FORGEROCK®

Gateway Guide
/ ForgeRock Identity Gateway 6.1

Latest update: 6.1.0

Paul Bryan
Mark Craig
Jamie Nelson
Guillaume Sauthier
Joanne Henry

Copyright © 2011-2018 ForgeRock AS.

Abstract

Instructions for installing and configuring ForgeRock® Identity Gateway.

Table of Contents

Preface

ForgeRock Identity Platform™ serves as the basis for our simple and comprehensive Identity and Access Management solution. We help our customers deepen their relationships with their customers, and improve the productivity and connectivity of their employees and partners. For more information about ForgeRock and about the platform, see https://www.forgerock.com.

1. About This Guide

IG integrates web applications, APIs, and microservices with the ForgeRock Identity Platform, without modifying the application or the container where they run. Based on reverse proxy architecture, it enforces security and access control in conjunction with the Access Management modules.

This guide is for access management designers and administrators who develop, build, deploy, and maintain IG for their organizations. It helps you to get started quickly, and learn more as you progress through the guide.

This guide assumes basic 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:

- Lightweight Directory Access Protocol (LDAP) if you use IG with LDAP directory services

- 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

2. Formatting Conventions

Most examples in the documentation are created in GNU/Linux or Mac OS X operating environments. If distinctions are necessary between operating environments, examples are labeled with the operating environment name in parentheses. To avoid repetition file system directory names are often given only in UNIX format as in `/path/to/server`, even if the text applies to `C:\path\to\server` as well.

Absolute path names usually begin with the placeholder `/path/to/`. This path might translate to `/opt/`, `C:\Program Files\`, or somewhere else on your system.

Command-line, terminal sessions are formatted as follows:

```
$ echo $JAVA_HOME
/path/to/jdk
```

Command output is sometimes formatted for narrower, more readable output even though formatting parameters are not shown in the command.

Program listings are formatted as follows:

```
class Test {
    public static void main(String [] args)  {
        System.out.println("This is a program listing.");
    }
}
```

3. Accessing Documentation Online

ForgeRock publishes comprehensive documentation online:

- The ForgeRock Knowledge Base offers a large and increasing number of up-to-date, practical articles that help you deploy and manage ForgeRock software.

  While many articles are visible to community members, ForgeRock customers have access to much more, including advanced information for customers using ForgeRock software in a mission-critical capacity.

- ForgeRock 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.

4. Using the ForgeRock.org Site

The ForgeRock.org site has links to source code for ForgeRock open source software, as well as links to the ForgeRock forums and technical blogs.

If you are a *ForgeRock customer*, raise a support ticket instead of using the forums. ForgeRock support professionals will get in touch to help you.

5. Getting Support and Contacting ForgeRock

ForgeRock provides support services, professional services, training through ForgeRock University, and partner services to assist you in setting up and maintaining your deployments. For a general overview of these services, see https://www.forgerock.com.

ForgeRock has staff members around the globe who support our international customers and partners. For details, visit https://www.forgerock.com, or send an email to ForgeRock at info@forgerock.com.

FORGEROCK

Chapter 1
About Identity Gateway

This chapter sets out the essentials of using IG, including:

- What problems IG solves and where it fits in your deployment

- How IG acts on HTTP requests and responses

- How the configuration files for IG are organized

- The roles played by routes, filters, handlers, and chains, which are the building blocks of an IG configuration

1.1. About IG

Most organizations have valuable existing services that are not easily integrated into newer architectures. These existing services cannot often be changed. Many client applications cannot communicate as they lack a gateway to bridge the gap. "Missing Gateway" illustrates one example of a missing gateway.

Missing Gateway
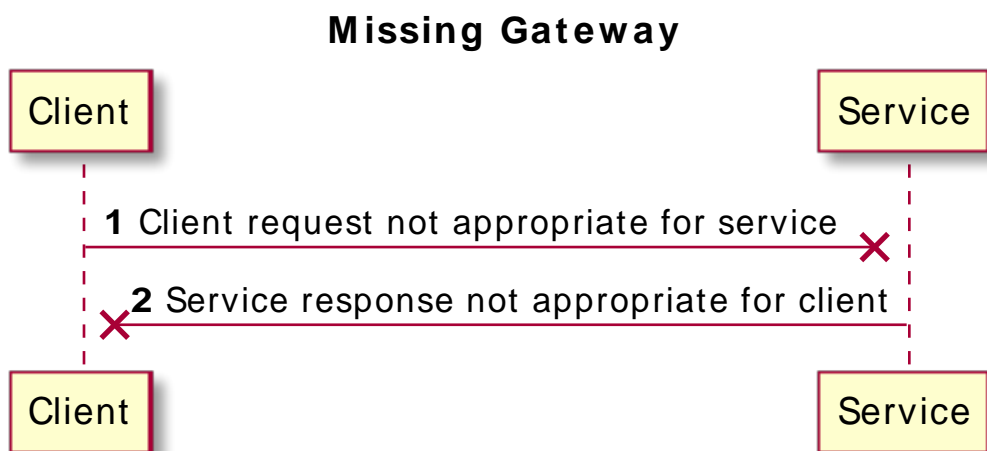
IG works as an HTTP gateway, based on reverse proxy architecture. IG is deployed on a network so it can intercept both client requests and server responses. "IG Deployed" illustrates a IG deployment.

