string>, optional*

The connection pool name. Use to identify a pool easily for maintenance and monitoring.

`"properties"`: *object, optional*

Server properties specific to the type of data source being used. The values of the object are evaluated as configuration expression<strings>.

For information about available options, refer to the data source documentation.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for passwords and cryptographic keys.

`"username"`: *configuration expression<string>, optional*

The username to access the database.

Example

For an example that uses JdbcDataSource, refer to Log In With Credentials From a Database and Recording Access Audit Events in a Database.

The following example configures a JdbcDataSource with a `dataSourceClassName` and `url`:

```
"config": {
  "username": "testUser",
  "dataSourceClassName": "org.h2.jdbcx.JdbcDataSource",
  "properties": {
    "url": "jdbc:h2://localhost:3306/auth"
  },
  "passwordSecretId": "database.password",
  "secretsProvider": "MySecretsProvider"
}
```

The following example configures a JdbcDataSource with `jdbcUrl` alone:

```
"config": {
  "username": "testUser",
  "jdbcUrl": "jdbc:h2://localhost:3306/auth",
  "passwordSecretId": "database.password",
  "secretsProvider": "MySecretsProvider"
}
```

The following example configures a JdbcDataSource with `jdbcUrl` and `driverName`. Use this format for older drivers, where `jdbcUrl` does not provide enough information:

```
"config": {
  "username": "testUser",
  "jdbcUrl": "jdbc:h2://localhost:3306/auth",
  "driverName": "org.h2.Driver",
  "passwordSecretId": "database.password",
  "secretsProvider": "MySecretsProvider"
}
```

More information

org.forgerock.openig.sql.JdbcDataSourceHeaplet

KerberosIdentityAssertionPlugin

Use with an IdentityAssertionHandler to validate Kerberos authentication tickets locally.

The KerberosIdentityAssertionPlugin doesn't support Windows New Technology LAN Manager (NTLM) tokens.

Usage

```
{
    "name": string,
    "type": "KerberosIdentityAssertionPlugin",
    "config": {
      "serviceLogin": ServiceLogin reference,
      "trustedRealms": [configuration_expression<string>, ...]
    }
}
```

Properties

`"serviceLogin"`: *ServiceLogin reference, required*

A service account object to log IG in to the Kerberos server so that IG can act on user tokens. IG will be able to validate user tokens, for example.

IG provides the following service account objects for the KerberosIdentityAssertionPlugin:

UsernamePasswordServiceLogin

Log IG in to the Kerberos server by using a service account username and password.

```
{
  "type": "UsernamePasswordServiceLogin",
  "config": {
    "username": configuration_expression<string>,
    "passwordSecretId": configuration expression<secret-id>,
    "secretsProvider": SecretsProvider reference
  }
}
```

`"username"`: *configuration expression<string>, required*

Service username.

`"passwordSecretId"`: *configuration expression<secret-id>, required if the proxy requires authentication*

The secret ID of the service account password.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for the password.

KeytabServiceLogin

Log IG in to the Kerberos server by using a Keytab file.

> **Important**
>
> This service account object is less secure than UsernamePasswordServiceLogin; use it only for testing or to ease migration. In production environments, always use the most secure options available.

```
{
  "type": "KeytabServiceLogin",
  "config": {
    "username": configuration_expression<string>,
    "keytabFile": configuration expression<secret-id>,
    "executor": ScheduledExecutorService reference
  }
}
```

`"username"`: *configuration expression<string>, required*

Service username.

`"keytabFile"`: *configuration expression<string>, required*

Path to the keytab file. Both the username and keytabFile are required for login.

`"executor"`: *ScheduledExecutorService reference, optional*

An executor service to schedule the execution of tasks during a keytab service login.

Default: ScheduledExecutorService or an executor service declared in the heap.

`"trustedRealms"`: *array of configuration expression<strings>, optional*

A list of one or more Kerberos realms that are expected to match the principal's realm from the user's Kerberos ticket.

Kerberos tickets are accepted only if the principal's realm matches a realm in the list.

Default: Empty

Examples

```
{
  "type": "KerberosIdentityAssertionPlugin",
  "config": {
    "serviceLogin": "UsernamePasswordServiceLogin",
    "trustedRealms": ["EXAMPLE.COM"]
  }
}
```

```
{
  "type": "UsernamePasswordServiceLogin",
  "config": {
    "username": "igsa",
    "passwordSecretId": "igsa.id",
    "secretsProvider": "mySecretsProvider"
  }
}
```

When using a Kerberos keytab file, generate it for IG with the Windows `ktpass` command. The following commands add and view a Service Principal Name (SPN) for the IG service account, `igsa`, and generate a keytab file for IG in the `example.com` realm mapped to the service account username. Run the commands as the Windows Administrator to ensure you have access to everything necessary:

```
# Add the SPN for the service account:
PS C:\path\to> setspn -s HTTP/ig.example.com igsa

# View the SPN for the service account:
PS C:\path\to> setspn -l igsa
Registered ServicePrincipalNames for CN=igsa,CN=Users,DC=example,DC=com:
        HTTP/ig.example.com

# Generate the keytab file:
PS C:\path\to> ktpass -out keytab.file -princ HTTP/ig.example.com@EXAMPLE.COM -pass `
+rndPass -maxPass 256 -mapuser igsa -crypto All -ptype KRB5_NT_PRINCIPAL -kvno 0
```

In the IG configuration, you can use the Kerberos principal as the username:

```
{
  "type": "KeytabServiceLogin",
  "config": {
    "username": "HTTP/ig.example.com@EXAMPLE.COM",
    "keytabFile": "/path/to/keytab.file"
  }
}
```

More information

org.forgerock.openig.assertion.plugin.kerberos.KerberosIdentityAssertionPlugin

org.forgerock.openig.assertion.plugin.IdentityAssertionPlugin

org.forgerock.openig.handler.assertion.IdentityAssertionClaims

The following APIs are used in this class:

- Kerberos: The Network Authentication Protocol
- Kerberos Authentication Overview
- Kerberos Requirements
- Single Sign-on Using Kerberos in Java
- Java Troubleshooting
- How do I enable debug logging for troubleshooting Kerberos and WDSSO issues in PingAM?

ProxyOptions

A proxy to which a ClientHandler or ReverseProxyHandler can submit requests, and an AmService can submit Websocket notifications.

Use this object to configure a proxy for AM notifications, and use it in a ClientHandler or ReverseProxyHandler, and again in an AmService notifications block.

Usage

Use one of the following ProxyOption types with the `proxyOptions` option of ClientHandler, ReverseProxyHandler, and AmService:

- No proxy.

```
{
  "name": string,
  "type": "NoProxyOptions"
}
```

- System defined proxy options.

```
{
  "name": string,
  "type": "SystemProxyOptions"
}
```

- Custom proxy

```
{
  "name": string,
  "type": "CustomProxyOptions",
  "config": {
    "uri": configuration expression<url>,
    "username": configuration expression<string>,
    "passwordSecretId": configuration expression<secret-id>,
    "secretsProvider": SecretsProvider reference
  }
}
```

Default: NoProxyOptions

Properties

`"uri"`: *configuration expression<url>, required*

URI of a server to use as a proxy for outgoing requests.

The result of the expression must be a string that represents a valid URI, but is not a real `java.net.URI` object.

`"username"`: *configuration expression<string>, required if the proxy requires authentication*

Username to access the proxy server.

`"passwordSecretId"`: *configuration expression<secret-id>, required if the proxy requires authentication*

The secret ID of the password to access the proxy server.

This secret ID must point to a GenericSecret.

`"secretsProvider"`: _ *SecretsProvider <reference>, required*

The SecretsProvider to query for the proxy's password.

Example

In the following example, the handler passes outgoing requests to the proxy server, which requires authentication:

```
"handler": {
  "type": "ClientHandler" or "ReverseProxyHandler",
  "config": {
     "proxyOptions": {
       "type": "CustomProxyOptions",
       "config": {
          "uri": "http://proxy.example.com:3128",
          "username": "proxyuser",
          "passwordSecretId": "myproxy.secret.id",
          "secretsProvider": "SystemAndEnvSecretStore"
       }
    }
  }
}
```

In the following example, the AmService notification service passes Websocket notifications to the proxy server, which requires authentication:

```
"type": "AmService",
   "config": {
     ...
     "notifications": {
       "proxyOptions": {
         "type": "CustomProxyOptions",
         "config": {
           "uri": "http://proxy.example.com:3128",
           "username": "proxyuser",
           "passwordSecretId": "myproxy.secret.id",
           "secretsProvider": "SystemAndEnvSecretStore"
         }
       }
     }
   }
}
```

ScheduledExecutorService

An executor service to schedule tasks for execution after a delay or for repeated execution with a fixed interval of time in between each execution. You can configure the number of threads in the executor service and how the executor service is stopped.

The `ScheduledExecutorService` is shared by all downstream components that use an executor service.

Usage

```
{
  "name": string,
  "type": "ScheduledExecutorService",
  "config": {
    "corePoolSize": configuration expression<number>,
    "gracefulStop": configuration expression<boolean>,
    "gracePeriod": configuration expression<duration>
  }
}
```

Properties

`"corePoolSize"`: *configuration expression<number>, optional*

The minimum number of threads to keep in the pool. If this property is an expression, the expression is evaluated as soon as the configuration is read.

The value must be an integer greater than zero.

Default: 1

`"gracefulStop"`: *configuration expression<boolean>, optional*

Defines how the executor service stops.

If true, the executor service does the following:

- Blocks the submission of new jobs.
- If a grace period is defined, waits for up to that maximum time for submitted and running jobs to finish.
- Removes submitted jobs without running them.
- Attempts to end running jobs.

If false, the executor service does the following:

- Blocks the submission of new jobs.
- If a grace period is defined, ignores it.
- Removes submitted jobs without running them.
- Attempts to end running jobs.

Default: true

`"gracePeriod"`: *configuration expression<duration>, optional*

The maximum time that the executor service waits for running jobs to finish before it stops. If this property is an expression, the expression is evaluated as soon as the configuration is read.

If all jobs finish before the grace period, the executor service stops without waiting any longer. If jobs are still running after the grace period, the executor service removes the scheduled tasks, and notifies the running tasks for interruption.

When `gracefulStop` is `false`, the grace period is ignored.

Default: 10 seconds

Example

The following example creates a thread pool to execute tasks. When the executor service is instructed to stop, it blocks the submission of new jobs, and waits for up to 10 seconds for submitted and running jobs to complete before it stops. If any jobs are still submitted or running after 10 seconds, the executor service stops anyway and prints a message.

```
{
    "name": "ExecutorService",
    "comment": "Default service for executing tasks in the background.",
    "type": "ScheduledExecutorService",
    "config": {
        "corePoolSize": 5,
        "gracefulStop": true,
        "gracePeriod": "10 seconds"
    }
}
```

More information

org.forgerock.openig.thread.ScheduledExecutorServiceHeaplet

ScriptableResourceUriProvider

Use a script to return a resource URL to include in policy decision requests to AM. The result of the script must be a string that represents a resource URL. The PolicyEnforcementFilter uses the returned resource URL as a key to identify cached policy decisions.

To increase performance, use ScriptableResourceUriProvider in conjunction with AM policies to maximize the the cache hit ratio.

When a request matches a cached policy decision, IG can reuse the decision without asking AM for a new decision. When caching is disabled, IG must ask AM to make a decision for each request.

Usage

```
"resourceUriProvider": {
  "type": "ScriptableResourceUriProvider",
  "config": {
    "type": configuration expression<string>,
    "file": configuration expression<string>, // Use either "file"
    "source": [ string, ... ],               // or "source", but not both.
    "args": object,
    "clientHandler": Handler reference
  }
}
```

Properties

For information about properties, refer to Scripts.

ServerTlsOptions

When IG is *server-side*, applications send requests to IG or request services from IG. IG is acting as a server of the application, and the application is acting as a client.

ServerTlsOptions configures the TLS-protected endpoint when IG is server-side. Use ServerTlsOptions in `admin.json`.

Usage

```
{
  "type": "ServerTlsOptions",
  "config": {
    "keyManager": [ Key manager reference, ...], // Use "keyManager" or
    "sni": object,                               // "sni", but not both
    "trustManager": [ Trust manager reference, ...],
    "sslCipherSuites": [ configuration expression<string>, ...],
    "sslContextAlgorithm": configuration expression<string>,
    "sslEnabledProtocols": [ configuration expression<string>, ...],
    "alpn": object,
    "clientAuth": configuration expression<enumeration>,
  }
}
```

Properties

Either `sni` or `keyManager` must be configured. When both are configured, `sni` takes precedence and a warning is logged. When neither is configured, an exception is thrown and a warning is logged.

`"keyManager"`: *array of key manager references, required if* `sni` *isn't configured*

One or more of the following objects to serve the same secret key and certificate pair for TLS connections to all server names in the deployment:

- SecretsKeyManager
- KeyManager (deprecated)

Key managers are used to prove the identity of the local peer during TLS handshake, as follows:

- When ServerTlsOptions is used in an HTTPS connector configuration (server-side), the key managers to which ServerTlsOptions refers are used to prove this IG's identity to the remote peer (client-side). This is the usual TLS configuration setting (without mTLS).
- When ClientTlsOptions is used in a ClientHandler or ReverseProxyHandler configuration (client-side), the key managers to which ClientTlsOptions refers are used to prove this IG's identity to the remote peer (server-side). This configuration is used in mTLS scenarios.

Default: None

`"trustManager"`: *array of trust manager references, optional*

One or more of the following objects to manage IG's public key certificates:

- SecretsTrustManager
- TrustAllManager
- TrustManager (deprecated)

> **Important**
>
> When the TrustManager object is configured, only certificates accessible through that TrustManager are trusted. Default and system certificates are no longer trusted.

Trust managers verify the identity of a peer by using certificates, as follows:

- When ServerTlsOptions is used in an HTTPS connector configuration (server-side), ServerTlsOptions refers to trust managers that verify the remote peer's identity (client-side). This configuration is used in mTLS scenarios.
- When ClientTlsOptions is used in a ClientHandler or a ReverseProxyHandler configuration (client-side), ClientTlsOptions refers to trust managers that verify the remote peer's identity (server-side). This is the usual TLS configuration setting (without mTLS).

If `trustManager` is not configured, IG uses the default Java truststore to verify the remote peer's identity. The default Java truststore depends on the Java environment. For example, `$JAVA_HOME/lib/security/cacerts`.

Default: No `trustManager` is set, and IG uses the default and system certificates

`"sni"`: *object*, *required if* `keyManager` *is not configured*

Server Name Indication (SNI) is an extension of the TLS handshake, to serve different secret key and certificate pairs to the TLS connections on different server names. Use this property to host multiple domains on the same machine. For more information, refer to Server Name Indication.

During a TLS handshake, vert.x accesses secret key and certificate pairs synchronously; they are loaded in memory at IG startup, and **must** be present. You must restart IG to update a secret key and certificate pair.

For an example that uses this property, refer to Serve different certificates for TLS connections to different server names.

```
{
  "sni": {
    "serverNames": map,
    "defaultSecretId": configuration expression<secret-id>,
    "secretsProvider": SecretsProvider reference
  }
}
```

`serverNames`: *map*, *required*

A map of one or more data pairs with the format `Map<String, String>`, where:

- The key is the name of server provided during TLS handshake, or a configuration expression that evaluates to the name
- The value is a string representing the secret ID of the servers' secret key/certificate pair. Alternatively, it can be a configuration expression that evaluates to that string.

  The following format is required:

```
{
  "serverNames": {
    "configuration expression<string>": "configuration expression<string>",
    ...
  }
}
```

In the following example, the keys and values in the map are strings:

```
"serverNames": {
  "app1.example.com": "my.app1.secretId",
  "app2.example.com": "my.app2.secretId",
  "*.test.com": "my.wildcard.test.secretId"
}
```

In the following example, the keys and values in the map are configuration expressions:

```
"serverNames": {
  "${server.name.available.at.config.time}" : "${secret.id.available.at.config.time}"
}
```

Note the following points:

- **One server cannot be mapped to multiple certificates.**

  IG cannot provide multiple certificates for the same server name, as is allowed by Java's key managers.

- **Multiple servers can be mapped to one certificate.**

  Map server names individually. In the following configuration, both server names use the same certificate:

  ```
  "serverNames": {
    "cat.com" : "my.secret.id",
    "dog.org" : "my.secret.id"
  }
  ```

  Use the `*` wildcard in the server name to map groups of server names. In the following configuration, `app1.example.com` and `app2.example.com` use the same certificate:

  ```
  "serverNames": {
    "*.example.com": "my.wildcard.secret.id"
  }
  ```

`"defaultSecretId"`: *configuration expression<secret-id>, required*

The secret ID representing the certificate to use when an unmapped server name is provided during TLS handshake.

This secret ID must point to a CryptoKey.

For information about how IG manages secrets, refer to About secrets.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for each secret ID.

`"sslCipherSuites"`: *array of configuration expression<strings>, optional*

Array of cipher suite names, used to restrict the cipher suites allowed when negotiating transport layer security for an HTTPS connection.

For information about the available cipher suite names, refer to the documentation for the Java virtual machine (JVM) where you run IG. For Oracle Java, refer to the list of JSSE Cipher Suite Names.

Default: Allow any cipher suite supported by the JVM.

`"sslContextAlgorithm"`: *configuration expression<string>, optional*

The `SSLContext` algorithm name, as listed in the table of SSLContext Algorithms for the Java Virtual Machine (JVM).

Default: `TLS`

`"sslEnabledProtocols"`: *array of configuration expression<strings>, optional*

Array of protocol names, used to restrict the protocols allowed when negotiating transport layer security for an HTTPS connection.

For information about the available protocol names, refer to the documentation for the Java Virtual Machine (JVM). For Oracle Java, refer to the list of Additional JSSE Standard Names.

Follow these protocol recommendations:

- Use TLS 1.3 when it is supported by available libraries, otherwise use TLS 1.2.
- If TLS 1.1 or TLS 1.0 is required for backwards compatibility, use it only with express approval from enterprise security.
- Do not use deprecated versions SSL 3 or SSL 2.

Default: TLS 1.3, TLS 1.2

`"alpn"`: *object, optional*

A flag to enable the Application-Layer Protocol Negotiation (ALPN) extension for TLS connections.

```
{
  "alpn": {
    "enabled": configuration expression<boolean>
  }
}
```

`enabled`*: configuration expression<boolean>, optional*

- `true`: Enable ALPN. Required for HTTP/2 connections over TLS
- `false`: Disable ALPN.

Default: `true`

`"clientAuth"`*: configuration expression<enumeration>, optional*

The authentication expected from the client. Use one of the following values:

- `REQUIRED`: Require the client to present authentication. If it is not presented, then decline the connection.
- `REQUEST`: Request the client to present authentication. If it is not presented, then accept the connection anyway.
- `NONE`: Accept the connection without requesting or requiring the client to present authentication.

Default: `NONE`

Example

See the following examples that use ServerTlsOptions:

- Set up IG for HTTPS (server-side)
- Serve different certificates for TLS connections to different server names

RequestResourceUriProvider

Return a resource URL to include in policy decision requests to AM. The PolicyEnforcementFilter uses the returned resource URL as a key to identify cached policy decisions.

To increase performance, use RequestResourceUriProvider in conjunction with AM policies to maximize the the cache hit ratio.

When a request matches a cached policy decision, IG can reuse the decision without asking AM for a new decision. When caching is disabled, IG must ask AM to make a decision for each request.

Usage

```
"resourceUriProvider": {
  "type": "RequestResourceUriProvider",
  "config": {
    "useOriginalUri": configuration expression<boolean>,
    "includeQueryParams": configuration expression<boolean>
  }
}
```

Properties

`useOriginalUri`: *configuration expression<boolean>, optional*

When 'true`, use the value of `UriRouterContext.originalUri` as the resource URL when requesting policy decisions from AM.

When `false`, use the current `Request.uri` value. Consider that the value might have been modified by the `baseURI` of the route or by any other filter executed before the PolicyEnforcementFilter.

Default: `false`

`includeQueryParams`: *configuration expression<boolean>, optional*

When `true`, include query parameters in the resource URL when requesting a policy decision from AM.

When `false`, strip **all** query parameters from the resource URL when requesting a policy decision from AM. To strip some but not all query parameters, use ScriptableResourceUriProvider.

Default: `true`

ScriptableIdentityAssertionPlugin

An out-of-the box implementation of IdentityAssertionPlugin to support use-cases that aren't provided by an IG plugin.

Use with an IdentityAssertionHandler for local processing, such as authentication. The plugin returns IdentityAssertionClaims to include in the identity assertion JWT IG sends to Identity Cloud.

The script does the following:

1. Validates the identity request JWT.
2. (Optional) Takes a single String that represents the principal or a principal and a map of additional claims from the IdentityRequestJwtContext.
3. If a PreProcessingFilter is configured, triggers the filter.
4. Returns principal and identity claims in the identity assertion JWT.

If script execution fails, the plugin creates an IdentityAssertionPluginException.

Usage

```
{
    "name": string,
    "type": "ScriptableIdentityAssertionPlugin",
    "config": {
        "preProcessingFilter": Filter reference,
        "type": configuration expression<string>,
        "file": configuration expression<string>, // Use either "file"
        "source": [ string, ... ],               // or "source", but not both
        "args": map,
        "clientHandler": Handler reference
    }
}
```

Properties

For information about other properties for ScriptableIdentityAssertionPlugin, refer to Scripts.

`"preProcessingFilter"`: *Filter reference, optional*

A Filter to perform user defined actions, such as local authentication and/or authorization.

Default: Pass the request without pre-processing.

Example

The following example applies a `preProcessingFilter` that uses a ScriptableFilter to test whether the user is authenticated. If a Basic Authorization Header isn't found, a response is generated to trigger a Basic Authentication.

```
{
  "name": "BasicAuthScriptablePlugin",
  "type": "ScriptableIdentityAssertionPlugin",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "import org.forgerock.openig.handler.assertion.IdentityAssertionClaims",
      "import org.forgerock.openig.handler.assertion.IdentityAssertionException",
      "if (request.headers.authorization != null && request.headers.authorization.values[0] == 'Basic
user:password') {",
          return new IdentityAssertionClaims("iguser", Map.of("auth", "basic"))",
      "}",
      "return newExceptionPromise(new IdentityAssertionException('Invalid authentication'))",
    ],
    "preProcessingFilter": {
      "type": "ScriptableFilter",
      "config": {
        "type": "application/x-groovy",
        "source": [
          "if (request.headers.authorization == null) {",
          "    Response response = new Response(Status.UNAUTHORIZED)",
          "    response.headers['WWW-Authenticate'] = \"Basic\"",
          "    return response",
          "}",
          "return next.handle(context, request)",
        ],
      },
    }
  }
}
```

More information

org.forgerock.openig.assertion.plugin.IdentityAssertionPlugin

org.forgerock.openig.handler.assertion.IdentityAssertionClaims

ScriptableIdentityAssertionPluginTechPreview

An out-of-the box implementation of IdentityAssertionPluginTechPreview.

Use with an IdentityAssertionHandlerTechPreview for local processing, such as authentication. The plugin returns IdentityAssertionClaims to include in the outgoing JWT sent to Identity Cloud.

> **Important**
>
> The IdentityAssertionHandlerTechPreview, ScriptableIdentityAssertionPluginTechPreview, and IdentityAssertionPluginTechPreview are available in Technology preview. They aren't yet supported, may be functionally incomplete, and are subject to change without notice.

The script must:

- Access the context, request, and claims of an incoming JWT, where the claims are available under the name `incomingClaims`.
- Return an `IdentityAssertionClaims` containing the assertions to add to the outgoing JWT sent to Identity Cloud.

Usage

```
{
    "name": string,
    "type": "ScriptableIdentityAssertionPluginTechPreview",
    "config": {
        "preProcessingFilter": Filter reference,
        "type": configuration expression<string>,
        "file": configuration expression<string>, // Use either "file"
        "source": [ string, ... ],               // or "source", but not both
        "args": map,
        "clientHandler": Handler reference
    }
}
```

Properties

For information about other properties for ScriptableIdentityAssertionPluginTechPreview, refer to Scripts.

`"preProcessingFilter"`: *Filter reference, required*

A Filter to perform user defined actions, such as local authentication and/or authorization. The Filter can be used to process the request before it reaches the script.

Example

The following example applies a `preProcessingFilter` that uses a ScriptableFilter to test whether the user is authenticated. If the user isn't authenticated, the request passes to another script to manage authentication.

```
{
  "name": "BasicAuthScriptablePlugin",
  "type": "ScriptableIdentityAssertionPluginTechPreview",
  "config": {
    "type": "application/x-groovy",
    "source": [
      "import org.forgerock.openig.handler.assertion.IdentityAssertionClaims",
      "import org.forgerock.openig.handler.assertion.IdentityAssertionException",
      "if (request.headers.authorization != null && request.headers.authorization.values[0] == 'Basic
user:password') {",
          return new IdentityAssertionClaims({Map.of("iguser", "user"))",
      "}",
      "return newExceptionPromise(new IdentityAssertionException('Invalid authentication'))",
    ],
    "preProcessingFilter": {
      "type": "ScriptableFilter",
      "config": {
        "type": "application/x-groovy",
        "source": [
          "if (request.headers.authorization == null) {",
          "    Response response = new Response(Status.UNAUTHORIZED)",
          "    response.headers['WWW-Authenticate'] = \"Basic\"",
          "    return response",
          "}",
          "return next.handle(context, request)",
        ],
      },
    }
  }
}
```

More information

org.forgerock.openig.handler.assertion.IdentityAssertionPlugin

IdentityAssertionClaims

TemporaryStorage

Allocates temporary buffers for caching streamed content during request processing. Initially uses memory; when the memory limit is exceeded, switches to a temporary file.

Usage

```
{
  "name": string,
  "type": "TemporaryStorage",
  "config": {
    "initialLength": configuration expression<number>,
    "memoryLimit": configuration expression<number>,
    "fileLimit": configuration expression<number>,
    "directory": configuration expression<string>
  }
}
```

Properties

`"initialLength"`: *configuration expression<number>, optional*

Initial size of the memory buffer.

Default: 8 192 bytes (8 KB). Maximum: The value of `"memoryLimit"`.

`"memoryLimit"`: *configuration expression<number>, optional*

Maximum size of the memory buffer. When the memory buffer is full, the content is transferred to a temporary file.

Default: 65 536 bytes (64 KB). Maximum: 2 147 483 647 bytes (2 GB).

`"fileLimit"`: *configuration expression<number>, optional*

Maximum size of the temporary file. If the file is bigger than this value, IG responds with an OverflowException.

Default: 1 073 741 824 bytes (1 GB). Maximum: 2 147 483 647 bytes (2 GB).

`"directory"`: *configuration expression<string>, optional*

The directory where temporary files are created.

Default: `$HOME/.openig/tmp` (on Windows, `%appdata%\OpenIG\tmp`)

More information

org.forgerock.openig.io.TemporaryStorageHeaplet

TrustAllManager

Blindly trusts all server certificates presented the servers for protected applications. It can be used instead of a TrustManager in test environments to trust server certificates that were not signed by a well-known CA, such as self-signed certificates.

The TrustAllManager is not safe for production use. Use a properly configured TrustManager instead.

Usage

```
{
    "name": string,
    "type": "TrustAllManager"
}
```

Example

The following example configures a client handler that blindly trusts server certificates when IG connects to servers over HTTPS:

```
{
    "name": "BlindTrustClientHandler",
    "type": "ReverseProxyHandler",
    "config": {
        "trustManager": {
            "type": "TrustAllManager"
        }
    }
}
```

More information

org.forgerock.openig.security.TrustAllManager

UmaService

The UmaService includes a list of resource patterns and associated actions that define the scopes for permissions to matching resources. When creating a share using the REST API described below, you specify a path matching a pattern in a resource of the UmaService.

Usage

```
{
    "name": string,
    "type": "UmaService",
    "config": {
        "protectionApiHandler": Handler reference,
        "amService": AmService reference,                    // Use either "amService"
        "wellKnownEndpoint": configuration expression<url>,  // or "wellKnownEndpoint", but not both.
        "resources": [ object, ... ]
    }
}
```

Properties

`"protectionApiHandler"`: *Handler reference, required*

The handler to use when interacting with the UMA Authorization Server to manage resource sets, such as a ClientHandler capable of making an HTTPS connection to the server.

For more information, refer to Handlers.

`"amService"`: *AmService reference, required if* `wellKnownEndpoint` *is not configured*

The AmService heap object to use for the URI to the well-known endpoint for this UMA Authorization Server. The endpoint is extrapolated from the `url` property of the AmService, and takes the realm into account.

If the UMA Authorization Server is AM, use this property to define the endpoint.

If `amService` is configured, it takes precedence over `wellKnownEndpoint`.

For more information, refer to UMA discovery in AM's *User-Managed Access (UMA) 2.0 guide*.

See also AmService.

`"wellKnownEndpoint"`: *configuration expression<url>, required if* `amService` *is not configured*

The URI to the well-known endpoint for this UMA Authorization Server.

If the UMA Authorization Server is not AM, use this property to define the endpoint.

If `amService` is configured, it takes precedence over `wellKnownEndpoint`.

Examples:

- In this example, the UMA configuration is in the default realm of AM:

```
https://am.example.com:8088/openam/uma/.well-known/uma2-configuration
```

- In this example, the UMA configuration is in a European customer realm:

```
https://am.example.com:8088/openam/uma/realms/root/realms/customer/realms/europe/.well-known/uma2-configuration
```

For more information, refer to AM as UMA Authorization Server in AM's *User-Managed Access (UMA) 2.0 guide*.

`"resources"`: *array of objects, required*

Resource objects matching the resources the resource owner wants to share.

Each resource object has the following form:

```
{
    "pattern": resource pattern,
    "actions": [
        {
            "scopes": [ scope string, ... ],
            "condition": runtime expression<boolean>
        },
        {
            ...
        }
    ]
}
```

Each resource pattern can represent an application, or a consistent set of endpoints that share scope definitions. The actions map each request to the associated scopes. This configuration serves to set the list of scopes in the following ways:

1. When registering a resource set, IG uses the list of actions to provide the aggregated, exhaustive list of all scopes that can be used.
2. When responding to an initial request for a resource, IG derives the scopes for the ticket based on the scopes that apply according to the request.
3. When verifying the RPT, IG checks that all required scopes are encoded in the RPT.

A description of each field follows:

`"pattern"`: *pattern*, *required*

A pattern matching resources to be shared by the resource owner, such as `.*` to match any resource path, and `/photos/.*` to match paths starting with `/photos/`.

See also Patterns.

`"actions"`: *array of objects*, *optional*

A set of scopes to authorize when the corresponding condition evaluates to `true`.

```
"actions": [
  {
    "scopes": [ "#read" ],
    "condition": "${request.method == 'GET'}"
  },
  {
    "scopes": [ "#create" ],
    "condition": "${request.method == 'POST'}"
  }
]
```

`"scopes"`: *array of configuration expression<strings>*, *optional*

One or more scopes that are authorized when the corresponding condition evaluates to `true`.

For example, the scope `#read` grants read-access to a resource.

`"condition"`: *runtime expression<boolean>, required*

When the condition evaluates to `true`, the corresponding scope is authorized.

For example, the condition `${request.method == 'GET'}` is true when reading a resource.

REST API for shares

The REST API for UMA shares is exposed at a registered endpoint. IG logs the paths to registered endpoints when the log level is `INFO` or finer. Look for messages such as the following in the log:

```
UMA Share endpoint available at
 '/openig/api/system/objects/_router/routes/00-uma/objects/umaservice/share'
```

To access the endpoint over HTTP or HTTPS, prefix the path with the IG scheme, host, and port to obtain a full URL, such as `http://localhost:8080/openig/api/system/objects/_router/routes/00-uma/objects/umaservice/share`.

The UMA REST API supports create (POST only), read, delete, and query (`_queryFilter=true` only). For an introduction to common REST APIs, refer to About ForgeRock Common REST.

In the present implementation, IG does not have a mechanism for persisting shares. When IG stops, the shares are discarded.

For information about API descriptors for the UMA share endpoint, refer to API descriptors. For information about Common REST, refer to About ForgeRock Common REST.

A share object has the following form:

```
{
    "path": pattern,
    "pat": UMA protection API token (PAT) string,
    "id": unique identifier string,
    "resource_id": unique identifier string,
    "user_access_policy_uri": URI string
}
```

`"path"`: *pattern, required*

A pattern matching the path to protected resources, such as `/photos/.*`.

This pattern must match a pattern defined in the UmaService for this API.

See also Patterns.

`"pat"`: *PAT string, required*

A PAT granted by the UMA Authorization Server given consent by the resource owner.

In the present implementation, IG has access only to the PAT, not to any refresh tokens.

`"id"`: *unique identifier string, read-only*

This uniquely identifies the share. This value is set by the service when the share is created, and can be used when reading or deleting a share.

`"resource_id"`: *unique identifier string, read-only*

This uniquely identifies the UMA resource set registered with the authorization server. This value is obtained by the service when the resource set is registered, and can be used when setting access policy permissions.

`"user_access_policy_uri"`: *URI string, read-only*

This URI indicates the location on the UMA Authorization Server where the resource owner can set or modify access policies. This value is obtained by the service when the resource set is registered.

More information

User-Managed Access (UMA) 2.0 Grant for OAuth 2.0 Authorization

org.forgerock.openig.uma.UmaSharingService

Property value substitution

In an environment with multiple IG instances, you can require similar but not identical configurations across the different instances.

Property value substitution enables you to do the following:

- Define a configuration that is specific to a single instance, for example, setting the location of the keystore on a particular host.
- Define a configuration whose parameters vary between different environments, for example, the URLs and passwords for test, development, and production environments.
- Disable certain capabilities on specific nodes.

Property value substitution uses *configuration tokens* to introduce variables into the server configuration. For information, refer to Configuration Tokens.

The substitution follows a process of token resolution, JSON evaluation, and data transformation, as described in the following sections:

- JSON Evaluation
- Token Resolution
- Transformations

Configuration Tokens

A configuration token is a simple reference to a value. When configuration tokens are resolved, the result is always a string. Transformation described in Transformations can be used to coerce the output type.

Configuration Tokens for File System

IG provides `ig.instance.dir` and `ig.instance.url` to define the file system directory and URL for configuration files.

Their values are computed at startup, and evaluate to a directory such as `$HOME/.openig` (`%appdata%\OpenIG`). You can use these tokens in your configuration without explicitly setting their values.

For information about how to change the default values, refer to Configuration location.

Syntax

Configuration tokens follow the syntax `&{token[|default]}`, as follows:

- Are preceded by an ampersand, `&`
- Are enclosed in braces, `{}`
- Define default values with a vertical bar (`|`) after the configuration token
- Are in lowercase
- Use the period as a separator, `.`

When a configuration token is supplied in a configuration parameter, it is always inside a string enclosed in quotation marks, as shown in the following example:

```
"&{listen.port|8080}"
```

To escape a string with the syntax of a configuration token, use a backslash (`\`). The following string is treated as normal text:

```
"\&{listen.port|8080}"
```

A configuration property can include a mix of static values and expressions, as shown in the following example:

```
"&{hostname}.example.com"
```

Configuration tokens can be nested inside other configuration tokens as shown in the following example:

```
"&{&{protocol.scheme}.port}"
```

Default values or values in the property resolver chain can be nested, as shown in the following example:

```
"&{&{protocol.scheme|http}.port|8080}"
```

JSON Evaluation

JSON evaluation is the process of substituting configuration tokens and transforming JSON nodes for an entire JSON configuration. After JSON evaluation, all configuration tokens and transformations in the configuration are replaced by values.

At startup, IG evaluates the configuration tokens in `config.json` and `admin.json`. When routes are deployed, IG evaluates the configuration tokens in the route.

Configuration tokens are matched with tokens available in the chain of resolvers, and the configuration token is substituted with the value available in the resolver. For information about each of the resolvers mentioned in the following section, refer to Token Resolution.

IG searches for matching tokens in the chain of resolvers, using the following order of precedence:

1. Local resolver:

   The route resolver for the route being deployed

2. Intermediate resolver:

   All intermediate route resolvers (for example, for parent routes to the route being deployed) up to the bootstrap resolver

3. Bootstrap resolver:

   1. Environment variables resolver
   2. System properties resolver
   3. Token source file resolvers
   4. Hardcoded default values

The first resolver that matches the token returns the value of the token.

If the token can't be resolved, IG uses the default value defined with the configuration token. If there is no default value, the token can't be resolved and an error occurs:

- If the configuration token is in `config.json` or `admin.json`, IG fails to start up.
- If the configuration token is in a route, the route fails to load.

When configuration tokens are nested inside other configuration tokens, the tokens are evaluated bottom-up, or leaf-first. For example, if the following configuration token takes only the default values, it is resolved as follows:

1. `"&{&{protocol.scheme|http}.port|8080}"`
2. `"&{http.port|8080}"`

   When `&{protocol.scheme|http}` takes the default value `http`.

3. `"8080"`

   When `&{http.port|8080}` takes the default value `8080`.

If the configuration includes a transformation, IG applies the transformation after the token is substituted. When transformations are nested inside other transformations, the transformations are applied bottom-up, or leaf-first. For more information, refer to Transformations.

Token Resolution

At startup, the bootstrap resolver builds a chain of resolvers to resolve configuration tokens included in `config.json` and `admin.json`. When a route is deployed, *route resolvers* build on the chain to add resolvers for the route.

Route Token Resolvers

When a route is deployed in IG a route resolver is created to resolve the configuration tokens for the route. The resolvers uses token values defined in the `properties` section of the route.

If the token can't be resolved locally, the route resolver accesses token values recursively in a parent route.

For more information, about route properties, refer to Route properties.

Environment Variables Resolver

When the bootstrap resolver resolves a configuration token to an environment variable, it replaces the lowercase and periods (`.`) in the token to match the convention for environment variables.

Environment variable keys are transformed as follows:

- Periods (.) are converted to underscores
- All characters are transformed to uppercase

The following example sets the value of an environment variable for the port number:

```
$ export LISTEN_PORT=8080
```

In the following IG configuration, the value of `port` is `8080`:

```
{
  "port": "&{listen.port}"
}
```

System Properties Resolver

The system property name must match a configuration token exactly. The following example sets a system property for a port number:

```
$ java -Dlisten.port=8080 -jar start.jar
```

In the following IG configuration, the value of `port` is `8080`:

```
{
  "port": "&{listen.port}"
}
```

Token Source File Resolvers

Token source files have the `.json` or `.properties` extension. The bootstrap resolver uses the files to add file resolvers to the chain of resolvers:

- **JSON file resolvers**

Token source files with the `.json` extension take a JSON format. The token name is mapped either to the JSON attribute name or to the JSON path.

Each of the following `.json` files set the value for the configuration token `product.listen.port`:

```
{
  "product.listen.port": 8080
}
```

```
{
  "product.listen": {
     "port": 8080
  }
}
```

```
{
  "product": {
    "listen": {
      "port": 8080
    }
  }
}
```

- **Properties file resolvers**

  Token source files with the `.properties` extension are Java properties files. They contain a flat list of key/value pairs, and keys must match tokens exactly.

  The following `.properties` file also sets the value for the tokens `listen.port` and `listen.address`:

```
listen.port=8080
listen.address=192.168.0.10
```

Token source files are stored in one or more directories defined by the environment variable `IG_ENVCONFIG_DIRS` or the system property `ig.envconfig.dirs`.

If token source files are in multiple directories, each directory must be specified in a comma-separated list. IG doesn't scan subdirectories. The following example sets an environment variable to define two directories that hold token source files:

```
$ export IG_ENVCONFIG_DIRS="/myconfig/directory1,/myconfig/directory2"
```

At startup, the bootstrap resolver scans the directories in the specified order, and adds a resolver to the chain of resolvers for each token source file in the directories.

Although the bootstrap resolver scans the directories in the specified order, within a directory it scans the files in a nondeterministic order.

Note the following constraints for using the same configuration token more than once:

- Do not define the same configuration token more than once in a single file. There is no error, but you won't know which token is used.
- Do not define the same configuration token in more than one file in a single directory. An error occurs.

> **Important**
>
> This constraint implies that you can't have backup `.properties` and `.json` files in a single directory if they define the same tokens.

- You can define the same configuration token once in several files that are located in different directories, but the first value that IG reads during JSON evaluation is used.

> **Note**
>
> When logging is enabled at the DEBUG level for token resolvers, the origin of the token value is logged.
> If you are using the default logback implementation, add the following line to your `logback.xml` to enable logging:
>
> ```
> <logger name="org.forgerock.config.resolvers" level="DEBUG" />
> ```

Transformations

A set of built-in transformations are available to coerce strings to other data types. The transformations can be applied to any string, including strings resulting from the resolution of configurations tokens.

After transformation, the JSON node representing the transformation is replaced by the result value.

The following sections describe how to use transformations, and describe the transformations available:

Usage

```
{
   "$transformation": string or transformation
}
```

A transformation is a JSON object with a required main attribute, starting with a `$`. The following example transforms a string to an integer:

```
{"$int": string}
```

The value of a transformation value can be a JSON string or another transformation that results in a string. The following example shows a nested transformation:

```
{
  "$array": {
    "$base64:decode": string
  }
}
```

The input string must match the format expected by the transformation. In the previous example, because the final transformation is to an array, the input string must be a string that represents an array, such as `"[ \"one\", \"two\" ]"`.

In the first transformation, the encoded string is transformed to a base64-decoded string. In the second, the string is transformed into a JSON array, for example, `[ "one", "two" ]`.

array

```
{"$array": string}
```

Returns a JSON array of the argument.

Argument	Returns
string String representing a JSON array.	***array*** JSON array of the argument.

The following example transformation results in the JSON array `[ "one", "two" ]`:

```
{"$array": "[ \"one\", \"two\" ]"}
```

bool

```
{"$bool": string}
```

Returns `true` if the input value equals `"true"` (ignoring case). Otherwise, returns `false`.

Argument	Returns
string String containing the boolean representation.	***boolean*** Boolean value represented by the argument.

If the configuration token `&{capture.entity}"` resolves to `"true"`, the following example transformation results in the value `true`:

```
{"$bool": "&{capture.entity}"}
```

decodeBase64

```
{
  "$base64:decode": string,
  "$charset": "charset"
}
```

Transforms a base64-encoded string into a decoded string. If `$charset` is specified, the decoded value is interpreted with the character set.

Argument	Parameters	Returns
string Base64-encoded string.	***$charset*** The name of a Java character set, as described in Class Charset.	***string*** Base64-decoded string in the given character set.

The following example transformation returns the `Hello` string:

```
{
 "$base64:decode": "SGVsbG8=",
 "$charset": "UTF-8"
}
```

encodeBase64

```
{
  "$base64:encode": string,
  "$charset": "charset"
}
```

Transforms a string into a base64-encoded string. Transforms to `null` if the string is `null`.

If `$charset` is specified, the string is encoded with the character set.

Argument	Parameters	Returns
string String to encode with the given character set.	***$charset*** The name of a Java character set, as described in Class Charset.	***string*** Base64-encoded string.

int

```
{"$int": string}
```

Transforms a string into an integer.

If the parameter is not a valid number in radix 10, returns `null`.

Argument	Returns
string String containing the integer representation.	***int*** Integer value represented by the argument.

The following example transformation results in the integer `1234`:

```
{"$int": "1234"}
```

list

```
{"$list": string}
```

Transforms a comma-separated list of strings into a JSON array of strings

Argument	Returns
string A string representing a comma-separated list of strings.	***array*** The JSON array of the provided argument. Values are not trimmed of leading spaces.

The following example transformation results in the array of strings `["Apple","Banana","Orange","Strawberry"]`:

```
{"$list": "Apple,Banana,Orange,Strawberry"}
```

The following example transformation results in the array of strings `["Apple"," Banana"," Orange"," Strawberry"]`, including the untrimmed spaces:

```
{"$list": "Apple, Banana, Orange, Strawberry"}
```

The following example transformation results in the array of strings `["1","2","3","4"]`, and not an array of JSON numbers `[1,2,3,4]`:

```
{"$list": "1,2,3,4"}
```

number

```
{"$number": string}
```

Transform a string into a Java number, as defined in Class Number.

Argument	Returns
strings A string containing the number representation.	***number*** The number value represented by the argument.

The following example transformation results in the number `0.999`:

```
{"$number": ".999"}
```

object

```
{"$object": string}
```

Transforms a string representation of a JSON object into a JSON object.

Argument	Returns
string String representation of a JSON object.	***object*** JSON object of the argument.

The following example transformation

```
{"$object": "{\"ParamOne\":{\"InnerParamOne\":\"InnerParamOneValue\",\"InnerParamTwo\": false}}"}
```

results in the following JSON object:

```
{
  "ParamOne": {
    "InnerParamOne": "myValue",
    "InnerParamTwo": false
  }
}
```

string

```
{"$string": placeholder string}
```

Transforms a string representation of a JSON object into a placeholder string. Placeholder strings are not encrypted.

Use this transformation for placeholder strings that that must not be encrypted.

Argument	Returns
string String representation of a JSON object.	***placeholder string*** Placeholder string.

This example transformation:

```
{
  "someAttributeExpectingString": { "$string": "&{ig.instance.dir}" }
}
```

results in this JSON object:

```
{
  "someAttributeExpectingString": "/path/to/ig"
}
```

Expressions

Use expressions that conform to the Unified Expression Language in JSR-245 to specify configuration parameters as expressions in routes. The result of an expression must match the expected type. For examples of expressions used in routes, refer to Examples.

Use expressions in routes for the following tasks:

Evaluate object properties

The following example returns the URI of the incoming request:

```
${request.uri}
```

Call functions decribed in Functions

Functions can be operands for operations and can yield parameters for other function calls.

The following example uses the function `find` to test whether the request URI is on the `/home` path:

```
${find(request.uri.path, '^/home')}
```

Call Java methods

The following example uses the method Java `String.startsWith()` to test whether the request starts with `/home`:

```
${request.uri.path.startsWith("/home")}
```

Retrieve Java system properties

The following example yields the home directory of the user executing the IG process:

```
${system['user.home']}
```

Retrieve environment variables

The following example yields the home directory of the user executing the IG process:

```
${env['HOME']}
```

The following example yields the path value of the `keystore.p12` file in the user's home directory. The result is just a String concatenation, there is no verification that the file actually exists:

```
${env['HOME']}/keystore.p12`
```

Perform logical operations such as `and`, `or`, *and* `not`

The following expression uses the operators `and` and `or` to determine where to dispatch a request. The expression is used in Share JWT sessions between multiple instances of IG:

```
${find(request.uri.path, '/webapp/browsing') and (contains(request.uri.query, 'one') or
empty(request.uri.query))}
```

Perform arbitrarily complex arithmetic, such as addition, substraction, division, and multiplication

The following expression yields the URI port number incremented by four:

```
${request.uri.port + 4}
```

Perform relational operations, such as numerical equality and inequality

The following example is for a DispatchHandler condition, where the request is dispatched if it contains a form field with the attribute `answer` whose value is greater than `42`:

```
"bindings": [
  {
    "condition": "#{request.entity.form['answer'] > 42}",
    "handler": ...
  }
]
```

Perform conditional operations of this form `<condition> ? <if-true> : <if-false>`

The following example tests whether the request path starts with `/home`. If so, the request is directed to `home`; otherwise, it is directed to `not-home`:

```
${request.uri.path.startsWith('/home') ? 'home' : 'not-home' }
```

Consume evaluated configuration tokens described in JSON Evaluation (runtime expressions only)

The following example returns `true` if the configuration token `my.status.code` resolves to `200`:

```
${integer(_token.resolve('my.status.code', '404')) == 200}
```

Expressions access the environment through the implicit object `openig`. The object has the following properties:

- `instanceDirectory`: Path to the base location for IG files. The default location is:

Linux

```
$HOME/.openig
```

Windows

```
%appdata%\OpenIG\config
```

- `configDirectory`: Path to the IG configuration files. The default location is:

Linux

```
$HOME/.openig/config
```

Windows

```
%appdata%\OpenIG\config
```

- `temporaryDirectory`: Path to the IG temporary files. The default location is:

Linux

```
$HOME/.openig/tmp
```

Windows

```
%appdata%\OpenIG\tmp
```

To change default values, refer to Change the base location of the IG configuration.

Syntax

Expression syntax must conform to Unified Expression Language described in JSR-245.

Route syntax

Expressions in routes are enclosed in quotation marks, for example: `"${request.method}"` and `"#{expression}"`.

Immediate and deferred evaluation syntax

Use the `${}` syntax for expressions to be evaluated at startup or when a route loads or reloads.

Use the `#{}` syntax for expressions to be evaluated later, when the evaluation result requires the request or response to be fully loaded.

Operator syntax

AM uses the `.` and `[]` operators to access properties:

- Use `.` for the following tasks:
  - Access object properties, such as public fields and Java bean properties.
  - Invoke methods on an object.
  - Access map entries when the specified entry name doesn't contain reserved characters.
- Use `[]` to access indexed elements:
  - If the object is a collection, such as an array, set, or list, use `[i]` to access an element at position `i`.
  - If the object isn't a collection, such as an array, set, or list, use `['prop']` to access the property `'prop'`. This is equivalent to `.` notation. The following expressions are equivalent:

    ```
    ${request.method}
    ```

    ```
    ${request['method']}
    ```

  - If the object is a map, use `['name']` to access the entry with name `'name'`.

Tip

To access map entries containing characters that are also expression operators, prevent parsing exceptions by using `[]` instead of `.`
For example, to access a map field containing a dash `-`, such as `dash-separated-name`, write the expression as:

```
${data['dash-separated-name']}
```

instead of:

```
${data.dash-separated-name}
```

In the second example, the dash `-` is interpreted as part of the String instead of as an expression operator.

Array syntax

The index of an element in an array is expressed as a number in brackets. For example, the following expression refers to the first `Content-Type` header value in a request:

```
${request.headers['Content-Type'][0]}
```

Map syntax

The map key is expressed in brackets. For example, the following expression is an example of Map entry access:

```
system['prop.name']
```

If a property doesn't exist, the index reference yields a `null` (empty) value.

Function syntax

Expressions can call built-in functions described in Functions.

Use the syntax `${function(parameter, …)}`, supplying one or more parameters to the function. For examples, refer to Expressions that use functions.

Functions can be operands for operations and can yield parameters for other function calls.

Method syntax

Use the syntax `${object.method()}` to call a method on the object instance.

If the object resolves to a String, then all methods in the `java.lang.String` class are usable.

For examples, refer to Expressions that use functions.

Escape syntax

The character `\` is treated as an escape character when followed by `${` or `\#{`.

For example, the expression `${true}` normally evaluates to `true`. To include the string `${true}` in an expression, write `\${true}`

When `\` is followed by any other character sequence, it isn't treated as an escape character.

Configuration expressions

IG evaluates configuration expressions at startup and when a route loads or reloads. Configuration expressions are always evaluated immediately and use the `${}` syntax.

Configuration expressions can refer to the following information:

- System heap properties
- Built-in functions listed in Functions
- Environment variables, `${env['variable']}`
- System properties, `${system['property']}`
- ExpressionInstant

Because configuration expressions are evaluated before requests are made, they can't refer to the runtime properties `request`, `response`, `context`, or `contexts`.

Runtime expressions

IG evaluates runtime expressions for each request and response, as follows:

Immediate evaluation of runtime expressions

If the expression consumes streamed content, for example, the content of a request or response, IG blocks the executing thread until all the content is available.

Runtime expressions whose evaluation is immediate are written with the `${}` syntax.

Deferred evaluation of runtime expressions

IG waits until all streamed content is available before it evaluates the expression. Deferred evaluation doesn't block executing threads.

Runtime expressions whose evaluation is deferred are written with the `#{}` syntax.

When the `streamingEnabled` property in `admin.json` is `true`, expressions that consume streamed content **must** be written with `#` instead of `$`.

Runtime expressions can refer to the following information:

- System heap properties
- Built-in functions listed in Functions
- Environment variables
- System properties
- ExpressionInstant

- `attributes`: org.forgerock.services.context.AttributesContext Map<String, Object>, obtained from `AttributesContext.getAttributes()`. For information, refer to AttributesContext.
- `context`: org.forgerock.services.context.Context object.
- `contexts`: map<string, context> object. For information, refer to Contexts.
- `request`: org.forgerock.http.protocol.Request object. For information, refer to Request.
- `response`: org.forgerock.http.protocol.Response object, available only when the expression is intended to be evaluated on the response flow. For information, refer to Response.
- `session`: org.forgerock.http.session.Session object, available only when the expression is intended to be evaluated for both request and response flow. For information, refer to SessionContext.

Embedded expressions

Consider the following points when embedding expressions:

- System properties, environment variables, or function expressions can be embedded within expressions.

  The following example embeds an environment variable in the argument for a `read()` function. The value of `entity` is set to the contents of the file `$HOME/.openig/html/defaultResponse.html`, where `$HOME/.openig` is the instance directory:

  ```
  "entity": "${read('&{ig.instance.dir}/html/defaultResponse.html')}"
  ```

- Expressions can't be embedded inside other expressions, as `${expression}`.
- Embedded elements can't be enclosed in `${}`.

Extensions

IG offers a plugin interface for extending expressions. See Key extension points.

L-value expressions

L-value expressions assign a value to the expression scope. For example, `"${session.gotoURL}"` assignes a value to the session attribute named `gotoURL`.

IG ignores attempts to write to read-only values.

L-value expressions must be specified using immediate evaluation syntax; use `$` instead of `#`.

Operators

The following operators are provided by Unified Expression Language:

- Index property value: `[]`, `.`
- Change precedence: `()`

- Arithmetic: `+` (binary), `-` (binary), `*`, `/`, `div`, `%`, `mod`, `-` (unary)
- Logical: `and`, `&&`, `or`, `||`, `not`, `!`
- Relational: `==`, `eq`, `!=`, `ne`, `<`, `lt`, `>`, `gt`, `⇐`, `le`, `>=`, `ge`
- Empty: `empty`
  Use this prefix operator to determine whether a value is null or empty.
- Conditional: `?`, `:`

Operators have the following precedence, from highest to lowest, and from left to right:

- `[]` `.`
- `()`
- `-` (unary) `not` `!` `empty`
- `*` `/` `div` `%` `mod`
- `+` (binary) `-` (binary)
- `<` `>` `⇐` `>=` `lt` `gt` `le` `ge`
- `==` `!=` `eq` `ne`
- `&&` `and`
- `||` `or`
- `?` `:`

Dynamic bindings

Configuration and runtime expressions can use ExpressionInstant.

The current instant is `${now.epochSeconds}`.

To add or subtract a period of time to the instant, add one or more of the following time periods to the binding:

- `plusMillis(integer)`, `minusMillis(integer)`
- `plusSeconds(integer)`, `minusSeconds(integer)`
- `plusMinutes(integer)`, `minusMinutes(integer)`
- `plusHours(integer)`, `minusHours(integer)`
- `plusDays(integer)`, `minusDays(integer)`

The following example binding refers to 30 minutes after the current instant:

```
${now.plusMinutes(30).epochSeconds}
```

The following example binding accesses the instant in RFC 1123 date format one day after the current instant:

```
${now.plusDays(1).rfc1123}
```

For more examples, refer to the `template` property of JwtBuilderFilter and the `attribute-name` property of SetCookieUpdateFilter.

Examples

Immediate evaluation of configuration expressions

The following example yields the value of a secret from a system property.

```
{
  "passwordSecretId": "${system['keypass']}"
}
```

The following example yields a file from the home directory of the user running the IG application server.

```
{
  "url": "file://${env['HOME']}/keystore.p12"
}
```

The following example of a `temporaryStorage` object takes the value of the system property `storage.ref`, which must a be string equivalent to the name of an object defined in the heap:

```
{
  "temporaryStorage": "${system['storage.ref']}"
}
```

Deferred evaluation of runtime expressions

The following example is a Route condition, where the Route is accessed if the request contains json with the attribute `answer`, whose value is `42`.

IG defers evaluation of the expression until it receives the entire body of the reqest, transfoms it to JSON view, and then introspects it for the attribute `answer`.

```
{
  "condition": "#{request.entity.json['answer'] == 42}",
  "handler": ...
}
```

The following example expression is for a JwtBuilderFilter that uses the content of the request mapped as a string. IG defers evaluation of the expression until it receives the entire body of the request:

```
{
  "template": {
    "content": "#{request.entity.string}"
  }
}
```

Immediate and deferred evaluation of runtime expressions

The following example expressions are for an AssignmentFilter that consumes an ID captured from a response.

IG evaluates the first expression immediately to define the target.

It then defers evaluation of the second expression until it receives the entire body of the response.

```
"onResponse": [
  {
    "target": "${response.headers['X-IG-FooBar']}",
    "value": "#{toString(response.entity.json['userId'])}"
  }
]
```

Expressions that use functions

In the following example, `"timer"` is defined by an expression that recovers the environment variable `"ENABLE_TIMER"` and transforms it into a boolean. Similarly, `"numberOfRequests"` is defined by an expression that recovers the system property `"requestsPerSecond"` and transforms it into an integer:

```
{
  "name": "throttle-simple-expressions1",
  "timer": "${bool(env['ENABLE_TIMER'])}",
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/throttle-simple-expressions1')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ThrottlingFilter",
          "name": "ThrottlingFilter-1",
          "config": {
            "requestGroupingPolicy": "",
            "rate": {
              "numberOfRequests": "${integer(system['requestsPerSecond'])}",
              "duration": "10 s"
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

If `"requestsPerSecond"=6` and `"ENABLE_TIMER"=true`, after the expressions are evaluated IG views the example route as follows:

```
{
  "name": "throttle-simple-expressions2",
  "timer": true,
  "baseURI": "http://app.example.com:8081",
  "condition": "${find(request.uri.path, '^/home/throttle-simple-expressions2')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ThrottlingFilter",
          "name": "ThrottlingFilter-1",
          "config": {
            "requestGroupingPolicy": "",
            "rate": {
              "numberOfRequests": 6,
              "duration": "10 s"
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

Functions

A set of built-in functions that can be called from within Expressions.

array

`array(strings...)`

Returns an array of the strings given as argument.

Parameters

strings

Strings to put in the array.

Returns

array

Resulting array of containing the given strings.

boolean

`bool(string)`

Returns a Boolean with a value represented by the specified string.

The returned Boolean represents a true value if the string argument is not `null` and is equal to the string `"true"`, ignoring case.

Parameters

string

String containing the boolean representation.

Returns

Boolean

Boolean value represented by the string.

contains

`contains(object, value)`

Returns `true` if the object contains the specified value. If the object is a string, a substring is searched for the value. If the object is a collection or array, its elements are searched for the value.

Parameters

object

Object to search for.

value

Value to search for.

Returns

true

If the object contains the specified value.

decodeBase64

`decodeBase64(string)`

Returns the base64-decoded string, or `null` if the string is not valid base64.

Parameters

string

Base64-encoded string to decode.

Returns

string

Base64-decoded string.

decodeBase64url

`decodeBase64url(string)`

Returns the decoded value of the provided base64url-encoded string, or `null` if the string was not valid base64url.

Parameters

string

Base64url-encoded string to decode.

Returns

string

Base64url-decoded string.

digestSha256

`digestSha256(byte array or string)`

Calculates the SHA-256 hash of an incoming object.

Parameters

byte array or string

The bytes to be hashed. If a string is provided, this function uses the UTF-8 charset to get the bytes from the string.

Returns

byte array

SHA-256 hash as a byte array, or null if the hash could not be calculated.

encodeBase64

`encodeBase64(string)`

Returns the base64-encoded string, or `null` if the string is `null`.

Parameters

string

String to encode into base64.

Returns

string

Base64-encoded string.

encodeBase64url

`encodeBase64url(string)`

Returns the base64url-encoded string, or `null` if the string is `null`.

Parameters

string

String to encode into base64url.

Returns

string

Base64url-encoded string.

fileToUrl

`fileToUrl(file)`

Converts a `java.io.File` into a string representation for the URL of the file or directory.

Parameters

file

File or directory for which to build the URL.

For example, `${fileToUrl(openig.configDirectory)}/myProperties.json`.

Returns

file

String representation for the URL of the file or directory, or `null` if the file or directory is `null`.

For example, `file:///home/gcostanza/.openig/config/myProperties.json`.

find

`find(string, pattern)`

Attempts to find the next subsequence of the input string that matches the pattern.

Parameters

string

The input string.

pattern

A regular expression pattern.

Returns

boolean

- `true` if a subsequence of the input string matches the regular expression pattern.
- `false` if the input string is null, or a subsequence of it does not match the regular expression pattern.

findGroups

```
findGroups(string, pattern)
```

Attempts to find a string that matches the regular expression or groups specified in the regular expression.

Parameters

string

The input on which the regular expression is applied.

pattern

A regular expression pattern.

Returns

array

An array containing the result of a find on the regular expression against the input string, or null if no result is found.

The first element of the array is the entire match, and each subsequent element correlates to a capture group specified in the regular expression.

formDecodeParameterNameOrValue

```
formDecodeParameterNameOrValue(string)
```

Returns the string that results from decoding the provided form encoded parameter name or value as per `application/x-www-form-urlencoded`, which can be `null` if the input is `null`.

Parameters

string

Parameter name or value.

Returns

string

String resulting from decoding the provided form encoded parameter name or value as per `application/x-www-form-urlencoded`.

formEncodeParameterNameOrValue

`formEncodeParameterNameOrValue(string)`

Returns the string that results from form encoding the provided parameter name or value as per `application/x-www-form-urlencoded`, which can be `null` if the input is `null`.

Parameters

string

Parameter name or value.

Returns

string

String resulting from form encoding the provided parameter name or value as per `application/x-www-form-urlencoded`.

indexOf

`indexOf(string, substring)`

Returns the index of the first instance of a specified substring inside a string.

Characters in the provided string are UTF-16, based on 16-bit code units. Each character is encoded as at least two bytes, and some extended characters are encoded as four bytes.

When this function processes a 2-byte character, it counts it as one 16-bit character. When it processes a 4-byte character, it counts it as two 16-bit characters.

Examples:

- The unicode character `a` (U+0061) has the UTF-16 value `0x0061`. The function `{{indexOf('afooBar', 'Bar')}}` evaluates to `4`.
- The unicode character 𐀗 (U+10057) has the UTF-16 value `0xD800 0xDC57`. The function `{{indexOf('𐀗 fooBar', 'Bar')}}` evaluates to `5`.

Parameters

string

String in which to search for the specified substring.

substring

Value to search for within the string.

Returns

number

Index of the first instance of the substring, or -1 if not found.

The index count starts from 1, not 0.

integer

```
integer(string)
```

Transforms the string parameter into an integer. If the parameter is not a valid number in radix 10, returns null.

Parameters

string

String containing the integer representation.

Returns

integer

Integer value represented by the string.

integerWithRadix

```
integer(string, radix)
```

Uses the radix as the base for the string, and transforms the string into a base-10 integer. For example:

- `("20", 8)`: Transforms `20` in base `8`, and returns `16`.
- `("11", 16)` Transforms `11` in base `16`, and returns `17`.

If either parameter is not a valid number, returns null.

Parameters

string

String containing the integer representation, and an integer containing the radix representation.

Returns

integer

Integer value in base-10.

ipMatch

```
ipMatch(string, string)
```

Returns `true` if the provided IP address matches the range provided by the Classless Inter-Domain Routing (CIDR), or `false` otherwise.

Parameters

string

IP address of a request sender, in IPv4 or IPv6

string

CIDR defining an IP address range

Returns

Boolean

`true` or `false`

join

```
join(values, separator)
```

Returns a string joined with the given separator, using either of the following values:

- Array of strings (`String[]`)
- Iterable value (`Iterable<String>`)

The function uses the `toString` result from each value.

Parameters

separator

Separator to place between joined values.

strings

Array of values to be joined.

Returns

string

String containing the joined values.

keyMatch

```
keyMatch(map, pattern)
```

Returns the first key found in a map that matches the specified regular expression pattern, or `null` if no such match is found.

Parameters

map

Map whose keys are to be searched.

pattern

String containing the regular expression pattern to match.

Returns

string

First matching key, or `null` if no match found.

length

`length(object)`

Returns the number of items in a collection, or the number of characters in a string.

Characters in the provided string are UTF-16, based on 16-bit code units. Each character is encoded as at least two bytes, and some extended characters are encoded as four bytes.

When this function processes a 2-byte character, it counts it as one 16-bit character. When it processes a 4-byte character, it counts it as two 16-bit characters.

Examples:

- The unicode character `a` (U+0061) has the UTF-16 value `0x0061`. The function `{{length('a')}}` evaluates to `1`.
- The unicode character 𐀗 (U+10057) has the UTF-16 value `0xD800 0xDC57`. The function `{{length('𐀗')}}` evaluates to `2`.

Parameters

object

A collection or string, whose length is to be determined.

Returns

number

Length of the collection or string, or 0 if length could not be determined.

matches (deprecated)

> **Important**
>
> This function is deprecated. Use the matchesWithRegex or find function instead. For more information, refer to the Deprecated section of the *Release Notes*.

`matches(string, pattern)`

Returns `true` if the string contains a match for the specified regular expression pattern.

Parameters

string

The input string.

pattern

A regular expression pattern.

Returns

true

String contains the specified regular expression pattern.

matchesWithRegex

```
matchesWithRegex(string, pattern)
```

Attempts to match the entire input string against the regular expression pattern.

Parameters

string

The input string.

pattern

A regular expression pattern.

Returns

boolean

- `true` if the entire input string matches the regular expression pattern.
- `false` if the input string is null, or the entire input string does not match the regular expression pattern.

matchingGroups (deprecated)

> **Important**
>
> This function is deprecated. Use the findGroups function instead. For more information, refer to the Deprecated section of the *Release Notes*.

```
matchingGroups(string, pattern)
```

Returns an array of matching groups for the specified regular expression pattern applied to the specified string, or `null` if no such match is found. The first element of the array is the entire match, and each subsequent element correlates to any capture group specified within the regular expression.

Parameters

string

String to be searched.

pattern

String containing the regular expression pattern to match.

Returns

array

Array of matching groups, or `null` if no such match is found.

pathToUrl

`pathToUrl(path)`

Converts the given path into the string representation of its URL.

Parameters

path

Path of a file or directory as a string.

For example, `${pathToUrl(system['java.io.tmpdir'])}`.

Returns

string

String representation for the URL of the path, or `null` if the path is `null`.

For example, `file:///var/tmp`.

pemCertificate

`(string)`

Convert the incoming character sequence into a certificate.

Parameters

string

Character sequence representing a PEM-formatted certificate

Returns

string

A `Certificate` instance, or null if the function failed to load a certificate from the incoming object.

read

`read(string)`

Takes a file name as a string, interprets the content of the file with the UTF-8 character set, and returns the content of the file as a plain string.

Provides the absolute path to the file, or a path relative to the location of the Java system property `user.dir`.

Parameters

string

Name of the file to read.

Returns

string

Content of the file, or `null` on error.

readProperties

`readProperties(string)`

Takes a Java Properties filename as a `string`, and returns the content of the file as a key/value map of properties, or `null` on error (due to the file not being found, for example).

Java properties files are expected to be encoded with ISO-8859-1. Characters that cannot be directly represented in ISO-8859-1 can be written using Unicode escapes, as defined in Unicode Escapes, in *The Java™ Language Specification*.

Parameters

string

The absolute path to the Java properties file, or a path relative to the location of the Java system property `user.dir`.

For example, to return the value of the `key` property in the Java properties file `/path/to/my.properties`, provide `${readProperties('/path/to/my.properties')['key']}`.

Returns

object

Key/value map of properties or `null` on error.

readWithCharset

`readWithCharset(string, string)`

Takes a file name as a string, interprets the content of the file with the specified Java character set, and returns the content of the file as a plain string.

Parameters

filename

Name of the file to read.

Provides the absolute path to the file, or a path relative to the location of the Java system property `user.dir`.

charsetName

Name of a Java character set with which to interpret the file, as described in Class Charset.

Returns

string

Content of the file, or `null` on error.

split

```
split(string, pattern)
```

Splits the specified string into an array of substrings around matches for the specified regular expression pattern.

Parameters

string

String to be split.

pattern

Regular expression to split substrings around.

Returns

array

Resulting array of split substrings.

toJson

```
toJson(JSON string)
```

Converts a JSON string to a JSON structure.

Parameters

JSON string

JSON string representing a JavaScript object.

For example, the string value contained in `contexts.amSession.properties.userDetails` contains the JSON object `{"email":"test@example.com"}`.

Returns

JSON structure

JSON structure, or `null` on error.

In the expression `"${toJson(contexts.amSession.properties.userDetails) .email}"`, the string value is treated as JSON, and the expression evaluates to `test@example.com`.

toLowerCase

`toLowerCase(string)`

Converts all of the characters in a string to lowercase.

Parameters

string

String whose characters are to be converted.

Returns

string

String with characters converted to lowercase.

toString

`toString(object)`

Returns the string value of an arbitrary object.

Parameters

object

Object whose string value is to be returned.

Returns

string

String value of the object.

toUpperCase

`toUpperCase(string)`

Converts all of the characters in a string to upper case.

Parameters

string

String whose characters are to be converted.

Returns

string

String with characters converted to upper case.

trim

`trim(string)`

Returns a copy of a string with leading and trailing whitespace omitted.

Parameters

string

String whose white space is to be omitted.

Returns

string

String with leading and trailing white space omitted.

urlDecode

`urlDecode(string)`

Returns the URL decoding of the provided string.

This is equivalent to formDecodeParameterNameOrValue.

Parameters

string

String to be URL decoded, which may be `null`.

Returns

string

URL decoding of the provided string, or `null` if string was `null`.

urlEncode

`urlEncode(string)`

Returns the URL encoding of the provided string.

This is equivalent to formEncodeParameterNameOrValue.

Parameters

string

String to be URL encoded, which may be `null`.

Returns

string

URL encoding of the provided string, or `null` if string was `null`.

urlDecodeFragment

`urlDecodeFragment(string)`

Returns the string that results from decoding the provided URL encoded fragment as per RFC 3986, which can be `null` if the input is `null`.

Parameters

string

URL encoded fragment.

Returns

string

String resulting from decoding the provided URL encoded fragment as per RFC 3986.

urlDecodePathElement

`urlDecodePathElement(string)`

Returns the string that results from decoding the provided URL encoded path element as per RFC 3986, which can be `null` if the input is `null`.

Parameters

string

The path element.

Returns

string

String resulting from decoding the provided URL encoded path element as per RFC 3986.

urlDecodeQueryParameterNameOrValue

`urlDecodeQueryParameterNameOrValue(string)`

Returns the string that results from decoding the provided URL encoded query parameter name or value as per RFC 3986, which can be `null` if the input is `null`.

Parameters

string

Parameter name or value.

Returns

string

String resulting from decoding the provided URL encoded query parameter name or value as per RFC 3986.

urlDecodeUserInfo

`urlDecodeUserInfo(string)`

Returns the string that results from decoding the provided URL encoded userInfo as per RFC 3986, which can be `null` if the input is `null`.

Parameters

string

URL encoded userInfo.

Returns

string

String resulting from decoding the provided URL encoded userInfo as per RFC 3986.

urlEncodeFragment

`urlEncodeFragment(string)`

Returns the string that results from URL encoding the provided fragment as per RFC 3986, which can be `null` if the input is `null`.

Parameters

string

Fragment.

Returns

string

The string resulting from URL encoding the provided fragment as per RFC 3986.

urlEncodePathElement

`urlEncodePathElement(string)`

Returns the string that results from URL encoding the provided path element as per RFC 3986, which can be `null` if the input is `null`.

Parameters

string

Path element.

Returns

string

String resulting from URL encoding the provided path element as per RFC 3986.

urlEncodeQueryParameterNameOrValue

`urlEncodeQueryParameterNameOrValue(string)`

Returns the string that results from URL encoding the provided query parameter name or value as per RFC 3986, which can be `null` if the input is `null`.

Parameters

string

Parameter name or value.

Returns

string

String resulting from URL encoding the provided query parameter name or value as per RFC 3986.

urlEncodeUserInfo

`urlEncodeUserInfo(string)`

Returns the string that results from URL encoding the provided userInfo as per RFC 3986, which can be `null` if the input is `null`.

Parameters

string

userInfo.

Returns

string

String resulting from URL encoding the provided userInfo as per RFC 3986.

More information

Some functions are provided by org.forgerock.openig.el.Functions.

Other functions are provided by org.forgerock.http.util.Uris.

Patterns

Patterns in configuration parameters and expressions use the standard Java regular expression Pattern class. For more information on regular expressions, see Oracle's tutorial on Regular Expressions.

Pattern templates

A regular expression pattern template expresses a transformation to be applied for a matching regular expression pattern. It may contain references to capturing groups within the match result. Each occurrence of `$g` (where g is an integer value) is substituted by the indexed capturing group in a match result. Capturing group zero `"$0"` denotes the entire pattern match. A dollar sign or numeral literal immediately following a capture group reference can be included as a literal in the template by preceding it with a backslash (`\`). Backslash itself must be also escaped in this manner.

More information

Java Pattern class

Regular Expressions tutorial

Scripts

IG uses Groovy 4 for scripting. For more information, refer to the Groovy Language Documentation.

Groovy scripts used in the IG configuration are restricted to the UTF-8 character set.

Use Groovy scripts with the following object types:

- ScriptableFilter, to customize flow of requests and responses
- ScriptableHandler, to customize creation of responses
- ScriptableThrottlingPolicy, to customize throttling rates
- ScriptableAccessTokenResolver to customize resolution and validation of OAuth 2.0 access tokens
- `ScriptableResourceAccess` in OAuth2ResourceServerFilter, to customize the list of OAuth 2.0 scopes required in an OAuth 2.0 access token
- ScriptableIdentityAssertionPlugin, to use with an IdentityAssertionHandler for local processing.
- ScriptableIdentityAssertionPluginTechPreview, to use with an IdentityAssertionHandlerTechPreview for local authentication or authorization.

When IG accesses a script, it compiles and then caches the script. IG uses the cached version until the script is changed.

After you update a script used in a route, leave at least one second before processing a request. The Groovy interpreter needs time to detect and take the update into account.

Important

When writing scripts or Java extensions that use the Promise API, avoid the blocking methods `get()`, `getOrThrow()`, and `getOrThrowUninterruptibly()`. A promise represents the result of an asynchronous operation; therefore, using a blocking method to wait for the result can cause deadlocks and/or race issues.
Instead, consider using `then()` methods, such as `thenOnResult()`, `thenAsync()`, or `thenCatch()`, which allow execution blocks to be executed when the response is available.

Blocking code example

```
def response = next.handle(ctx, req).get() // Blocking method 'get' used
response.headers['new']="new header value"
return response
```

Non-blocking code example

```
return next.handle(ctx, req)
            //Process result when it is available
           .thenOnResult { response ->
             response.headers['new']="new header value"
           }
```

Usage

```
{
  "name": string,
  "type": scriptable object type,
  "config": {
    "type": string,
    "file": configuration expression<string>,  // Use either "file"
    "source": [ string, ... ],                 // or "source", but not both.
    "args": map or configuration expression<map>,
    "clientHandler": Handler reference
  }
}
```

Properties

`"type"`: *string*, *required*

The Internet media type (formerly MIME type) of the script, "application/x-groovy" for Groovy

`"file"`: *configuration expression*<*string*>, *required if* `source` *is not used*

Path to the file containing the script; mutually exclusive with `source`. Specify `file` as follows:

For Groovy files from default packages

- Place Groovy files in the base script directory, `$HOME/.openig/scripts/groovy` (on Windows, `%appdata%\OpenIG\scripts\groovy`). For example, place `myScript.groovy` from the default package in `$HOME/.openig/scripts/groovy`.
- Specify `file` with the filename of the Groovy file. For the previous example, specify:

```
"config": {
  "type": "application/x-groovy",
  "file": "myScript.groovy"
}
```

For Groovy files from non-default packages

- Place Groovy files in a subdirectory of the base script directory that corresponds to the package name. For example, place `myScript.groovy` from the package `com.example.groovy` in `$HOME/.openig/scripts/groovy/com/example/groovy`.
- Specify `file` with the relative path from the base script directory **and** the filename. For the previous example, specify:

```
"config": {
  "type": "application/x-groovy",
  "file": "com/example/groovy/myScript.groovy"
}
```

Important

IG runs scripts from an absolute path, or from a path relative to the base script directory. Routes that refer to scripts otherwise, such as through a URL, fail to deploy.
Do one of the following to prevent errors:

- Move scripts to the base script directory or the correct subdirectory of the base script directory
- Refer to scripts through an absolute path

`"source"`*: array of <strings>, required if* `file` *is not used*

The script as one or more strings; mutually exclusive with `file`.

The following example shows the source of a script as an array of strings:

```
"source": [
    "Response response = new Response(Status.OK)",
    "response.entity = 'foo'",
    "return response"
]
```

`"args"`: *map or configuration expression<map>, optional*

A map of one or more data pairs with the format `Map<String, String>`, where:

- The key is the name of a configuration parameter in a script
- The value is a string to use in the script, or a configuration expression that evaluates to the string

The following formats are allowed:

```
{
  "args": {
    "string": "configuration expression<string>",
    ...
  }
}
```

```
{
  "args": "configuration expression<map>"
}
```

In the following example, the property is a map whose values are scalars, arrays, and objects:

```
{
  "args": {
    "title": "Coffee time",
    "status": 418,
    "reason": [
      "Not Acceptable",
      "I'm a teapot",
      "Acceptable"
    ],
    "names": {
      "1": "koffie",
      "2": "kafe",
      "3": "cafe",
      "4": "kafo"
    }
  }
}
```

- A script can access the `args` parameters in the same way as other global objects. The following example sets the response status to `I'm a teapot`:

  ```
  response.status = Status.valueOf(418, reason[1])
  ```

  For information about the 418 status coderefer to RFC 7168: 418 I'm a Teapot.

- The following example configures arguments as strings and numbers for a ScriptableThrottlingPolicy:

```
"args": {
  "status": "gold",
  "rate": 6,
  "duration": "10 seconds"
}
```

The following lines set the throttling rate to 6 requests each 10 seconds when the response status is `gold`:

```
if (attributes.rate.status == status) {
  return new ThrottlingRate(rate, duration)
}
```

- The following example configures arguments that reference a SampleFilter defined in the heap:

```
{
  "heap": [
    {
      "name": "SampleFilter",
      "type": "SampleFilter",
      "config": {
        "name": "X-Greeting",
        "value": "Hello world"
      }
    }
  ]
}
```

In the following example, the property is a map whose value is an expression to pass SampleFilter to the script:

```
{
  "args": {
    "filter": "${heap['SampleFilter']}"
  }
}
```

The script can then reference SampleFilter as `filter`.

`"clientHandler"`: *ClientHandler reference, optional*

A Handler for making outbound HTTP requests to third-party services. In a script, `clientHandler` is wrapped within the global object `http`.

Default: The default ClientHandler.

Available objects

The following global objects are available to scripts:

Any parameters passed as args

You can use the configuration to pass parameters to the script by specifying an args object.

The args object is a map whose values can be scalars, arrays, and objects. The args object can reference objects defined in the heap by using expressions, for example, `"${heap['ObjectName']}"`.

The values for script arguments can be defined as configuration expressions, and evaluated at configuration time.

Script arguments cannot refer to `context` and `request`, but `context` and `request` variables can be accessed directly within scripts.

Take care when naming keys in the args object. If you reuse the name of another global object, cause the script to fail and IG to return a response with HTTP status code 500 Internal Server Error.

All heap objects

The heap object configuration, described in Heap objects.

`openig`

An implicit object that provides access to the environment when expressions are evaluated.

`attributes`

The attributes object provides access to a context map of arbitrary attributes, which is a mechanism for transferring transient state between components when processing a single request.

Use `session` for maintaining state between successive requests from the same logical client.

`builder`

For ScriptableJwtValidatorCustomizer only.

Used by the ScriptableJwtValidatorCustomizer and JwtValidationFilter to create constraints to test JWT claims and sub-claims. The purpose of the ScriptableJwtValidatorCustomizer is to enrich the `builder` object.

For information about methods to enrich the `builder` instance, refer to JwtValidator.Builder.

`constraints`

The constraints object, all its static methods, `constant(String)`, and `claim(String)`.

Use this object for JWT validation with the `customizer` property of JwtValidationFilter.

`claim(String)` must be followed by one of the following methods: `asString()`, `asInteger()`, `asLong()`, `asDouble()`, `asBoolean()`, `as(yourCustomJsonValueTransformer)`

`context`

The processing context.

This context is the leaf of a chain of contexts. It provides access to other Context types, such as SessionContext, AttributesContext, and ClientContext, through the `context.asContext(ContextClass.class)` method.

`contexts`

a map<string, context> object. For information, refer to Contexts.

`request`

The HTTP request.

> **Important**
>
> The `request.form` method, used in scripts to read or set query and form parameters, is deprecated. Use the following replacement settings:
>
> - `Request.getQueryParams()` to read query parameters
> - `Entity.getForm()` to read form parameters
> - `Entity.setForm()` to set form parameters
>
> For more information, refer to the Deprecated section of the *Release Notes*.

`globals`

This object is a Map that holds variables that persist across successive invocations.

`http`

An embedded client for making outbound HTTP requests, which is an org.forgerock.http.Client.

If a `"clientHandler"` is set in the configuration, then that Handler is used. Otherwise, the default ClientHandler configuration is used.

For information, refer to Handlers.

`logger`

The logger object provides access to a unique SLF4J logger instance for scripts, where the logger instance is named with the script name.

For information about logging for scripts, refer to Logging in scripts.

`next`

The object named `next` refers to the next element in the chain, which can be the following filter or the terminal handler. If the next object in the chain is a filter, IG wraps it in a handler.

`session`

The session object provides access to the session context, which is a mechanism for maintaining state when processing a successive requests from the same logical client or end user.

Use `attributes` for transferring transient state between components when processing a single request.

Imported classes

The following classes are imported automatically for Groovy scripts:

- org.forgerock.http.Client

- org.forgerock.http.Filter
- org.forgerock.http.Handler
- org.forgerock.http.protocol.Header
- org.forgerock.openig.filter.throttling.ThrottlingRate
- org.forgerock.http.util.Uris
- org.forgerock.util.AsyncFunction
- org.forgerock.util.Function
- org.forgerock.util.promise.NeverThrowsException
- org.forgerock.util.promise.Promise
- org.forgerock.util.promise.Promises
- org.forgerock.services.context.Context
- org.forgerock.http.protocol.*
- org.forgerock.http.oauth2.AccessTokenInfo
- org.forgerock.json.JsonValue, and all its static methods, including `json(Object)`, `array(Object…)`, `object(fields…)`, and `field(String, Object)`
- org.forgerock.openig.util.JsonValues and all its static methods.
- org.forgerock.openig.tools.jwt.validation.Constraints and all its static methods.

More information

- ScriptableFilter, org.forgerock.openig.filter.ScriptableFilter, and org.forgerock.http.Filter
- ScriptableHandler, org.forgerock.openig.handler.ScriptableHandler, and org.forgerock.http.Handler
- ScriptableThrottlingPolicy, org.forgerock.openig.filter.throttling.ScriptableThrottlingPolicy.Heaplet, and org.forgerock.openig.filter.throttling.ThrottlingPolicy
- `ScriptableResourceAccess` in OAuth2ResourceServerFilter, org.forgerock.openig.filter.oauth2.ScriptableResourceAccess, and org.forgerock.http.oauth2.ResourceAccess
- `ScriptableAccessTokenResolver` in OAuth2ResourceServerFilter, org.forgerock.openig.filter.oauth2.ScriptableAccessTokenResolver, and org.forgerock.http.oauth2.AccessTokenResolver
- `ScriptableJwtValidatorCustomizer` in JwtValidationFilter and org.forgerock.openig.filter.jwt.ScriptableJwtValidatorCustomizer

Route properties

Configuration parameters, such as host names, port numbers, and directories, can be declared as property variables in the IG configuration or in an external JSON file. The variables can then be used in expressions in routes and in `config.json` to set the value of configuration parameters.

Properties can be inherited across the router, so a property defined in `config.json` can be used in any of the routes in the configuration. Storing the configuration centrally and using variables for parameters that can be different for each installation makes it easier to deploy IG in different environments without changing a single line in your route configuration.

Usage

Simple property configured inline

```
{
  "properties": {
    "<variable name>": "valid JSON value"
  }
}
```

Group property configured inline

```
{
  "properties": {
    "<group name>": {
      "<variable name>": "valid JSON value", ...
    }
  }
}
```

Properties configured in one or more external files

```
{
  "properties": {
    "$location": expression
  }
}
```

In this example, `description1` and `description2` prefix the variable names contained in the external file.

```
{
  "properties": {
    "description1": {
      "$location": expression
    },
    "description2": {
      "$location": expression
    }
  }
}
```

Properties

`"<variable name>"`: *configuration expression*<*string*>

The name of a variable to use in the IG configuration. The variable can be used in expressions in routes or in `config.json` to assign the value of a configuration parameter.

The value assigned to the variable can be any valid JSON value: string, number, boolean, array, object, or null.

- In the following example from `config.json`, the URL of an application is declared as a property variable named `appLocation`. The variable is then used by the `baseURI` parameter of the handler, and can be used again in other routes in the configuration.

```
{
  "properties": {
    "appLocation": "http://app.example.com:8081"
  },
  "handler": {
    "type": "Router",
    "baseURI": "${appLocation}",
    "capture": "all"
  }
}
```

- In the following example, the property variable `ports` is added to define an array of port numbers used by the configuration. The `ports` variable is referenced in the `appLocation` variable, and is resolved at runtime with the value in the ports array:

```
{
  "properties": {
    "ports": [8080, 8081, 8088],
    "appLocation": "http://app.example.com:${ports[1]}"
  },
  "handler": {
    "type": "Router",
    "baseURI": "${appLocation}",
    "capture": "all"
  }
}
```

- In the following example route, the request path is declared as the property variable `uriPath`, with the value `hello`, and the variable is used by the route condition:

```
{
  "properties": {
    "uriPath": "hello"
  },
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/plain; charset=UTF-8" ]
      },
      "entity": "Hello world!"
    }
  },
  "condition": "${matchesWithRegex(request.uri.path, '^/welcome') or matchesWithRegex(request.uri.path,
'&{uriPath}')}"
}
```

When IG is set up as described in the Quick install, requests to `ig.example.com:8080/hello` or `ig.example.com:8080/welcome` can access the route.

`"<group name>"`: <*object*>, *required*

The name of a group of variables to use in the IG configuration. The group name and variable name are combined using dot notation in an expression.

In the following example from `config.json`, the property group `directories` contains two variables that define the location of files:

```
{
  "properties": {
    "directories": {
      "config": "${openig.configDirectory.path}",
      "auditlog": "/tmp/logs"
    }
  }
}
```

The group name and variable name are combined using dot notation in the following example to define the directory where the audit log is stored:

```
{
  "type": "AuditService",
  "config": {
    "eventHandlers": [
      {
        "class": "org.forgerock.audit.handlers.csv.CsvAuditEventHandler",
        "config": {
          "name": "csv",
          "logDirectory": "${directories.auditlog}",
          . . .
```

`"$location"`*: configuration expression<string>, required*

The location and name of one or more JSON files where property variables are configured.

Files must be `.json` files, and contain property variables with a key/value format, where the key cannot contain the period (`.`) separator.

For example, this file is correct:

```
{
  "openamLocation": "http://am.example.com:8088/openam/",
  "portNumber": 8081
}
```

This file would cause an error:

```
{
  "openam.location": "http://am.example.com:8088/openam/",
  "port.number": 8081
}
```

Examples

Property variables configured in one file

In the following example, the location of the file that contains the property variables is defined as an expression:

```
{
  "properties": {
    "$location": "${fileToUrl(openig.configDirectory)}/myProperties.json"
  }
}
```

In the following example, the location of the file that contains the property variables is defined as a string:

```
{
  "properties": {
    "$location": "file:///Users/user-id/.openig/config/myProperties.json"
  }
}
```

The file location can be defined as any real URL.

The file `myProperties.json` contains the base URL of an AM service and the port number of an application.

```
{
  "openamLocation": "http://am.example.com:8088/openam/",
  "appPortNumber": 8081
}
```

Property variables configured in multiple files

In the following example, the property variables are contained in two files, defined as a set of strings:

```
{
  "properties": {
    "urls": {
      "$location": "file://path-to-file/myUrlProperties.json"
    },
    "ports": {
       "$location": "file://path-to-file/myPortProperties.json"
    }
  }
}
```

The file `myUrlProperties.json` contains the base URL of the sample application:

```
{
  "appUrl": "http://app.example.com"
}
```

The file `myPortProperties.json` contains the port number of an application:

```
{
  "appPort": 8081
}
```

The base config file, `config.json`, can use the properties as follows:

```
{
  "properties": {
    "urls": {
      "$location": "file:///Users/user-id/.openig/config/myUrlProperties.json"
    },
    "ports": {
      "$location": "file:///Users/user-id/.openig/config/myPortProperties.json"
    }
  },
  "handler": {
    "type": "Router",
    "name": "_router",
    "baseURI": "${urls.appUrl}:${ports.appPort}",
    . . .
```

Contexts

The root object for request context information.

Contexts is a map of available contexts, which implement the Context interface. The `contexts` map is populated dynamically when creating bindings to evaluate expressions and scripts.

If a context type appears multiple times in the chain of contexts, only the last value of the context is exposed in the contexts map. For example, if a route contains two JwtBuilderFilters that each provided data in the JwtBuilderContext, only data from the last processed JwtBuilderFilter is contained in `${contexts.jwtBuilder}`. Data from the first processed JwtBuilderFilter can be accessed by scripts and extensions through the `Context` API. The following example script accesses data from the first processed JwtBuilderFilter:

```
Context second = context.get("jwtBuilder")
     .map(Context::getParent)
     .flatMap(ctx -> ctx.get("jwtBuilder"))
     .orElse(null);
```

The map keys are strings and the values are context objects. All context objects use their version of the following properties:

`"context-Name"`*: string*

Context name.

`"context-ID"`*: string*

Read-only string uniquely identifying the context object.

`"context-rootContext"`*: boolean*

True if the context object is a RootContext (has no parent).

`"context-Parent"`*: Context object*

Parent of this context object.

Contexts provide contextual information to downstream filters and handlers about a handled request. The context can include information about the client making the request, the session, the authentication or authorization identity of the user, and any other state information associated with the request.

Contexts provide a way to access state information throughout the duration of the HTTP session between the client and protected application. Interaction with additional services can also be captured in the context.

Filters can enrich existing contexts (store objects in sessions or attributes), or by provide new contexts tailored for a purpose. Therefore, the list of available context is dynamic and depends on which filters have been executed when a context is queried. For example, a context can be queried by a script, at startup, or at runtime.

Unlike session information, which spans multiple request/response exchanges, contexts last only for the duration of the request/response exchange, and are then lost.

Summary of contexts

Type	Accessible at	Populated by	Contains
AttributesContext	`contexts.attributes.attributes` and `attributes`	IG core, when a request enters IG	Map of request attributes for use by filters at different positions in the chain
AuthRedirectContext	`${contexts.AuthRedirectContext}`	FragmentFilter and DataPreservationFilter	Indication to the FragmentFilter and DataPreservationFilter that a login redirect is pending
CapturedUserPasswordContext	`${contexts.capturedPassword}`	CapturedUserPasswordFilter	Decrypted AM password of the current user
CdSsoContext	`${contexts.cdsso}`	CrossDomainSingleSignOnFilter	CDSSO token properties, session user ID, full claims set.
CdSsoFailureContext	`${contexts.cdssoFailure}`	CrossDomainSingleSignOnFilter	Information about errors occurring during CDSSO authentication
ClientContext	`${contexts.client}`	IG core, when a request enters IG	Information about the client sending the request, and the client certificate if using mTLS
IdentityRequestJwtContext	`${contexts.IdentityRequestJwtContext}`	IdentityAssertionHandler	Information and claims for an identity request JWT issued by Identity Cloud to IG.

Type	Accessible at	Populated by	Contains
IdpSelectionLoginContext	`${contexts.idpSelectionLogin}`	AuthorizationCodeOAuth2ClientFilter when `loginHandler` is specified.	The original target URI for a request received by IG.
JwtBuilderContext	`${contexts.jwtBuilder}`	JwtBuilderFilter	Built JWT as string, JsonValue, or map
JwtValidationContext	`${contexts.jwtValidation}`	JwtValidationFilter and IdTokenValidationFilter	Properties of a JWT after validatedation
JwtValidationErrorContext	`${contexts.jwtValidationError}`	JwtValidationFilter and IdTokenValidationFilter	Properties of a JWT after validation fails
OAuth2Context	`${contexts.oauth2}`	OAuth2ResourceServerFilter	Properties of an OAuth 2.0 access token after validation
OAuth2TokenExchangeContext	`${contexts.oauth2TokenExchange}`	OAuth2TokenExchangeFilter	Issued token and its scopes
OAuth2FailureContext	`${contexts.oauth2Failure}`	AuthorizationCodeOAuth2ClientFilter and OAuth2TokenExchangeFilter	OAuth 2.0 authorization operation error and error description
PolicyDecisionContext	`${contexts.policyDecision}`	PolicyEnforcementFilter	Attributes and advices returned by AM policy decisions
SessionContext	`${contexts.session}`	IG core, when a request enters IG	Information about stateful and stateless sessions
SessionInfoContext	`${contexts.amSession}`	SessionInfoFilter	AM session information and properties
SsoTokenContext	`${contexts.ssoToken}`	SingleSignOnFilter and CrossDomainSingleSignOnFilter	SSO tokens and their validation information
StsContext	`${contexts.sts}`	TokenTransformationFilter	Result of a token transformation

Type	Accessible at	Populated by	Contains
TransactionIdContext	`${contexts.transactionId}`	IG core, when a request enters IG	ForgeRock transaction ID of a request
UriRouterContext	`${contexts.router}`	IG core, when a request traverses a route	Routing information associated with a request
UserProfileContext	`${contexts.userProfile}`	UserProfileFilter	User profile information

AttributesContext

Provides a map for request attributes. When IG processes a single request, it injects transient state information about the request into this context. Attributes stored when processing one request are not accessible when processing a subsequent request.

IG automatically provides access to the `attributes` field through the `attributes` bindings in expressions. For example, to access a username with an expression, use `${attributes.credentials.username}` instead of `${contexts.attributes.attributes.credentials.username}`

Use SessionContext to maintain state between successive requests from the same logical client.

Properties

The context is named `attributes`, and is accessible at `${attributes}`. The context has the following property:

`"attributes"`: *map*

Map with the format `Map<String, Object>`, where:

- Key: Attribute name
- Value: Attribute value

Cannot be null.

More information

org.forgerock.services.context.AttributesContext

AuthRedirectContext

Used by the following filters to indicate that a login redirect is pending:

- FragmentFilter
- DataPreservationFilter

This context is not intended for use in scripts or extensions.

For a single request there must be at most one instance of AuthRedirectContext in the context hierarchy. Confirm for the presence of an AuthRedirectContext before adding a new instance or adding query parameters to an existing instance.

The context is named `AuthRedirectAwareContext`, and is accessible at `${contexts.AuthRedirectContext}`.

Properties

`"impendingIgRedirectNotified"`*: boolean*

Returns `true` if an IG redirect attempt is pending. Otherwise, returns `false`.

`"notifyImpendingIgRedirectAndUpdateUri"`*: URI*

Notifies that an IG redirection has been attempted, and returns an updated URI as follows:

- If no query parameters are added to the context, return the original URI.
- If query parameters are added to the context, apply them to the URI and return an updated URI.
- If the added query parameters have the same name as existing query parameters, replace the existing parameters and return an updated URI.

For example, a request to `example.com/profile` triggers a login redirect to `example.com/login`. After authentication, the request is expected to be redirected to the original URI, `example.com/profile`.

`"addQueryParameter"`*: java.lang.String*

Adds a query parameter to the context, for use by `notifyImpendingIgRedirectAndUpdateUri`.

More information

org.forgerock.openig.filter.AuthRedirectContext

CapturedUserPasswordContext

Provides the decrypted AM password of the current user. When the CapturedUserPasswordFilter processes a request, it injects the decrypted password from AM into this context.

Properties

The context is named `capturedPassword`, and is accessible at `${contexts.capturedPassword}`. The context has the following properties:

`"raw"`*: byte*

The decrypted password as bytes.

`"value"`*: java.lang.String*

The decrypted password as a UTF-8 string.

More information

org.forgerock.openig.openam.CapturedUserPasswordContext

CdSsoContext

Provides the cross-domain SSO properties for the CDSSO token, the user ID of the session, and the full claims set. When the CrossDomainSingleSignOnFilter processes a request, it injects the information in this context.

Properties

The context is named `cdsso`, and is accessible at `${contexts.cdsso}`. The context has the following properties:

`"claimsSet"`: *org.forgerock.json.jose.jwt.JwtClaimsSet*

Full JwtClaimsSet for the identity of the authenticated user. Cannot be null.

Access claims as follows:

- Claims with a getter by using the property name. For example, access `getSubject` with `contexts.cdsso.claimsSet.subject`.
- All other claims by using the `getClaim` method. For example, access `subname` with `contexts.cdsso.claimsSet.getClaim('subname')`.

`"cookieInfo"`: *org.forgerock.openig.http.CookieBuilder*

Configuration data for the CDSSO authentication cookie, with the following attributes:

- `name`: Cookie name (string)
- `domain`: (Optional) Cookie domain (string)
- `path`: Cookie path (string)

No attribute can be null.

`"redirectEndpoint"`: *java.lang.String*

Redirect endpoint URI configured for communication with AM. Cannot be null.

`"sessionUid"`: *java.lang.String*

Universal session ID. Cannot be null.

`"token"`: *java.lang.String*

Value of the CDSSO token. Cannot be null.

More information

org.forgerock.openig.openam.CdSsoContext

CdSsoFailureContext

Contains the error details for any error that occurred during cross-domain SSO authentication. When the CrossDomainSingleSignOnFilter processes a request, should an error occur that prevents authentication, the error details are captured in this context.

Properties

The context is named `cdssoFailure`, and is accessible at `${contexts.cdssoFailure}`. The context has the following properties:

`"error"`: *java.lang.String*

The error that occurred during authentication. Cannot be null.

`"description"`: *java.lang.String*

A description of the error that occurred during authentication. Cannot be null.

`"throwable"`: *java.lang.Throwable*

Any `Throwable` associated with the error that occured during authentication. Can be null.

More information

org.forgerock.openig.openam.CdSsoFailureContext

ClientContext

Information about the client sending a request. When IG receives a request, it injects information about the client sending the request into this context.

Properties

The context is named `client`, and is accessible at `${contexts.client}`. The context has the following properties:

`"certificates"`: *java.util.List* <*java.security.cert.Certificate*>

List of X.509 certificates presented by the client. If the client does not present any certificates, IG returns an empty list. Never `null`.

The following example uses the certificate associated with the incoming HTTP connection:

```
{
  "name": "CertificateThumbprintFilter-1",
  "type": "CertificateThumbprintFilter",
  "config": {
    "certificate": "${contexts.client.certificates[0]}"
  }
}
```

`"isExternal"`: *boolean*

True if the client connection is external.

`"isSecure"`: *boolean*

True if the client connection is secure.

`"localAddress"`: *java.lang.String*

The IP address of the interface that received the request.

`"localPort"`: *integer*

The port of the interface that received the request.

`"remoteAddress"`: *java.lang.String*

The IP address of the client (or the last proxy) that sent the request.

`"remotePort"`: *integer*

The source port of the client (or the last proxy) that sent the request.

`"remoteUser"`: *java.lang.String*

The login of the user making the request, or `null` if unknown. This is likely to be `null` unless you have deployed IG with a non-default deployment descriptor that secures the IG web application.

`"userAgent"`: *java.lang.String*

The value of the User-Agent HTTP header in the request if any, otherwise `null`.

More information

org.forgerock.services.context.ClientContext

IdentityRequestJwtContext

Provides the properties of an identity request JWT issued by Identity Cloud to IG as part of an Identity Cloud authentication journey with an IdentityGatewayAssertionNode node.

The context is created by the IdentityAssertionHandler.

Properties

The context is named `identityRequestJwt`, and is accessible at `${contexts.identityRequestJwt}`. The context has the following properties:

`"dataClaims"`: *java.util.Map*

Map of claims that can be required by a plugin, in the format `Map<String, Object>`, where:

- Key: Claim name
- Value: Claim value

Claims are documented on a per-plugin basis.

If no claim is provided, this is an empty map.

`"nonce"`: *java.lang.String*

Unique ID generated by the IdentityGatewayAssertionNode and returned in the identity assertion JWT.

Can't be null

`"redirect"`: *java.net.URI*

The URL on which to send the identity assertion JWT.

Can't be null

`"version"`: *java.lang.String*

The JWT version; only the value `v1` is supported.

Can't be null

IdpSelectionLoginContext

Provides the original target URI for the request received by IG. Use this context with `loginHandler` in AuthorizationCodeOAuth2ClientFilter.

Properties

The context is named `idpSelectionLogin` and is accessible at `${contexts.idpSelectionLogin}`. The context has the following property:

`"originalUri"`: *URI*

The original target URI for the request received by IG. The value of this field is read-only.

JwtBuilderContext

When the JwtBuilderFilter processes a request, it stores provided data in this context. This context returns the JWT as string, JsonValue, or map for downstream use.

Properties

The context is named `jwtBuilder`, and is accessible at `${contexts.jwtBuilder}`, with the following properties:

`"value"`: *java.lang.String*

The base64url encoded UTF-8 parts of the JWT, containing name-value pairs of data. Cannot be null.

`"claims"`: *java.util.Map*

Map with the format `Map<String,Object>`, where:

- Key: Claim name
- Value: Claim value

`"claimsAsJsonValue"`: *org.forgerock.json.JsonValue*

Claims as a JSon value.

More information

org.forgerock.openig.filter.JwtBuilderFilter

org.forgerock.openig.filter.JwtBuilderContext

JwtValidationContext

Provides the properties of a JWT after validation. When the JwtValidationFilter validates a JWT, or the IdTokenValidationFilter validates an id_token, it injects a copy of the JWT and its claims into this context.

Properties

The context is named `jwtValidation`, and is accessible at `${contexts.jwtValidation}`. The context has the following properties:

`"value"`: *java.lang.String*

The value of the JWT. Cannot be null.

`"claims"`: *org.forgerock.json.jose.jwt.JwtClaimsSet*

A copy of the claims as a JwtClaimsSet.

`"info"`: *java.util.Map*

A map in the format `Map<String, Object>`, where:

- Key: Claim name
- Value: Claim value

`"jwt"`: *org.forgerock.json.jose.jwt.Jwt*

A copy of the JWT.

More information

org.forgerock.openig.filter.jwt.JwtValidationFilter

org.forgerock.openig.filter.oauth2.client.IdTokenValidationFilterHeaplet

org.forgerock.openig.filter.jwt.JwtValidationContext

org.forgerock.openig.filter.jwt.JwtValidationErrorContext

JwtValidationErrorContext

Provides the properties of a JWT after validation fails. When the JwtValidationFilter fails to validate a JWT, or the IdTokenValidationFilter fails to validate an id_token, it injects the JWT and a list of violations into this context.

Properties

The context is named `jwtValidationError`, and is accessible at `${contexts.jwtValidationError}`. The context has the following properties:

`"jwt"`: *java.lang.String*

The value of the JWT. Cannot be null.

`"violations"`: *java.util.List*<*Violation*>

A list of violations.

More information

org.forgerock.openig.filter.jwt.JwtValidationFilter

org.forgerock.openig.filter.oauth2.client.IdTokenValidationFilterHeaplet

org.forgerock.openig.filter.jwt.JwtValidationContext

org.forgerock.openig.filter.jwt.JwtValidationErrorContext

OAuth2Context

Provides OAuth 2.0 access tokens. When the OAuth2ResourceServerFilter processes a request, it injects the access token into this context.

Properties

The context name is `oauth2`, and is accessible at `${contexts.oauth2}`. The context has the following properties:

`"accessToken"`: *org.forgerock.http.oauth2.AccessTokenInfo*

The AccessTokenInfo is built from the following properties:

`"info"`: *java.util.Map*

A map with the format `Map<String,Object>`, where

- Key: Claim name
- Value: Claim value in raw JSON

`"token"`: *java.lang.String*

Access token identifier issued from the Authorization Server.

`"scopes"`: *java.util.Set*

A set scopes associated to this token, with the format `Set<String>`.

`"expiresAt"`: *java.lang.Long*

Timestamp of when the token expires, in milliseconds since epoch.

More information

org.forgerock.http.oauth2.OAuth2Context

org.forgerock.http.oauth2.AccessTokenInfo

OAuth2TokenExchangeContext

When the OAuth2TokenExchangeFilter succesfully issues a token, it injects the issued token and its scopes into this context.

Properties

The context name is `OAuth2TokenExchangeContext`, and is accessible at `${contexts.oauth2TokenExchange}`.

The context has the following properties:

`"issuedToken"`: *java.lang.String*

The token issued by the Authorization Server.

`"issuedTokenType"`: *java.lang.String*

The token type URN.

`"scopes"`: *java.util.Set*<*java.lang.String*>

One or more scopes associated with the issued token, for example, `"scope1", "scope2", "scope3"`.

`"rawInfo"`: *org.forgerock.json.JsonValue*

The raw token info as issued by the Authorization Server.

More information

org.forgerock.openig.filter.oauth2.OAuth2TokenExchangeContext

OAuth2FailureContext

OAuth2TokenExchangeFilter

RFC 6749: Error Response

OAuth2FailureContext

When an OAuth 2.0 authorization operation fails, the error and error description provided by the authorization service are injected into this context for use downstream.

The amount and type of information in the context depends on when a failure occurs.

This context is created by AuthorizationCodeOAuth2ClientFilter and OAuth2TokenExchangeFilter.

This context supports OAuth 2.0 error messages in the format given by RFC 6749.

Properties

The context is named `OAuth2Failure`, and is accessible at `${contexts.oauth2Failure}`. The context has the following properties:

`"error"`: *java.lang.String*

The error field name.

`"description"`: *java.lang.String*

Error description field name.

`"exception"`: *org.forgerock.openig.filter.oauth2.client.OAuth2ErrorException*

The OAuth 2.0 exception associated with the token exchange error.

Examples

For examples that use `${contexts.oauth2Failure.error}` and `${contexts.oauth2Failure.description}`, refer to the routes in OAuth 2.0 token exchange and Discover and dynamically register with OpenID Connect providers.

More information

org.forgerock.openig.filter.oauth2.OAuth2FailureContext

OAuth2TokenExchangeContext

OAuth2TokenExchangeFilter

RFC 6749: Error Response

PolicyDecisionContext

Provides attributes and advices returned by AM policy decisions. When the PolicyEnforcementFilter processes a request, it injects the attributes and advices into this context.

Properties

The context is named `policyDecision`, and is accessible at `${contexts.policyDecision}`. The context has the following properties:

`"attributes"`: *java.util.Map*

A map with the format `Map<String, List<String>>`, where:

- Key: Attribute name.
- Value: A One or more attribute values provided in the policy decision. Can be empty, but not null.

`"jsonAttributes"`: *java.util.Map*

A map with the format `Map<String, List<String>>`, where:

- Key: Attribute name.

- Value: One or more attribute values provided in the policy decision. Can be empty, but not null.

"advices": *java.util.Map*

A map with the format `Map<String, List<String>>`, where:

- Key: Advice name.
- Value: One or more advice values provided in the policy decision. Can be empty, but not null.

"jsonAdvices": *java.util.Map*

A map with the format `Map<String, List<String>>`, where:

- Key: Advice name
- Value: One or more advice values provided in the policy decision. Can be empty, but not null.

"actions": *java.util.Map*

A map with the format `Map<String, Boolean>` where:

- Key: Action name.
- Value: `true` when an action is allowed for the specified resource, `false` otherwise. Cannot be null.

"jsonActions": *json.JsonValue*

A map with the format `Map<String, Boolean>`, where:

- Key: Action name.
- Value: `true` when an action is allowed for the specified resource, `false` otherwise. Cannot be null.

"resource": *java.lang.String*

The resource value used in the policy request. Can be empty, but not null.

More information

org.forgerock.openig.openam.PolicyDecisionContext.html

SessionContext

Provides access to information about stateful and stateless sessions.

To process a single request, consider using AttributesContext to transfer transient state between components and prevent IG from creating additional sessions.

IG automatically provides access to the `session` field through the `session` bindings in expressions. For example, to access a username with an expression, use `${session.username}` instead of `${contexts.session.session.username}`

Properties

The context is named `session`, and is accessible at `${contexts.session}`. The context has the following properties:

"session": *java.util.Map*

A map with the format `Map<String, Object>`, where:

- Key: Session property name
- Value: Session property value

Any object type can be stored in the session.

More information

org.forgerock.http.session.SessionContext

SessionInfoContext

Provides AM session information and properties. When the SessionInfoFilter processes a request, it injects info and properties from the AM session into this context.

Properties

The context is named `amSession`, and is accessible at `${contexts.amSession}`. The context has the following properties:

"asJsonValue()": *json.JsonValue*

Raw JSON.

"latestAccessTime": *java.time.Instant*

The timestamp of when the session was last used. Can be null if the DN is not resident on the SSO token, or if the time cannot be obtained from the session.

"maxIdleExpirationTime": *java.time.Instant*

The timestamp of when the session would time out for inactivity. Can be null if the DN is not resident on the SSO token, or if the time cannot be obtained from the session.

"maxSessionExpirationTime": *java.time.Instant*

The timestamp of when the session would time out regardless of activity. Can be null if the DN is not resident on the SSO token, or if the time cannot be obtained from the session.

"properties": *java.util.Map*

A read-only map with the format `Map<String, String>`, where

- Key: Name of a property bound to the session
- Value: Value of the property

The following properties are retrieved:

- When `sessionProperties` in AmService is configured, listed session properties with a value.
- When `sessionProperties` in AmService is not configured, all session properties with a value.

- Properties with a value that are required by IG but not specified by `sessionProperties` in AmService. For example, when the session cache is enabled, session properties related to the cache are automatically retrieved.

Properties with a value are returned, properties with a null value are not returned

Can be empty, but not null.

`"realm"`: *java.lang.String*

The realm as specified by AM, in a user-friendly slash (/) separated format. Can be null if the DN is not resident on the SSO token.

`"sessionHandle"`: *java.lang.String*

The handle to use for logging out of the session. Can be null if the handle is not available for the session.

`"universalId"`: *java.lang.String*

The DN that AM uses to uniquely identify the user. Can be null if it cannot be obtained from the SSO token.

`"username"`: *java.lang.String*

A user-friendly version of the username. Can be null if the DN is not resident on the SSO token, or empty if it cannot be obtained from the DN.

More information

org.forgerock.openig.openam.SessionInfoContext

SsoTokenContext

Provides SSO tokens and their validation information. When the SingleSignOnFilter or CrossDomainSingleSignOnFilter processes a request, it injects the value of the SSO token and additional information in this context.

Properties

The context is named `ssoToken`, and is accessible at `${contexts.ssoToken}`. The context has the following properties:

`"info"`: *java.util.Map*

A map with the format `Map<String, Object>`, where

- Key: Property bound to the SSO token, such as `realm` or `uid`
- Value: Value of the property

Information associated with the SSO token, such as `realm` or `uid`. Cannot be null.

`"loginEndpoint"`: *java.lang.String*

A string representing the URL of the login endpoint, evaluated from the configuration of SingleSignOnFilter.

`"value"`: *java.lang.String*

The value of the SSO token. Cannot be null.

More information

org.forgerock.openig.openam.SsoTokenContext

StsContext

Provides the result of a token transformation. When the TokenTransformationFilter processes a request, it injects the result into this context.

Properties

The context is named `sts`, and is accessible at `${contexts.sts}`. The context has the following properties:

`"issuedToken"`: *java.lang.String*

The result of the token transformation.

More information

org.forgerock.openig.openam.StsContext

UriRouterContext

Provides routing information associated with a request. When IG routes a request, it injects information about the routing into this context.

Properties

The context is named `router`, and is accessible at `${contexts.router}`. The context has the following properties:

`"baseUri"`: *java.lang.String*

The portion of the request URI which has been routed so far.

`"matchedUri"`: *java.lang.String*

The portion of the request URI that matched the URI template.

`"originalUri"`: *URI*

The original target URI for the request, as received by IG. The value of this field is read-only.

`"remainingUri"`: *java.lang.String*

The portion of the request URI that is remaining to be matched.

`"uriTemplateVariables"`: *java.util.Map*

A map with the format `Map<String, String>`, where:

- Key: Name of a URI template variable
- Value: Value of a URI template variable

More information

org.forgerock.http.routing.UriRouterContext

UserProfileContext

When the UserProfileFilter processes a request, it injects the user profile information into this context. This context provides raw JSON representation, and convenience accessors that map commonly used LDAP field names to a context names.

Properties

The context is named `userProfile`, and is accessible at `${contexts.userProfile}`. The context has the following properties:

`"username"`: *java.lang.String*

User-friendly version of the username. This field is always fetched. If the underlying data store doesn't include `username`, this field is null.

Example of use: `${contexts.userProfile.username}`

`"realm"`: *java.lang.String*

Realm as specified by AM, in a user-friendly slash (/) separated format. Can be null.

Example of use: `${contexts.userProfile.realm}`

`"distinguishedName"`: *java.lang.String*

Distinguished name of the user. Can be null.

Example of use: `${contexts.userProfile.distinguishedName}`

`"commonName"`: *java.lang.String*

Common name of the user. Can be null.

Example of use: `${contexts.userProfile.commonName}`

`"rawInfo"`: *java.util.Map*

An unmodifiable map in the format `Map<String, Object>`, where:

- Key: Name of a field in an AM user profile
- Value: Value of a field in an AM user profile

This context contains the object structure of the AM user profile. Any individual field can be retrieved from the map. Depending on the requested fields, the context can be empty or values can be null.

Examples of use: `${contexts.userProfile.rawInfo}`, `${contexts.userProfile.rawInfo.username}`, `${contexts.userProfile.rawInfo.employeeNumber[0]}`.

`"asJsonValue()"`: *json.JsonValue*

User profile information structured as JSON.

Example of use: `${contexts.userProfile.asJsonValue()}`

More information

org.forgerock.openig.openam.UserProfileContext

UserProfileFilter

TransactionIdContext

The transaction ID of a request. When IG receives a request, it injects the transaction ID into this context.

Properties

The context is named `transactionId`, and is accessible at `${contexts.transactionId}`. The context has the following properties:

`"transactionId"`: *org.forgerock.services.TransactionId*

The ID of the transaction.

More information

org.forgerock.services.TransactionIdContext

org.forgerock.services.context.TransactionIdContext

Requests and responses

Request

An HTTP request message. Access the content of the request by using expressions.

Properties

`"method"`: *java.lang.String*

The HTTP method; for example, `GET`.

`"uri"`: *java.net.URI*

The fully-qualified URI of the resource being accessed; for example, `http://www.example.com/resource.txt`.

`"version"`: *java.lang.String*

The protocol version used for the request; for example, `HTTP/2`.

`"headers"`: *org.forgerock.http.protocol.Headers*

One or more headers in the request, with the format `header_name: [ header_value, … ]`. The following example accesses the first value of the request header `UserId`:

```
pass:[${request.headers['UserId'][0]}
```

"cookies": *org.forgerock.http.protocol.RequestCookies*

Incoming request cookies, with the format `cookie_name: [ cookie_value, … ]`. The following example accesses the first value of the request cookie `my-jwt`:

```
pass:[${request.cookies['my-jwt'][0].value}
```

"entity": *Entity*

The message body. The following example accesses the subject token from the request entity:

```
pass:[#{request.entity.form['subject_token'][0]}]
```

"queryParams": *Form*

Returns a copy of the query parameters decoded as a form. Modifications to the returned form are not reflected in the request.

More information

org.forgerock.http.protocol.Request

Response

An HTTP response message. Access the content of the response by using expressions.

Properties

"cause": *java.lang.Exception*

The cause of an error if the status code is in the range 4xx-5xx. Possibly null.

"status": *Status*

The response status.

"version": *java.lang.String*

The protocol version used the response; for example, `HTTP/2`.

"headers": *org.forgerock.http.protocol.Headers*

One or more headers in the response. The following example accesses the first value of the response header `Content-Type`:

```
pass:[${response.headers['Content-Type'][0]}]
```

"trailers": *org.forgerock.http.protocol.Headers*

One or more trailers in the response. The following example accesses the first value of the response trailer `Content-Length`:

```
pass:[${response.trailers['Content-Length'][0]}]
```

"entity": *Entity*

The message entity body. The following example accesses the user ID from the response:

```
pass:[#{toString(response.entity.json['userId'])}]
```

More information

org.forgerock.http.protocol.Response

Status

An HTTP response status.

Properties

"code": *integer*

Three-digit integer reflecting the HTTP status code.

"family": *enumeration*

Family Enum value representing the class of response that corresponds to the code:

Family.INFORMATIONAL

Status code reflects a provisional, informational response: 1xx.

Family.SUCCESSFUL

The server received, understood, accepted and processed the request successfully. Status code: 2xx.

Family.REDIRECTION

Status code indicates that the client must take additional action to complete the request: 3xx.

Family.CLIENT_ERROR

Status code reflects a client error: 4xx.

Family.SERVER_ERROR

Status code indicates a server-side error: 5xx.

`Family.UNKNOWN`

Status code does not belong to one of the known families: 600+.

`"reasonPhrase"`: *string*

The human-readable reason-phrase corresponding to the status code.

`"isClientError"`: *boolean*

True if Family.CLIENT_ERROR.

`"isInformational"`: *boolean*

True if Family.INFORMATIONAL.

`"isRedirection"`: *boolean*

True if Family.REDIRECTION.

`"isServerError"`: *boolean*

True if Family.SERVER_ERROR.

`"isSuccessful"`: *boolean*

True if Family.SUCCESSFUL.

More information

Response Status Codes.

org.forgerock.http.protocol.Status

URI

Represents a Uniform Resource Identifier (URI) reference.

Properties

`"scheme"`: *string*

The scheme component of the URI, or `null` if the scheme is undefined.

`"authority"`: *string*

The decoded authority component of the URI, or `null` if the authority is undefined.

Use "rawAuthority" to access the raw (encoded) component.

`"userInfo"`: *string*

The decoded user-information component of the URI, or `null` if the user information is undefined.

Use "rawUserInfo" to access the raw (encoded) component.

"host": *string*

The host component of the URI, or `null` if the host is undefined.

"port": *number*

The port component of the URI, or `null` if the port is undefined.

"path": *string*

The decoded path component of the URI, or `null` if the path is undefined.

Use "rawPath" to access the raw (encoded) component.

"query": *string*

The decoded query component of the URI, or `null` if the query is undefined.

> **Note**
>
> The query key and value is decoded. However, because a query value can be encoded more than once in a redirect chain, even though it is decoded it can contain unsafe ASCII characters.

Use "rawQuery" to access the raw (encoded) component.

"fragment": *string*

The decoded fragment component of the URI, or `null` if the fragment is undefined.

Use "rawFragment" to access the raw (encoded) component.

More information

org.forgerock.http.MutableUri

Access token resolvers

The following objects are available to resolve OAuth 2.0 access tokens:

TokenIntrospectionAccessTokenResolver

In OAuth2ResourceServerFilter, use the token introspection endpoint, `/oauth2/introspect`, to resolve access tokens and retrieve metadata about the token. The endpoint typically returns the time until the token expires, the OAuth 2.0 *scopes* associated with the token, and potentially other information.

The introspection endpoint is defined as a standard method for resolving access tokens, in RFC-7662, OAuth 2.0 Token Introspection.

Usage

Use this resolver with the `accessTokenResolver` property of OAuth2ResourceServerFilter.

```
"accessTokenResolver": {
  "type": "TokenIntrospectionAccessTokenResolver",
  "config": {
    "amService": AmService reference,          // Use either "amService"
    "endpoint": configuration expression<url>, // or "endpoint", but not both.
    "providerHandler": Handler reference
  }
}
```

Properties

`"amService"`: *AmService reference, required if* `endpoint` *is not configured*

The AmService heap object to use for the token introspection endpoint. The endpoint is extrapolated from the `url` property of the AmService.

When the Authorization Server is AM, use this property to define the token introspection endpoint.

If `amService` is configured, it takes precedence over `endpoint`.

See also AmService.

`"endpoint"`: *configuration expression<url>, required if* `amService` *is not configured*

The URI for the token introspection endpoint. Use `/oauth2/introspect`.

When the Authorization Server is not AM, use this property to define the token introspection endpoint.

If `amService` is configured, it takes precedence over `endpoint`.

`"providerHandler"`: *Handler reference, optional*

Invoke this HTTP client handler to send token info requests.

Provide either the name of a Handler object defined in the heap or an inline Handler configuration object.

Default: ForgeRockClientHandler

If you use the AM token introspection endpoint, this handler can be a `Chain` containing a `HeaderFilter` to add the authorization to the request header, as in the following example:

```
"providerHandler": {
  "type": "Chain",
  "config": {
    "filters": [
      {
        "type": "HeaderFilter",
        "config": {
          "messageType": "request",
          "add": {
            "Authorization": [ "Basic ${encodeBase64('<client_id>:<client_secret>')}" ]
          }
        }
      }
    ],
    "handler": "ForgeRockClientHandler"
  }
}
```

Example

For an example route that uses the token introspection endpoint, refer to Validate access tokens through the introspection endpoint.

More information

org.forgerock.openig.filter.oauth2.TokenIntrospectionAccessTokenResolverHeaplet

OAuth2ResourceServerFilter

StatelessAccessTokenResolver

Locally resolve and validate stateless access tokens issued by AM, without referring to AM.

AM can be configured to secure access tokens by signing or encrypting. The StatelessAccessTokenResolver must be configured for signature or encryption according to the AM configuration.

Usage

Use this resolver with the `accessTokenResolver` property of OAuth2ResourceServerFilter.

```
"accessTokenResolver": {
  "type": "StatelessAccessTokenResolver",
  "config": {
    "issuer": configuration expression<string>,
    "secretsProvider": SecretsProvider reference,
    "verificationSecretId": configuration expression<secret-id>, // Use "verificationSecretId" or
    "decryptionSecretId": configuration expression<secret-id>,   // "decryptionSecretId", but not both
    "skewAllowance": configuration expression<duration>
  }
}
```

Properties

`"issuer"`: *configuration expression<string>, required*

URI of the AM instance responsible for issuing access tokens.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for passwords and cryptographic keys.

`"verificationSecretId"`: *configuration expression<secret-id>, required if AM secures access tokens with a signature*

The secret ID for the secret used to verify the signature of signed access tokens.

This secret ID must point to a CryptoKey.

Depending on the type of secret store that is used to verify signatures, use the following values:

- For JwkSetSecretStore, use any non-empty string that conforms to the field convention for secret-id. The value of the string is not used.
- For other types of secret stores:
  - `null`: No signature verification is required.
  - A `kid` as a string: Signature verification is required with the provided `kid`. The StatelessAccessTokenResolver searches for the matching `kid` in the SecretsProvider.

For information about how signatures are validated, refer to Validate the signature of signed tokens. For information about how each type of secret store resolves named secrets, refer to Secrets.

Use either `verificationSecretId` or `decryptionSecretId`, according to the configuration of the token provider in AM. If AM is configured to sign **and** encrypt tokens, encryption takes precedence over signing.

`"decryptionSecretId"`: *configuration expression<secret-id>, required if AM secures access tokens with encryption*

The secret ID for the secret used to decrypt the JWT, for confidentiality.

This secret ID must point to a CryptoKey.

Use either `verificationSecretId` or `decryptionSecretId`, according to the configuration of the token provider in AM. If AM is configured to sign **and** encrypt the token, encryption takes precedence over signing.

`"skewAllowance"`: *configuration expression<duration>, optional*

The duration to add to the validity period of a JWT to allow for clock skew between different servers.

A `skewAllowance` of 2 minutes affects the validity period as follows:

- A JWT with an `iat` of 12:00 is valid from 11:58 on the IG clock.
- A JWT with an `exp` 13:00 is expired after 13:02 on the IG clock.

Default: To support a zero-trust policy, the skew allowance is by default `zero`.

Example

For examples of how to set up and use StatelessAccessTokenResolver to resolve signed and encrypted access tokens, refer to Validate stateless access tokens with the StatelessAccessTokenResolver.

More information

org.forgerock.openig.filter.oauth2.StatelessAccessTokenResolver

OAuth2ResourceServerFilter

ConfirmationKeyVerifierAccessTokenResolver

In OAuth2ResourceServerFilter, use the ConfirmationKeyVerifierAccessTokenResolver to verify that certificate-bound OAuth 2.0 bearer tokens presented by clients use the same mTLS-authenticated HTTP connection.

When a client obtains an access token from AM by using mTLS, AM can optionally use a confirmation key to bind the access token to a certificate. When the client connects to IG using that certificate, the ConfirmationKeyVerifierAccessTokenResolver verifies that the confirmation key corresponds to the certificate.

This proof-of-possession interaction ensures that only the client in possession of the key corresponding to the certificate can use the access token to access protected resources.

To use the ConfirmationKeyVerifierAccessTokenResolver, the following configuration is required in AM:

- OAuth 2.0 clients must be registered using an X.509 certificate, that is self-signed or signed in public key infrastructure (PKI)
- The AM client authentication method must be `self_signed_client_auth` or `tls_client_auth`.
- AM must be configured to bind a confirmation key to each client certificate.

The ConfirmationKeyVerifierAccessTokenResolver delegates the token resolution to a specified AccessTokenResolver, which retrieves the token information. The ConfirmationKeyVerifierAccessTokenResolver verifies the confirmation keys bound to the access token, and then acts as follows:

- If there is no confirmation key, pass the request down the chain.
- If the confirmation key matches the client certificate, pass the request down the chain.
- If the confirmation key doesn't match the client certificate, throw an error.
- If the confirmation key method is not supported by IG, throw an error.

For an example that uses the ConfirmationKeyVerifierAccessTokenResolver, refer to Validate Certificate-Bound Access Tokens.

For information about issuing certificate-bound OAuth 2.0 access tokens, refer to Certificate-bound proof-of-possession in AM's *OAuth 2.0 guide*. For information about authenticating an OAuth 2.0 client using mTLS certificates, refer to Mutual TLS in AM's *OAuth 2.0 guide* .

Usage

Use this resolver with the `accessTokenResolver` property of OAuth2ResourceServerFilter.

```
"accessTokenResolver": {
  "type": "ConfirmationKeyVerifierAccessTokenResolver",
  "config": {
    "delegate": AccessTokenResolver reference
  }
}
```

Properties

`"delegate"`: *AccessTokenResolver reference, required*

The access token resolver to use for resolving access tokens. Use any access token resolver described in Access token resolvers.

Examples

For an example that uses the ConfirmationKeyVerifierAccessTokenResolver with the following route, refer to Validate Certificate-Bound Access Tokens.

More information

org.forgerock.openig.filter.oauth2.cnf.ConfirmationKeyVerifierAccessTokenResolver

OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens

OAuth2ResourceServerFilter

ScriptableAccessTokenResolver

In OAuth2ResourceServerFilter, use a Groovy script to resolve access tokens against an Authorization Server.

Receive a string representing an access token and use a Groovy script to create an instance or promise of `org.forgerock.http.oauth2.AccessTokenInfo`.

Usage

Use this resolver with the `accessTokenResolver` property of OAuth2ResourceServerFilter.

```
"accessTokenResolver": {
  "type": "ScriptableAccessTokenResolver",
  "config": {
    "type": configuration expression<string>,
    "file": configuration expression<string>,  // Use either "file"
    "source": [ string, ... ],                 // or "source", but not both.
    "args": map,
    "clientHandler": Handler reference
  }
}
```

Properties

For information about properties for ScriptableAccessTokenResolver, refer to Scripts.

More information

org.forgerock.openig.filter.oauth2.ScriptableAccessTokenResolver

OAuth2ResourceServerFilter

CacheAccessTokenResolver

Enable and configure caching of OAuth 2.0 access tokens, based on *Caffeine*. For more information, refer to the GitHub entry, Caffeine.

This resolver configures caching of OAuth 2.0 access tokens, and delegates their resolution to another AccessTokenResolver. Use this resolver with AM or any OAuth 2.0 access token provider.

For an alternative way to cache OAuth 2.0 access tokens, configure the `cache` property of OAuth2ResourceServerFilter.

Usage

```
{
  "name": string,
  "type": "CacheAccessTokenResolver",
  "config": {
    "delegate": AccessTokenResolver reference,
    "enabled": configuration expression<boolean>,
    "defaultTimeout": configuration expression<duration>,
    "executor": Executor reference,
    "maximumSize": configuration expression<number>,
    "maximumTimeToCache": configuration expression<duration>,
    "amService": AmService reference,
    "onNotificationDisconnection": configuration expression<enumeration>
  }
}
```

Properties

`"delegate"`: *AccessTokenResolver reference, required*

Delegate access token resolution to one of the access token resolvers in Access token resolvers.

To use AM WebSocket notification to evict revoked access tokens from the cache, the delegate must be able to provide the token metadata required to update the cache.

- The `notification` property of AmService is enabled.
- The delegate AccessTokenResolver provides the token metadata required to update the cache.

`enabled`: *configuration expression<boolean>, optional*

Enable caching.

When an access token is cached, IG can reuse the token information without repeatedly asking the Authorization Server to verify the access token. When caching is disabled, IG must ask the Authorization Server to validate the access token for each request.

Default: `true`

`defaultTimeout`: *configuration expression<duration>, optional*

The duration for which to cache an OAuth 2.0 access token when it doesn't provide a valid expiry value or `maximumTimeToCache`.

If the `defaultTimeout` is longer than the `maximumTimeToCache`, then the `maximumTimeToCache` takes precedence.

Default: `1 minute`

`"executor"`: *Executor reference, optional*

An executor service to schedule the execution of tasks, such as the eviction of entries from the cache.

Default: `ForkJoinPool.commonPool()`

`"maximumSize"`: *configuration expression<number>, optional*

The maximum number of entries the cache can contain.

Default: Unlimited/unbound

`"maximumTimeToCache"`: *configuration expression<duration>, optional*

The maximum duration for which to cache access tokens.

Cached access tokens are expired according to their expiry time and `maximumTimeToCache`, as follows:

- If the expiry time is *before* the current time plus the `maximumTimeToCache`, the cached token is expired when the expiry time is reached.
- If the expiry time is *after* the current time plus the `maximumTimeToCache`, the cached token is expired when the `maximumTimeToCache` is reached

  The duration cannot be `zero` or `unlimited`.

Default: The token expiry time or `defaultTimeout`

`"amService"`: *AmService reference, optional*

An AmService to use for the WebSocket notification service.

When an access token is revoked on AM, the CacheAccessTokenResolver can delete the token from the cache when both of the following conditions are true:

- The `notification` property of AmService is enabled.
- The delegate AccessTokenResolver provides the token metadata required to update the cache.

  When a refresh_token is revoked on AM, all associated access tokens are automatically and immediately revoked.

See also AmService.

`onNotificationDisconnection`: *configuration expression<enumeration>, optional*

An `amService` must be configured for this property to have effect.

The strategy to manage the cache when the WebSocket notification service is disconnected, and IG receives no notifications for AM events. If the cache is not cleared it can become outdated, and IG can allow requests on revoked sessions or tokens.

Cached entries that expire naturally while the notification service is disconnected are removed from the cache.

Use one of the following values:

- `NEVER_CLEAR`
  - When the notification service is disconnected:
    - Continue to use the existing cache.
    - Deny access for requests that are not cached, but do not update the cache with these requests.
  - When the notification service is reconnected:
    - Continue to use the existing cache.
    - Query AM for incoming requests that are not found in the cache, and update the cache with these requests.
- `CLEAR_ON_DISCONNECT`
  - When the notification service is disconnected:
    - Clear the cache.
    - Deny access to all requests, but do not update the cache with these requests.
  - When the notification service is reconnected:
    - Query AM for all requests that are not found in the cache. (Because the cache was cleared, the cache is empty after reconnection.)
    - Update the cache with these requests.
- `CLEAR_ON_RECONNECT`
  - When the notification service is disconnected:
    - Continue to use the existing cache.
    - Deny access for requests that are not cached, but do not update the cache with these requests.
  - When the notification service is reconnected:
    - Query AM for all requests that are not found in the cache. (Because the cache was cleared, the cache is empty after reconnection.)
    - Update the cache with these requests.

Default: `CLEAR_ON_DISCONNECT`

Example

For an example that uses the CacheAccessTokenResolver, refer to Cache access tokens.

Caches

Session cache

When a user authenticates with AM, this cache stores information about the session. IG can reuse the information without asking AM to verify the session token (SSO token or CDSSO token) for each request.

If WebSocket notifications are enabled, the cache evicts entries based on session notifications from AM, making the cached content more accurate and reliable.

By default, session information isn't cached. To increase performance, consider enabling and configuring the cache. Find out more from `sessionCache` in AmService.

Policy cache

When the PolicyEnforcementFilter requests and receives a policy decision from AM, it stores the decision in the policy cache.

When a request matches a cached policy decision, IG can reuse the decision without asking AM for a new decision. When caching is disabled, IG must ask AM to make a decision for each request.

> **Tip**
>
> Maximize the cache hit ratio by using RequestResourceUriProvider or ScriptableResourceUriProvider in conjunction with AM policies. The PolicyEnforcementFilter identifies cached policy decisions by the resource URL returned by these URI providers.
> Find more information from the resourceUriProvider property of PolicyEnforcementFilter.

If WebSocket notifications are enabled, the cache evicts entries based on policy notifications from AM, making the cached content more accurate and reliable.

By default, policy decisions aren't cached.

User profile cache

When the UserProfileFilter retrieves user information, it caches it. IG can reuse the cached data without repeatedly querying AM to retrieve it.

By default, profile attributes aren't cached.

Access token cache

When a user presents an access token to the OAuth2ResourceServerFilter, the access token cache stores the token. IG can reuse the token information without asking the Authorization Server to verify the access token for each request.

By default, access tokens aren't cached. To increase performance by caching access tokens, consider configuring a cache in one of the following ways:

- Configure a CacheAccessTokenResolver for a cache based on Caffeine.
- Configure the `cache` property of OAuth2ResourceServerFilter.

Open ID Connect user information cache

When a downstream filter or handler needs user information from an OpenID Connect provider, IG fetches it lazily. By default, IG caches the information for 10 minutes to prevent repeated calls over a short time.

Find out more from `cacheExpiration` in AuthorizationCodeOAuth2ClientFilter.

Cache metrics at the Prometheus Scrape Endpoint

The Prometheus Scrape Endpoint exposes the following meters and metrics:

`ig_cache_gets_total`

A counter monitoring type, incremented when a cache request hits or misses an entry.

Label	Possible values
`content`	`session`, `policy_decision`, `user_profile`, `access_token`
`result`	`hit`, `miss`

Example:

```
ig_cache_gets_total{content="session",...result="hit",...} 13.0
ig_cache_gets_total{content="session",...,result="miss"...} 1.0
ig_cache_gets_total{content="policy_decision",...,result="hit",...} 5.0
ig_cache_gets_total{content="policy_decision",...,result="miss",...} 2.0
```

`ig_cache_loads`

This meter exposes the following metrics:

`ig_cache_loads_seconds`

A timer monitoring type, measuring the time in seconds spent successfully or unsuccessfully loading entries in the cache.

Label	Possible values
`content`	`session`, `policy_decision`, `user_profile`, `access_token`
`result`	`success`, `failure`
`quantile`	`0.5`, `0.75`, `0.95`, `0.98`, `0.99`, `0.999`

Example:

```
ig_cache_loads_seconds{content="session",...result="success",...quantile="0.5",} 0.057710516
ig_cache_loads_seconds{content="session",...result="success",...quantile="0.75",} 0.057710516
ig_cache_loads_seconds{content="session",...result="success",...quantile="0.95",} 0.057710516
ig_cache_loads_seconds{content="session",...result="success",...quantile="0.98",} 0.057710516
ig_cache_loads_seconds{content="session",...result="success",...quantile="0.99",} 0.057710516
ig_cache_loads_seconds{content="session",...result="success",...quantile="0.999",} 0.057710516
```

`ig_cache_loads_seconds_total`

A timer monitoring type, measuring the cumulated time in seconds spent successfully or unsuccessfully loading entries in the cache.

Label	Possible values
`content`	`session`, `policy_decision`, `user_profile`, `access_token`
`result`	`success`, `failure`

Example:

```
ig_cache_loads_seconds_total{content="session",...result="failure",...} 0.0
ig_cache_loads_seconds_total{content="session",...result="success",...} 0.057710516
ig_cache_loads_seconds_total{content="policy_decision",...,result="failure",...} 0.0
ig_cache_loads_seconds_total{content="policy_decision",...,result="success",...} 0.144314803
```

`ig_cache_loads_count`

A counter monitoring type, incremented when a cache request is successfully or unsuccessfully loaded in the cache.

Label	Possible values
`content`	`session`, `policy_decision`, `user_profile`, `access_token`
`result`	`success`, `failure`

Example:

```
ig_cache_loads_count{content="session",...result="failure",...} 0.0
ig_cache_loads_count{content="session",...result="success",...} 1.0
ig_cache_loads_count{content="policy_decision",...,result="failure",...} 0.0
ig_cache_loads_count{content="policy_decision",...,result="success",...} 2.0
```

`ig_cache_evictions`

This meter exposes the following metrics:

`ig_cache_evictions_count`

A counter monitoring type, incremented when an entry is evicted from the cache.

Label	Possible values
`content`	`session`, `policy_decision`, `user_profile`, `access_token`
`cause`	`COLLECTED`, `EXPIRED`, `EXPLICIT`, `REPLACED`, `SIZE`

Example

```
ig_cache_evictions_count{cause="COLLECTED",content="session",...} 0.0
ig_cache_evictions_total{cause="EXPIRED",content="session",...} 0.0
ig_cache_evictions_count{cause="EXPIRED",content="session",...} 0.0
ig_cache_evictions_total{cause="EXPLICIT",content="session",...} 0.0
ig_cache_evictions_count{cause="EXPLICIT",content="session",...} 0.0
ig_cache_evictions_total{cause="REPLACED",content="session",...} 0.0
ig_cache_evictions_count{cause="REPLACED",content="session",...} 0.0
ig_cache_evictions_total{cause="SIZE",content="session",...} 0.0
ig_cache_evictions_count{cause="SIZE",content="session",...} 0.0
ig_cache_evictions_total{cause="COLLECTED",content="policy_decision",...} 0.0
ig_cache_evictions_count{cause="COLLECTED",content="policy_decision",...} 0.0
ig_cache_evictions_total{cause="EXPIRED",content="policy_decision",...} 1.0
ig_cache_evictions_count{cause="EXPIRED",content="policy_decision",...} 1.0
ig_cache_evictions_total{cause="EXPLICIT",content="policy_decision",...} 0.0
ig_cache_evictions_count{cause="EXPLICIT",content="policy_decision",...} 0.0
ig_cache_evictions_total{cause="REPLACED",content="policy_decision",...} 0.0
ig_cache_evictions_count{cause="REPLACED",content="policy_decision",...} 0.0
ig_cache_evictions_total{cause="SIZE",content="policy_decision",...} 0.0
ig_cache_evictions_count{cause="SIZE",content="policy_decision",...} 0.0
```

`ig_cache_evictions_total`

A counter monitoring type, incremented when an entry is evicted from the cache. Each evicted entry has the weight `1`, so this metric is equal to `ig_cache_evictions_count`.

Label	Possible values
`content`	`session`, `policy_decision`, `user_profile`, `access_token`
`cause`	`COLLECTED`, `EXPIRED`, `EXPLICIT`, `REPLACED`, `SIZE`

Example

```
ig_cache_evictions_total{cause="COLLECTED",content="session",...} 0.0
ig_cache_evictions_count{cause="COLLECTED",content="session",...} 0.0
ig_cache_evictions_total{cause="EXPIRED",content="session",...} 0.0
ig_cache_evictions_count{cause="EXPIRED",content="session",...} 0.0
ig_cache_evictions_total{cause="EXPLICIT",content="session",...} 0.0
ig_cache_evictions_count{cause="EXPLICIT",content="session",...} 0.0
ig_cache_evictions_total{cause="REPLACED",content="session",...} 0.0
ig_cache_evictions_count{cause="REPLACED",content="session",...} 0.0
ig_cache_evictions_total{cause="SIZE",content="session",...} 0.0
ig_cache_evictions_count{cause="SIZE",content="session",...} 0.0
ig_cache_evictions_total{cause="COLLECTED",content="policy_decision",...} 0.0
ig_cache_evictions_count{cause="COLLECTED",content="policy_decision",...} 0.0
ig_cache_evictions_total{cause="EXPIRED",content="policy_decision",...} 1.0
ig_cache_evictions_count{cause="EXPIRED",content="policy_decision",...} 1.0
ig_cache_evictions_total{cause="EXPLICIT",content="policy_decision",...} 0.0
ig_cache_evictions_count{cause="EXPLICIT",content="policy_decision",...} 0.0
ig_cache_evictions_total{cause="REPLACED",content="policy_decision",...} 0.0
ig_cache_evictions_count{cause="REPLACED",content="policy_decision",...} 0.0
ig_cache_evictions_total{cause="SIZE",content="policy_decision",...} 0.0
ig_cache_evictions_count{cause="SIZE",content="policy_decision",...} 0.0
```

Secrets

IG uses the Commons Secrets API to manage secrets, such as passwords and cryptographic keys.

For more information about how IG manages secrets, refer to About secrets.

Base64EncodedSecretStore

Manage a repository of generic secrets, such as passwords or simple shared secrets, whose values are base64-encoded, and hard-coded in the route.

This Secret store can only manage the GenericSecret type.

The secrets provider queries the Base64EncodedSecretStore for a named secret, identified by the `secret-id` in the `"secret-id": "string"` pair. The Base64EncodedSecretStore returns the matching secret.

The secrets provider builds the secret, checking that the secret's constraints are met, and returns a unique secret. If the secret's constraints are not met, the secrets provider cannot build the secret and the secret query fails.

Secrets from Base64EncodedSecretStore never expire.

> **Important**
>
> Use Base64EncodedSecretStore for testing or evaluation only, to store passwords locally. In production, use an alternative secret store.

For a description of how secrets are managed, refer to About secrets

Usage

```
{
  "name": string,
  "type": "Base64EncodedSecretStore",
  "config": {
    "secrets": map or configuration expression<map>
  }
}
```

Properties

`"secrets"`: *map or configuration expression<map>, required*

Map of one or more data pairs with the format `Map<String, String>`, where:

- The key is the ID of a secret used in a route
- The value is the base64-encoded value of a secret, or a configuration expression that evaluates to the base64-encoded value of a secret

The following formats are allowed:

```
{
  "secrets": {
    "secret-id": "configuration expression<string>",
    ...
  }
}
```

```
{
  "secrets": "configuration expression<map>"
}
```

In the following example, the property is a map whose values are provided by strings:

```
{
  "type": "Base64EncodedSecretStore",
  "config": {
    "secrets": {
      "agent.password": "d2VsY29tZQ==",
      "crypto.header.key": "Y2hhbmdlaXQ="
    }
  }
}
```

In the following example, the property is a map whose values are provided by a configuration token and a configuration expression. The values are substituted when the route is loaded:

```
{
  "type": "Base64EncodedSecretStore",
  "config": {
    "secrets": {
      "agent.password": "&{secret.value|aGVsbG8=}",
      "crypto.header.key": "${readProperties('file.property')['b64.key.value']}"
    }
  }
}
```

Log level

To facilitate debugging secrets for the Base64EncodedSecretStore, in `logback.xml` add a logger defined by the fully qualified package name of the Base64EncodedSecretStore. The following line in `logback.xml` sets the log level to `ALL`:

```
<logger name="org.forgerock.openig.secrets.Base64EncodedSecretStore" level="ALL">
```

Example

For an example that uses Base64EncodedSecretStore, refer to `client-credentials.json` in Using OAuth 2.0 client credentials.

More information

Secrets

org.forgerock.openig.secrets.Base64EncodedSecretStore

FileSystemSecretStore

Manage a store of secrets held in files, specified as follows:

- Each file must contain only one secret.
- The file must be in the directory specified by the property `directory`.
- The filename must match the `mappings` property `secretId`.

- The file content must match the `mappings` property `format`. For example, if the mapping specifies `BASE64`, the file content must be base64-encoded.

This Secret store can manage secrets of both GenericSecret and CryptoKey types when used with dedicated formats.

Secrets are read lazily from the filesystem.

The secrets provider queries the FileSystemSecretStore for a named secret, identified by the name of a file in the specified directory, without the prefix/suffix defined in the store configuration. The FileSystemSecretStore returns the secret that exactly matches the name.

The secrets provider builds the secret, checking that the secret's constraints are met, and returns a unique secret. If the secret's constraints are not met, the secrets provider cannot build the secret and the secret query fails.

For a description of how secrets are managed, refer to About secrets

Usage

```
{
  "name": string,
  "type": "FileSystemSecretStore",
  "config": {
    "directory": configuration expression<string>,
    "format": SecretPropertyFormat reference,
    "suffix": configuration expression<string>,
    "mappings": [ object, ... ],
    "leaseExpiry": configuration expression<duration>,
    "autoRefresh": object
  }
}
```

Properties

`"directory"`: *configuration expression<string>, required*

File path to a directory containing secret files. This object checks the specified directory, but not its subdirectories.

`format`: *SecretPropertyFormat reference, optional*

Format in which the secret is stored. Use one of the following values or formats:

- `BASE64`: Base64-encoded
- `PLAIN`: Plain text
- A JwkPropertyFormat
- A PemPropertyFormat

Default: `BASE64`

`"suffix"`: *configuration expression<string>, optional*

File suffix.

When set, the FileSystemSecretStore will append that suffix to the secret ID and try to find a file with the mapped name.

Default: None

`"mappings"`: *array of objects, optional*

One or more mappings to define a secret:

`secretId`: *configuration expression<secret-id>, required*

The ID of the secret used in your configuration.

`format`: *SecretPropertyFormat reference, required*

The format and algorithm of the secret. Use SecretKeyPropertyFormat or PemPropertyFormat.

`"leaseExpiry"`: *configuration expression<duration>, optional*

The amount of time that secrets produced by this store can be cached before they must be refreshed.

If the duration is `zero` or `unlimited`, IG issues a warning, and uses the default value.

Default: 5 minutes

`"autoRefresh"`: *object, optional*

Automatically reload the FileSystemSecretStore when a file is edited or deleted in the directory given by `directory`.

When `autoRefresh` is triggered, secrets and keys are refreshed even if the `leaseExpiry` has not expired. When `autoRefresh` is triggered, the `leaseExpiry` is reset.

```
{
  "enabled": configuration expression<boolean>,
  "executor": ScheduledExecutorService reference
}
```

`enabled`: *configuration expression<boolean>, optional*

Flag to enable or disable automatic reload:

- `true`: Enable
- `false`: Disable

Default: `true`

`"executor"`: *ScheduledExecutorService reference, optional*

A ScheduledExecutorService to monitor the filesystem.

Default: The default ScheduledExecutorService in the heap

Log level

To facilitate debugging secrets for the FileSystemSecretStore, in `logback.xml` add a logger defined by the fully qualified package name of the property resolver. The following line in `logback.xml` sets the log level to `ALL`:

```
<logger name="org.forgerock.secrets.propertyresolver" level="ALL">
```

Example

For an example that uses FileSystemSecretStore, refer to Pass runtime data in a JWT signed with a PEM.

More information

Secrets

org.forgerock.openig.secrets.FileSystemSecretStoreHeaplet

HsmSecretStore

Manage a store of secrets with a hardware security module (HSM) device or a software emulation of an HSM device, such as SoftHSM.

This Secret store can only manage secrets of the CryptoKey type.

The secrets provider queries the HsmSecretStore for a named secret, identified by a secret ID and a stable ID, corresponding to the `secret-id` / `aliases` mapping. The HsmSecretStore returns a list of matching secrets.

The secrets provider builds the secret, checking that the secret's constraints are met, and returns a unique secret. If the secret's constraints are not met, the secrets provider cannot build the secret and the secret query fails.

For a description of how secrets are managed, refer to About secrets

Usage

```
{
  "name": string,
  "type": "HsmSecretStore",
  "config": {
    "providerName": configuration expression<string>,
    "storePasswordSecretId": configuration expression<secret-id>,
    "secretsProvider": SecretsProvider reference,
    "mappings": [ object, ... ],
    "leaseExpiry": configuration expression<duration>,
    "storePassword": configuration expression<secret-id> //deprecated
   }
}
```

Properties

`"providerName"`: *configuration expression<string>, required*

The name of the pre-installed Java Security Provider supporting an HSM. Use a physical HSM device, or a software emulation of an HSM device, such as SoftHSM.

For the SunPKCS11 provider, concatenate `"providerName"` with the prefix `SunPKCS11-`. For example, declare the following for the name `FooAccelerator`:

```
"providerName": "SunPKCS11-FooAccelerator"
```

`"storePasswordSecretId"`: *configuration expression<secret-id>, optional*

The secret ID of the password to access the HsmSecretStore.

This secret ID must point to a GenericSecret.

For information about how IG manages secrets, refer to About secrets.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for the `storePassword`.

`"mappings"`: *array of objects, required*

One or more mappings of one secret ID to one or more aliases.

```
"mappings" : {
  "secretId": configuration expression<secret-id>,
  "aliases": array of configuration expression<string>, //use aliases or
  "aliasesMatching": [ string, ... ] //aliasesMatching but not both
}
```

`"secretId"`: *configuration expression<secret-id>, required*

The secret ID of the key.

`"aliases"`: *array of configuration expression<strings>, required if* `aliasesMatching` *is not used*

One or more key aliases. Named aliases are mapped to the secret ID.

Use `aliases` or `aliasesMatching` but not both.

`"aliasesMatching"`: *array of <strings>, required if* `aliases` *is not used*

One or more regular expressions to match key aliases. Aliases that match the expressions are mapped to the secret ID.

Use `aliases` or `aliasesMatching` but not both.

Some KeyStores, such as a global Java TrustStore, can contain hundreds of valid certificates. Use this property to map multiple aliases to a secret ID without listing them all in the mapping.

The secret store uses the mappings as follows:

- When the secret is used to create signatures or encrypt values, the secret store uses the *active secret*, the first alias in the list.
- When the secret is used to verify signatures or decrypt data, the secret store tries all of the mapped aliases in the list, starting with the first, and stopping when it finds a secret that can successfully verify signature or decrypt the data.

The following example maps the named aliases to the named secret IDs:

```
"mappings": [
  {
    "secretId": "id.key.for.signing.jwt",
    "aliases": [ "signingkeyalias", "anothersigningkeyalias" ]
  },
  {
    "secretId": "id.key.for.encrypting.jwt",
    "aliases": ["encryptionkeyalias"]
  }
]
```

The following example maps aliases that match the regular expression `.*` to the named secret ID:

```
"mappings": [
  {
    "secretId": "id.key.for.signing.jwt",
    "aliasesMatching": [".*"]
  }
]
```

`secretId`: *configuration expression<secret-id>, required*

The ID of the secret used in your configuration.

`aliases`: *array of configuration expression<strings>, required*

One or more aliases for the secret ID. :leveloffset: +2

`"leaseExpiry"`: *configuration expression<duration>, optional*

The amount of time that secrets produced by this store can be cached before they must be refreshed.

If the duration is `zero` or `unlimited`, IG issues a warning, and uses the default value.

Default: 5 minutes

`"storePassword"`: *configuration expression<secret-id>, required if storePasswordSecretId not set*

> **Important**
>
> The use of this property is deprecated. If the KeyStore is password-protected, use `storePasswordSecretId`. For more information, refer to the Deprecated section of the *Release Notes*.

The secret ID of the password to access the HsmSecretStore.

+ For information about how IG manages secrets, refer to About secrets.

Log level

To facilitate debugging secrets for the HsmSecretStore, in `logback.xml` add a logger defined by the fully qualified package name of the HsmSecretStore. The following line in `logback.xml` sets the log level to `ALL`:

```
<logger name="org.forgerock.secrets.keystore" level="ALL">
```

Example

To set up this example:

1. Set up and test the example in JwtBuilderFilter, and then replace the KeyStoreSecretStore in that example with an HsmSecretStore.
2. Set an environment variable for the HsmSecretStore password, `storePassword`, and then restart IG.

   For example, if the HsmSecretStore password is `password`, set the following environment variable:

   ```
   export HSM_PIN='cGFzc3dvcmQ='
   ```

   The password is retrieved by the SystemAndEnvSecretStore, and must be base64-encoded.
3. Create a provider config file, as specified in the PKCS#11 Reference guide.
4. Depending on your version of Java, create a `java.security.ext` file for the IG instance, with the following content:

   ```
   security.provider.<number>=<provider-name> <path-to-provider-cfg-file>
   ```

   or

   ```
   security.provider.<number>=<class-name> <path-to-provider-cfg-file>
   ```
5. Start the IG JVM with the following system property that points to the provider config file:

   ```
   -Djava.security.properties=file://path-to-security-extension-file
   ```

The following example route is based on the examples in JwtBuilderFilter, replacing the KeyStoreSecretStore with an HsmSecretStore:

```
{
  "name": "hsm-jwt-signature",
  "condition": "${find(request.uri.path, '/hsm-jwt-signature$')}",
  "baseURI": "http://app.example.com:8081",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://am.example.com:8088/openam"
      }
    },
    {
      "name": "HsmSecretStore-1",
      "type": "HsmSecretStore",
      "config": {
        "providerName": "SunPKCS11-SoftHSM",
        "storePasswordSecretId": "hsm.pin",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "mappings": [{
          "secretId": "id.key.for.signing.jwt",
          "aliases": [ "signature-key" ]
        }]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "name": "SingleSignOnFilter-1",
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      }, {
        "name": "UserProfileFilter-1",
        "type": "UserProfileFilter",
        "config": {
          "username": "${contexts.ssoToken.info.uid}",
          "userProfileService": {
            "type": "UserProfileService",
            "config": {
              "amService": "AmService-1"
            }
          }
        }
      }, {
        "name": "JwtBuilderFilter-1",
        "type": "JwtBuilderFilter",
        "config": {
```

```
            "template": {
              "name": "${contexts.userProfile.commonName}",
              "email": "${contexts.userProfile.rawInfo.mail[0]}"
            },
            "secretsProvider": "HsmSecretStore-1",
            "signature": {
              "secretId": "id.key.for.signing.jwt"
            }
          }
        }, {
          "name": "HeaderFilter-1",
          "type": "HeaderFilter",
          "config": {
            "messageType": "REQUEST",
            "add": {
              "x-openig-user": ["${contexts.jwtBuilder.value}"]
            }
          }
        }],
        "handler": "ReverseProxyHandler"
      }
    }
  }
```

More information

Secrets

org.forgerock.openig.secrets.HsmSecretStoreHeaplet

JwkPropertyFormat

The format of a secret used with FileSystemSecretStore to decode JSON Web Key (JWK) formatted keys into secrets.

Usage

```
{
  "name": string,
  "type": "JwkPropertyFormat"
}
```

Example

```
{
  "heap" : [
    {
      "name": "jwkPropertyFormat",
      "type": "JwkPropertyFormat"
    },
    {
      "name": "MySecretStore",
      "type": "FileSystemSecretStore",
      "config": {
        "format": "jwkPropertyFormat",
        "directory": "&{secretsDir}/idcloud",
        "mappings": [ {
          "secretId": "my.privateKey.jwk",
          "format": "jwkPropertyFormat"
        }
      ]
    }
  }
}
```

More information

org.forgerock.secrets.propertyresolver.JwkPropertyFormat

JwkSetSecretStore

A secret store of JSON Web Keys (JWK) from a local or remote JWK Set.

This Secret store can only manage secrets of the CryptoKey type.

The secrets provider builds the secret, checking that the secret's constraints are met, and returns a unique secret. If the secret's constraints are not met, the secrets provider cannot build the secret and the secret query fails.

For a description of how secrets are managed, refer to About secrets.

For information about JWKs and JWK Sets, refer to JSON Web Key (JWK).

Usage

```
{
  "name": string,
  "type": "JwkSetSecretStore",
  "config": {
    "jwkUrl": configuration expression<url>,
    "handler": Handler reference,
    "cacheTimeout": configuration expression<duration>,
    "cacheMissCacheTime": configuration expression<duration>,
    "leaseExpiry": configuration expression<duration>
  }
}
```

Properties

`"jwkUrl"`: *configuration expression<url>, required*

A URL that contains the client's public keys in JWK format.

`"handler"`: *Handler reference, optional*

An HTTP client handler to communicate with the `jwkUrl`.

Usually set this property to the name of a ClientHandler configured in the heap, or a chain that ends in a ClientHandler.

Default: ClientHandler

`"cacheTimeout"`: *configuration expression<duration>, optional*

Delay before the cache is reloaded. The cache contains the `jwkUrl`.

The cache cannot be deactivated. If a value lower than 10 seconds is configured, a warning is logged and the default value is used instead.

Default: 2 minutes

`"cacheMissCacheTime"`: *configuration expression<duration>, optional*

If the `jwkUrl` is looked up in the cache and is not found, this is the delay before the cache is reloaded.

Default: 2 minutes

`"leaseExpiry"`: *configuration expression<duration>, optional*

The amount of time that secrets produced by this store can be cached before they must be refreshed.

If the duration is `zero` or `unlimited`, IG issues a warning, and uses the default value.

Default: 5 minutes

Log level

To facilitate debugging secrets for the JwkSetSecretStore, in `logback.xml` add a logger defined by the fully qualified package name of the JwkSetSecretStore. The following line in `logback.xml` sets the log level to `ALL`:

```
<logger name="org.forgerock.secrets.jwkset" level="ALL">
```

Example

For an example of how to set up and use JwkSetSecretStore to validate signed access tokens, refer to Validate signed access_tokens with the StatelessAccessTokenResolver and JwkSetSecretStore.

In the following example, a StatelessAccessTokenResolver validates a signed access token by using a JwkSetSecretStore:

```
"accessTokenResolver": {
  "type": "StatelessAccessTokenResolver",
  "config": {
    "secretsProvider": {
      "type": "JwkSetSecretStore",
      "config": {
        "jwkUrl": "http://am.example.com:8088/openam/oauth2/connect/jwk_uri"
      },
      "issuer": "http://am.example.com:8088/openam/oauth2",
      "verificationSecretId": "verification.secret.id"
    }
  }
}
```

The JWT signature is validated as follows:

- If the JWT contains a `kid` with a matching secret in the JWK set:
  - The secrets provider queries the JwkSetSecretStore for a named secret.
  - The JwkSetSecretStore returns the matching secret, identified by a stable ID.
  - The StatelessAccessTokenResolver tries to validate the signature with that named secret. If it fails, the token is considered as invalid.

  In the route, note that the property `verificationSecretId` must be configured but is not used in named secret resolution.
- If the JWT contains a `kid` without a matching secret in the JWK set:
  - The secrets provider queries the JwkSetSecretStore for a named secret.
  - Because the referenced JWK set doesn't contain a matching secret, named secret resolution fails. IG tries valid secret resolution in the same way as when the JWT doesn't contain a `kid`.
- If the JWT doesn't contain a `kid`:
  - The secrets provider queries the JwkSetSecretStore for list of valid secrets, whose secret ID is `verification.secret.id`.
  - The JwkSetSecretStore returns all secrets in the JWK set whose purpose is signature verification. For example, signature verification keys can have the following JWK parameters:

    ```
    {
      "use": "sig"
    }
    ```

    ```
    {
      "key_opts": [ "verify" ]
    }
    ```

    Secrets are returned in the order that they are listed in the JWK set.

- The StatelessAccessTokenResolver tries to validate the signature with each secret sequentially, starting with the first, and stopping when it succeeds.
- If none of the valid secrets can verify the signature, the token is considered as invalid.

More information

org.forgerock.openig.secrets.JwkSetSecretStoreHeaplet

JSON Web Key (JWK)

KeyStoreSecretStore

Manages a secret store for cryptographic keys and certificates, based on a standard Java keystore.

> **Warning**
>
> Legacy keystore types such as JKS and JCEKS are supported but are not secure. Consider using the PKCS#12 keystore type.

This Secret store can only manage secrets of the CryptoKey type.

The secrets provider queries the KeyStoreSecretStore for a named secret, identified by a secret ID and a stable ID, corresponding to the `secret-id` / `aliases` mapping. The KeyStoreSecretStore returns a secret that exactly matches the name, and whose purpose matches the secret ID and any purpose constraints.

The secrets provider builds the secret, checking that the secret's constraints are met, and returns a unique secret. If the secret's constraints are not met, the secrets provider cannot build the secret and the secret query fails.

For a description of how secrets are managed, refer to About secrets

Usage

```
{
  "name": string,
  "type": "KeyStoreSecretStore",
  "config": {
    "file": configuration expression<string>,
    "storeType": configuration expression<string>,
    "storePasswordSecretId": configuration expression<string>,
    "entryPasswordSecretId": configuration expression<string>,
    "secretsProvider": SecretsProvider reference,
    "mappings": [ object, ... ],
    "leaseExpiry": configuration expression<duration>,
    "autoRefresh": object,
    "storePassword": configuration expression<string>, //deprecated
    "keyEntryPassword": configuration expression<string> //deprecated
   }
}
```

Properties

`"file"`: *configuration expression<string>, required*

The path to the KeyStore file.

`"storeType"`: *configuration expression<string>, optional*

The secret store type.

Default: PKCS12

`"storePasswordSecretId"`: *configuration expression<secret-id>, optional*

The secret ID of the password to access the KeyStore.

This secret ID must point to a GenericSecret.

IG searches for the value of the password until it finds it, first locally, then in parent routes, then in `config.json`.

To create a store password, add a file containing the password. The filename must corresponds to the secret ID, and the file content must contain only the password, with no trailing spaces or carriage returns.

Default: None; the KeyStore is not password-protected

`"entryPasswordSecretId"`: *configuration expression<secret-id>, optional*

The secret ID of the password to access entries in the KeyStore.

This secret ID must point to a GenericSecret.

To create an entry password, add a file containing the password. The filename must correspond to the secret ID, and the file content must contain only the password, with no trailing spaces or carriage returns.

When this property is used, the password must be the same for all entries in the KeyStore. If the KeyStore uses different passwords for entries, `entryPasswordSecretId` doesn't work.

Default: The value of `storePasswordSecretId`

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for the keystore password and key entry password.

`"mappings"`: *array of objects, required*

One or more mappings of one secret ID to one or more aliases.

```
"mappings" : {
  "secretId": configuration expression<secret-id>,
  "aliases": array of configuration expression<string>, //use aliases or
  "aliasesMatching": [ string, ... ] //aliasesMatching but not both
}
```

`"secretId"`: *configuration expression<secret-id>, required*

The secret ID of the key.

`"aliases"`: *array of configuration expression<strings>, required if* `aliasesMatching` *is not used*

One or more key aliases. Named aliases are mapped to the secret ID.

Use `aliases` or `aliasesMatching` but not both.

`"aliasesMatching"`: *array of <strings>, required if* `aliases` *is not used*

One or more regular expressions to match key aliases. Aliases that match the expressions are mapped to the secret ID.

Use `aliases` or `aliasesMatching` but not both.

Some KeyStores, such as a global Java TrustStore, can contain hundreds of valid certificates. Use this property to map multiple aliases to a secret ID without listing them all in the mapping.

The secret store uses the mappings as follows:

- When the secret is used to create signatures or encrypt values, the secret store uses the *active secret*, the first alias in the list.
- When the secret is used to verify signatures or decrypt data, the secret store tries all of the mapped aliases in the list, starting with the first, and stopping when it finds a secret that can successfully verify signature or decrypt the data.

The following example maps the named aliases to the named secret IDs:

```
"mappings": [
  {
    "secretId": "id.key.for.signing.jwt",
    "aliases": [ "signingkeyalias", "anothersigningkeyalias" ]
  },
  {
    "secretId": "id.key.for.encrypting.jwt",
    "aliases": ["encryptionkeyalias"]
  }
]
```

The following example maps aliases that match the regular expression `.*` to the named secret ID:

```
"mappings": [
  {
    "secretId": "id.key.for.signing.jwt",
    "aliasesMatching": [".*"]
  }
]
```

`secretId`: *configuration expression<secret-id>, required*

The ID of the secret used in your configuration.

`aliases`: *array of configuration expression<strings>, required*

One or more aliases for the secret ID. :leveloffset: +2

`"leaseExpiry"`: *configuration expression<duration>, optional*

The amount of time that secrets produced by this store can be cached before they must be refreshed.

If the duration is `zero` or `unlimited`, IG issues a warning, and uses the default value.

Default: 5 minutes

`"autoRefresh"`: *object, optional*

Automatically reload the KeystoreSecretStore when the keystore is edited or deleted.

```
{
  "enabled": configuration expression<boolean>,
  "executor": ScheduledExecutorService reference
}
```

`enabled`: *configuration expression<boolean>, optional*

Flag to enable or disable automatic reload:

- `true`: Enable
- `false`: Disable

Default: `true`

`"executor"`: *ScheduledExecutorService reference, optional*

A ScheduledExecutorService to monitor the keystore.

Default: The default ScheduledExecutorService in the heap

`"storePassword"`: *configuration expression<secret-id>, required*

> **Important**
>
> This property is deprecated. If the KeyStore is password-protected, use `storePasswordSecretId`. For more information, refer to the Deprecated section of the *Release Notes*.

The secret ID of the password to access the KeyStore.

This secret ID must point to a GenericSecret.

IG searches for the value of the password until it finds it, first locally, then in parent routes, then in `config.json`.

To create a store password, add a file containing the password. The filename must corresponds to the secret ID, and the file content must contain only the password, with no trailing spaces or carriage returns.

`"keyEntryPassword"`*: configuration expression<secret-id>, optional*

> **Important**
>
> This property is deprecated; use the `entryPasswordSecretId` instead. For more information, refer to the Deprecated section of the *Release Notes*.

The secret ID of the password to access entries in the KeyStore.

This secret ID must point to a GenericSecret.

To create an entry password, add a file containing the password. The filename must correspond to the secret ID, and the file content must contain only the password, with no trailing spaces or carriage returns.

When this property is used, the password must be the same for all entries in the keystore. If the keystore uses different passwords for entries, `keyEntryPassword` doesn't work.

Default: The value of `storePassword`

Log level

To facilitate debugging secrets for the KeyStoreSecretStore, in `logback.xml` add a logger defined by the fully qualified package name of the KeyStoreSecretStore. The following line in `logback.xml` sets the log level to `ALL`:

```
<logger name="org.forgerock.secrets.keystore" level="ALL">
```

Example

For examples of routes that use KeyStoreSecretStore, see the examples in JwtBuilderFilter.

In the following example, a StatelessAccessTokenResolver validates a signed access token by using a KeyStoreSecretStore:

```
"accessTokenResolver": {
  "type": "StatelessAccessTokenResolver",
  "config": {
    "secretsProvider": {
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "IG_keystore.p12",
        "storeType": "PKCS12",
        "storePasswordSecretId": "keystore.secret.id",
        "entryPasswordSecretId": "keystore.secret.id",
        "mappings": [{
          "secretId": "verification.secret.id",
          "aliases": [ "verification.key.1", "verification.key.2" ]
        }]
      },
      "issuer": "http://am.example.com:8088/openam/oauth2",
      "verificationSecretId": "verification.secret.id"
    }
  }
}
```

The JWT signature is validated as follows:

- If the JWT contains a `kid` with a mapped value, for example `verification.key.1`:
  - The secrets provider queries the KeyStoreSecretStore for a named secret with the secret ID `verification.secret.id` and the stable ID `verification.key.1`.
  - Because the KeyStoreSecretStore contains that mapping, the KeyStoreSecretStore returns a named secret.
  - The StatelessAccessTokenResolver tries to validate the JWT signature with the named secret. If it fails, the token is considered as invalid.
- If the JWT contains a `kid` with an unmapped value, for example, `verification.key.3`:
  - The secrets provider queries the KeyStoreSecretStore for a named secret with the secret ID `verification.secret.id` and the stable ID `verification.key.3`.
  - Because the KeyStoreSecretStore doesn't contain that mapping, named secret resolution fails. IG tries valid secret resolution in the same way as when the JWT doesn't contain a `kid`.
- If the JWT doesn't contain a `kid`:
  - The secrets provider queries the KeyStoreSecretStore for all valid secrets, whose alias is mapped to the secret ID `verification.secret.id`. There are two valid secrets, with aliases `verification.key.1` and `verification.key.2`.
  - The StatelessAccessTokenResolver first tries to verify the signature with `verification.key.1`. If that fails, it tries `verification.key.2`.
  - If neither of the valid secrets can verify the signature, the token is considered as invalid.

More information

org.forgerock.secrets.keystore.KeyStoreSecretStore

org.forgerock.openig.secrets.KeyStoreSecretStoreHeaplet

PemPropertyFormat

The format of a secret used with a mappings configuration in FileSystemSecretStore and SystemAndEnvSecretStore. Privacy-Enhanced Mail (PEM) is a file format for storing and sending cryptographic keys, certificates, and other data, based on standards in Textual Encodings of PKIX, PKCS, and CMS Structures. By default, OpenSSL generates keys using the PEM format.

Encryption methods and ciphers used for PEM encryption must be supported by the Java Cryptography Extension.

PEM keys have the following format, where the PEM label is associated to the type of stored cryptographic material:

```
-----BEGIN {PEM label}-----
Base64-encoded cryptographic material
-----END {PEM label}-----
```

PEM Label	Stored Cryptographic Material
`CERTIFICATE`	X.509 Certificate
`PUBLIC KEY`	X.509 SubjectPublicKeyInfo
`PRIVATE KEY`	PKCS#8 Private Key
`ENCRYPTED PRIVATE KEY`	Encrypted PKCS#8 Private Key
`EC PRIVATE KEY`	EC Private Key
`RSA PRIVATE KEY`	PKCS#1 RSA Private Key
`RSA PUBLIC KEY`	PKCS#1 RSA Public Keys
`DSA PRIVATE KEY`	PKCS#1-style DSA Private Key
`HMAC SECRET KEY`	HMAC Secret Keys
`AES SECRET KEY`	AES Secret Keys
`GENERIC SECRET`	Generic Secrets (passwords, API keys, etc)

Note the following points about the key formats:

- PKCS#1 is the standard that defines RSA. For more information, refer to RFC 8017: RSA Public Key Syntax.
- PKCS#1-style DSA and EC keys are not defined in any standard, but are adapted from the RSA format.
- `HMAC SECRET KEY`, `AES SECRET KEY`, and `GENERIC SECRET` are a ForgeRock extension, and not currently supported by any other tools.

The following example is non-standard PEM encoding of an HMAC symmetric secret key. The payload is base64-encoded random bytes that are the key material, with no extra encoding.

```
-----BEGIN HMAC SECRET KEY-----
Pj/Vel...thB0U=
-----END HMAC SECRET KEY-----
```

Run the following example command to create the key:

```
cat <<EOF
-----BEGIN HMAC SECRET KEY-----
$(head -c32 /dev/urandom | base64)
-----END HMAC SECRET KEY-----
EOF
```

Usage

```
{
  "name": string,
  "type": "PemPropertyFormat",
  "config": {
   "decryptionSecretId": configuration expression<secret-id>,
   "secretsProvider": SecretsProvider reference
  }
}
```

Properties

`"decryptionSecretId"`: *configuration expression*<*secret-id*>, *optional*

The secret ID for the secret to decrypt a PKCS#8 private key.

This secret ID must point to a GenericSecret.

`"secretsProvider"`: *SecretsProvider reference*, *required when* `decryptionSecretId` *is used*

The SecretsProvider to query for the decryption secret.

Example

For examples of use, see Pass runtime data in a JWT signed with a PEM and Pass runtime data in a JWT signed and encrypted with a PEM.

More information

org.forgerock.openig.secrets.PemPropertyFormatHeaplet

SecretsKeyManager

Uses the Commons Secrets API to manage keys that authenticate a TLS connection to a peer. The configuration references the keystore that holds the keys.

Usage

```
{
   "name": string,
   "type": "SecretsKeyManager",
   "config": {
     "signingSecretId": configuration expression<secret-id>,
     "secretsProvider": SecretsProvider reference
 }
}
```

Properties

`"signingSecretId"`*: configuration expression<secret-id>, required*

The secret ID used to retrieve private signing keys.

This secret ID must point to a CryptoKey.

`"secretsProvider"`*: SecretsProvider reference, required*

The SecretsProvider to query for secrets to resolve the private signing key.

Example

The following example uses a private key found from a keystore for TLS handshake.

```
{
  "type": "SecretsKeyManager",
  "config": {
    "signingSecretId": "key.manager.secret.id",
    "secretsProvider": {
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "path/to/certs/ig.example.com.p12",
        "storePasswordSecretId": "keystore.pass",
        "secretsProvider": "SecretsPasswords",
        "mappings": [{
          "secretId": "key.manager.secret.id",
          "aliases": [ "ig.example.com" ]
        }]
      }
    }
  }
}
```

More information

Secrets

org.forgerock.openig.secrets.SecretsKeyManagerHeaplet

SecretKeyPropertyFormat

The format of a secret used with a secret store.

Usage

```
{
  "name": string,
  "type": "SecretKeyPropertyFormat",
  "config": {
    "format": SecretPropertyFormat reference,
    "algorithm": configuration expression<string>
  }
}
```

Properties

`format`: *SecretPropertyFormat *reference*, optional*

Format in which the secret is stored. Use one of the following values, or define a format:

- `BASE64`: Base64-encoded
- `PLAIN`: Plain text

Default: `BASE64`

`"algorithm"`: *configuration expression<string>, required*

The algorithm name used for encryption and decryption. Use algorithm names given in Java Security Standard Algorithm Names.

Example

```
{
  "type": "SecretKeyPropertyFormat",
  "config": {
    "format": "PLAIN",
    "algorithm": "AES"
  }
}
```

More information

Secrets

org.forgerock.openig.secrets.SecretKeyPropertyFormatHeaplet

SecretsProvider

Uses the specified secret stores to resolve queried secrets, such as passwords and cryptographic keys. Attempts to resolve the secret with the secret stores in the order that they are declared in the array.

Usage

```
{
  "name": string,
  "type": "SecretsProvider",
  "config": {
    "stores": [ SecretStore reference, ... ]
  }
}
```

This object can alternatively be configured in a compact format, without the SecretsProvider declaration, as follows:

- With an inline secret store:

```
"secretsProvider": {
  "type": "secret store type1",
  "config": {...}
}
```

- With multiple inline secret stores:

```
"secretsProvider": [
  {
    "type": "secret store type1",
    "config": {...}
  }
  {
    "type": "secret store type2",
    "config": {...}
  }
]
```

- With a referenced secret store:

```
"secretsProvider": "mySecretStore1"
```

- With multiple referenced secret stores:

```
"secretsProvider": [
  "mySecretStore1", "mySecretStore2"
]
```

See Example for more example configurations.

Properties

`"stores"`: *array of SecretStore references, required*

One or more secret stores to provide access to stored secrets. Configure secret stores described in Secrets.

Example

The following SecretsProvider is used in Discover and dynamically register with OpenID connect providers.

```
"secretsProvider": {
  "type": "SecretsProvider",
  "config": {
    "stores": [
      {
        "type": "KeyStoreSecretStore",
        "config": {
          "file": "/path/to/keystore.p12",
          "mappings": [
            {
              "aliases": [ "myprivatekeyalias" ],
              "secretId": "private.key.jwt.signing.key"
            }
          ],
          "storePasswordSecretId": "keystore.secret.id",
          "storeType": "PKCS12",
          "secretsProvider": "SystemAndEnvSecretStore-1"
        }
      }
    ]
  }
}
```

The following example shows the equivalent SecretsProvider configuration with an inline compact format:

```
"secretsProvider": {
  "name": "KeyStoreSecretStore-1",
  "type": "KeyStoreSecretStore",
  "config": {
    "file": "/path/to/keystore.p12",
    "mappings": [
      {
        "aliases": [ "myprivatekeyalias" ],
        "secretId": "private.key.jwt.signing.key"
      }
    ],
    "storePasswordSecretId": "keystore.secret.id",
    "storeType": "PKCS12"
    "secretsProvider": "SystemAndEnvSecretStore-1"
  }
}
```

The following example shows the equivalent SecretsProvider configuration with a compact format, referencing a KeyStoreSecretStore object in the heap:

```
"secretsProvider": "KeyStoreSecretStore-1"
```

More information

StatelessAccessTokenResolver

Secrets

org.forgerock.secrets.SecretsProvider

SecretsTrustManager

Uses the Commons Secrets API to manage trust material that verifies the credentials presented by a peer. Trust material is usually public key certificates. The configuration references the secrets store that holds the trust material.

Usage

```
{
  "name": string,
  "type": "SecretsTrustManager",
  "config": {
    "verificationSecretId": configuration expression<secret-id>,
    "certificateVerificationSecretId": configuration expression<secret-id>,
    "secretsProvider": SecretsProvider reference,
    "checkRevocation": configuration expression<boolean>
  }
}
```

Properties

`"verificationSecretId"`: *configuration expression<secret-id>, required if certificateVerificationSecretId isn't used*

Either `verificationSecretId` or `certificateVerificationSecretId` is required.

The secret ID to retrieve trusted certificates. This secret ID must point to a CryptoKey.

Consider the following requirements for using certificates with `verificationSecretId`:

- Certificates loaded from keystores can be used with the following constraint:
  - The KeyUsage extension `digitalSignature` must be set **or** no KeyUsage extension must be set
- Certificates loaded from JWKs or JWK sets can be used with the following constraints:
  - The `use` parameter must be set to `sig` **or** the `use` parameter must not be set
  - The `key_ops` parameter must contain `verify` **or** the `key_ops` parameter must not be set
- Certificates loaded from PEM can be used without constraint.

`"certificateVerificationSecretId"`: *configuration expression<secret-id>, required if verificationSecretId isn't used*

Either `verificationSecretId` or `certificateVerificationSecretId` is required.

The secret ID to retrieve certificates for trusted certificate authorities (CA). Use this property when you trust client certificates **only because** they are signed by a trusted CA.

Consider the following requirements:

- Certificates loaded from keystores can be used with the following constraint:
  - The KeyUsage extension `keyCertSign` must be set **or** no KeyUsage extension must be set
- Certificates loaded from JWKs or JWK sets can be used with the following constraints:
  - The `use` parameter must not be set
  - The `key_ops` parameter must not be set
- Certificates loaded from PEM can be used without constraint.

`"secretsProvider"`: *SecretsProvider reference, required*

The SecretsProvider to query for secrets to resolve trusted certificates.

`"checkRevocation"`: *configuration expression<boolean>, optional*

Specifies whether to check for certificate revocation.

Default: `true`

Example

The following example trusts a list of certificates found in a given keystore:

```
{
  "name": "SecretsTrustManager-1",
  "type": "SecretsTrustManager",
  "config": {
    "verificationSecretId": "trust.manager.secret.id",
    "secretsProvider": {
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "path/to/certs/truststore.p12",
        "storePasswordSecretId": "keystore.pass",
        "secretsProvider": "SecretsPasswords",
        "mappings": [{
          "secretId": "trust.manager.secret.id",
          "aliases": [ "alias-of-trusted-cert-1", "alias-of-trusted-cert-2" ]
        }]
      }
    }
  }
}
```

The following example trusts a list of CA-signed certificates found in a given keystore:

```
{
  "type": "SecretsTrustManager",
  "config": {
    "certificateVerificationSecretId": "ca.secret.id",
    "secretsProvider": {
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "&{ig.instance.dir}/certs/truststore.p12",
        "storePassword": "keystore.pass",
        "secretsProvider": "SecretsPasswords",
        "mappings": [{
          "secretId": "ca.secret.id",
          "aliases": [ "alias-of-trusted-cacert-1", "alias-of-trusted-cacert-2" ]
        }]
      }
    }
  }
}
```

More information

Secrets

org.forgerock.openig.secrets.SecretsTrustManagerHeaplet

SystemAndEnvSecretStore

Manage a store of secrets from system properties and environment variables.

This secret store can manage GenericSecret and CryptoKey secret types when used with dedicated formats.

A secret ID must conform to the convention described in secret-id. The reference is then transformed to match the environment variable name, as follows:

- Periods (.) are converted to underscores.
- Characters are transformed to uppercase.

For example, `my.secret.id` is transformed to `MY_SECRET_ID`.

The secrets provider queries the SystemAndEnvSecretStore for a named secret, identified by the name of a system property or environment variable. The SystemAndEnvSecretStore returns a secret that exactly matches the name.

The secrets provider builds the secret, checking that the secret's constraints are met, and returns a unique secret. If the secret's constraints are not met, the secrets provider cannot build the secret and the secret query fails.

For a description of how secrets are managed, refer to About secrets

Usage

```
{
  "name": string,
  "type": "SystemAndEnvSecretStore",
  "config": {
    "format": SecretPropertyFormat reference,
    "mappings": [ object, ... ],
    "leaseExpiry": configuration expression<duration>
    }
}
```

Properties

`format`: *SecretPropertyFormat reference, optional*

Format in which the secret is stored. Use one of the following values, or define a format:

- `BASE64`: Base64-encoded
- `PLAIN`: Plain text

Default: `BASE64`

`"mappings"`: *array of objects, optional*

One or more mappings to define a secret:

`secretId`: *configuration expression<secret-id>, required*

The ID of the secret used in your configuration.

`format`: *SecretPropertyFormat reference, required*

The format and algorithm of the secret. Use SecretKeyPropertyFormat or PemPropertyFormat.

`"leaseExpiry"`: *configuration expression<duration>, optional*

The amount of time that secrets produced by this store can be cached before they must be refreshed.

If the duration is `zero` or `unlimited`, IG issues a warning, and uses the default value.

Default: 5 minutes

Log level

To facilitate debugging secrets for the SystemAndEnvSecretStore, in `logback.xml` add a logger defined by the fully qualified package name of the property resolver. The following line in `logback.xml` sets the log level to `ALL`:

```
<logger name="org.forgerock.secrets.propertyresolver" level="ALL">
```

Example

For an example of how to uses a SystemAndEnvSecretStore to manage a password, refer to the example in Authenticate with SSO through the default authentication service

More information

Secrets

org.forgerock.openig.secrets.SystemAndEnvSecretStoreHeaplet

TrustManager (deprecated)

> **Important**
>
> This object is deprecated; use SecretsTrustManager instead. For more information, refer to the Deprecated section of the *Release Notes*.

The configuration of a Java Secure Socket Extension TrustManager to manage trust material (typically X.509 public key certificates) for IG. The configuration references the keystore that holds the trust material.

When IG acts as a client, it uses a trust manager to verify that the server is trusted. When IG acts as a server, it uses a trust manager to verify that the client is trusted.

Usage

```
{
    "name": string,
    "type": "TrustManager",
    "config": {
      "keystore": KeyStore reference,
      "alg": configuration expression<string>
    }
}
```

Properties

`"keystore"`: *KeyStore reference, required*

The KeyStore (deprecated) object that references the store for key certificates. When `keystore` is used in a KeyManager, it queries for private keys; when `keystore` is used in a TrustManager, it queries for certificates.

Provide either the name of the keystore object defined in the heap or an inline keystore configuration object.

`"alg"`: *configuration expression<string>, optional*

The certificate algorithm to use.

Default: the default for the platform, such as `SunX509`.

Example

The following example configures a trust manager that depends on a KeyStore configuration. This configuration uses the default certificate algorithm:

```
{
  "name": "MyTrustManager",
  "type": "TrustManager",
  "config": {
    "keystore": {
      "type": "KeyStore",
      "config": {
        "url": "file://${env['HOME']}/keystore.p12",
        "passwordSecretId": "${system['keypass']}",
        "secretsProvider": "SystemAndEnvSecretStore"
      }
    }
  }
}
```

More information

org.forgerock.openig.security.TrustManagerHeaplet

JSSE reference guide, KeyManager, KeyStore

Supported standards

IG implements the following RFCs, Internet-Drafts, and standards:

OAuth 2.0

- RFC 6749: The OAuth 2.0 Authorization Framework
- RFC 6750: The OAuth 2.0 Authorization Framework: Bearer Token Usage
- RFC 7515: JSON Web Signature (JWS)
- RFC 7516: JSON Web Encryption (JWE)
- RFC 7517: JSON Web Key (JWK)
- RFC 7518: JSON Web Algorithms (JWA)
- RFC 7519: JSON Web Token (JWT)
- RFC 7523: JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication and Authorization Grants
- RFC 7591: OAuth 2.0 Dynamic Client Registration Protocol
- RFC 7662: OAuth 2.0 Token Introspection
- RFC 7800: Proof-of-Possession Key Semantics for JSON Web Tokens (JWTs)

RFC 8705: OAuth 2.0 Mutual-TLS Client Authentication and Certificate-Bound Access Tokens

OpenID Connect 1.0

IG can be configured to play the role of OpenID Connect relying party. The OpenID Connect specifications depend on OAuth 2.0, JSON Web Token, Simple Web Discovery and related specifications. The following specifications make up OpenID Connect 1.0.

- OpenID Connect Core 1.0 defines core OpenID Connect 1.0 features.

  > **Note**
  >
  > In section 5.6 of the specification, IG supports *Normal Claims*. The optional *Aggregated Claims* and *Distributed Claims* representations are not supported by IG.

- OpenID Connect Discovery 1.0 defines how clients can dynamically discover information about OpenID Connect providers.
- OpenID Connect Dynamic Client Registration 1.0 defines how clients can dynamically register with OpenID Connect providers.
- OAuth 2.0 Multiple Response Type Encoding Practices defines additional OAuth 2.0 response types used in OpenID Connect.

User-Managed Access (UMA) 2.0

User-Managed Access (UMA) 2.0 Grant for OAuth 2.0 Authorization

Federated Authorization for User-Managed Access (UMA) 2.0

Representational State Transfer (REST)

Style of software architecture for web-based, distributed systems. IG's APIs are RESTful APIs.

Security Assertion Markup Language (SAML)

Standard, XML-based framework for implementing a SAML service provider. IG supports multiple versions of SAML including 2.0, 1.1, and 1.0.

Specifications are available from the OASIS standards page.

Other Standards

RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON). JSON text is encoded with Unicode; IG reads and stores JSON as Unicode.

RFC 2616: Hypertext Transfer Protocol — HTTP/1.1.

RFC 2617: HTTP Authentication: Basic and Digest Access Authentication, supported as an authentication module.

RFC 5280: Internet X.509 Public Key Infrastructure Certificate, supported for certificate-based authentication.

RFC 5785: Defining Well-Known Uniform Resource Identifiers (URIs).

RFC 6265: HTTP State Management Mechanism regarding HTTP Cookies and `Set-Cookie` header fields.

Internationalization

IG supports internationalization (i18n) to facilitate localization for target audiences that vary in culture, region, or language.

Information type	Character set/encoding
HTTP header names and values	US-ASCII
HTTP trailer names and values	US-ASCII
Response entities for StaticResponseHandler	The `Content-Type` header must be set. For text content, the character set must also be specified; for example: • `Content-Type: text/html; charset=utf-8` • `Content-Type: text/plain; charset=utf-8` The entity must conform to the content type.
Text in request and response entities for CaptureDecorator	If the `Content-Type` header is set for the request or response, the decorator uses it to decode the text in request or response messages, and then writes them to the logs. If the `Content-Type` header is not set, the decorator does not write the request or response messages to the logs.
Logs	The system default character set where IG is running. To use a different character set, configure `logback.xml` as described in Change the character set and format of log messages.
IG configuration files	UTF-8
Hostnames	US-ASCII Non US-ASCII characters must be escaped with Punycode encoding.
URIs	US-ASCII Non US-ASCII and reserved characters must be escaped with percent-encoding.