IG Deployed

IG Deployed
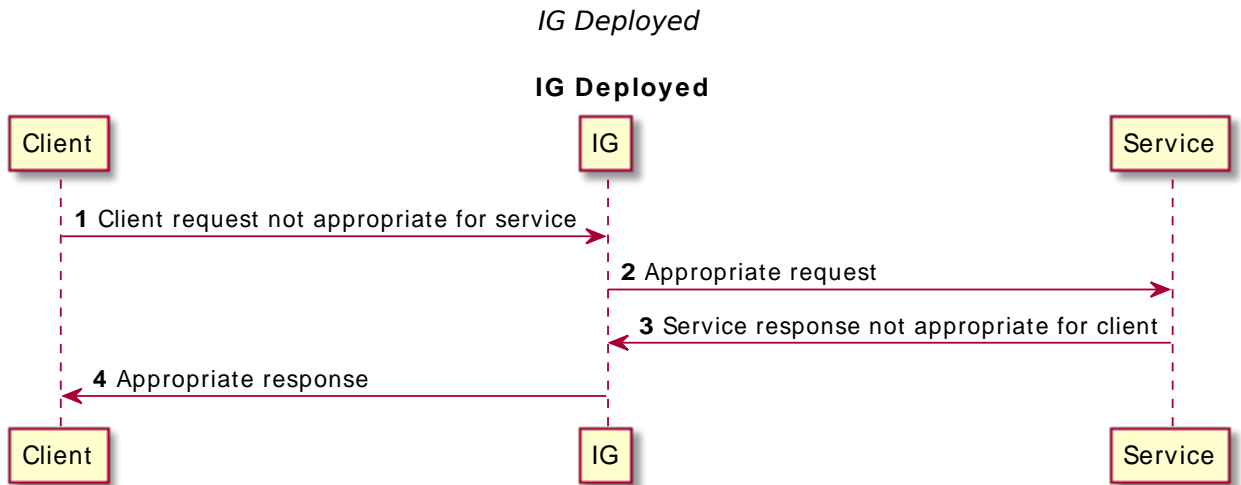
Clients interact with protected servers through IG. IG can be configured to add new capabilities to existing services without affecting current clients or servers.

The list that follows features you can add to your solution by using IG:

• 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 supports these capabilities as out of the box configuration options. Once you understand the essential concepts covered in this chapter, try the additional instructions in this guide to use IG to add other features.

1.2. IG Object Model

IG handles HTTP requests and responses in user-defined chains, making it possible to manage and to monitor processing at any point in a chain. The IG object model provides both access to the requests and responses that pass through each chain, and also context information associated with each request.

Contexts provide information about the client making the request, the session, the authentication or authorization identity of the principal, and any other state information associated with the request. Contexts provide a means to access state information throughout the duration of the HTTP session between the client and protected application, including when this involves interaction with additional services.

1.3. Configuration Directories and Files

By default, IG configuration files are located under `$HOME/.openig` on Linux, macOS, and UNIX systems, and `%appdata%\OpenIG` on Windows systems. For information about how to change the default locations , see "Changing the Default Location of the Configuration Folders".

IG uses the following configuration directories:

- `$HOME/.openig/config`, `%appdata%\OpenIG\config`

  IG administration and gateway configuration files. For information, see AdminHttpApplication(5) in the *Configuration Reference* and GatewayHttpApplication(5) in the *Configuration Reference*.

- `$HOME/.openig/config/routes`, `%appdata%\OpenIG\config\routes`

  IG route configuration files. For more information see "*Configuring Routers and Routes*".

- `$HOME/.openig/SAML`, `%appdata%\OpenIG\SAML`

  IG SAML 2.0 configuration files. For more information see "*Acting As a SAML 2.0 Service Provider*".

- `$HOME/.openig/scripts/groovy`, `%appdata%\OpenIG\scripts\groovy`

  IG script files, for Groovy scripted filters and handlers. For more information see "*Extending Identity Gateway*".

- `$HOME/.openig/tmp`, `%appdata%\OpenIG\tmp`

  IG temporary files. This location can be used for temporary storage.

1.4. Routing and Routes

Routers are handlers that perform the following tasks:

- Define the routes directory and loads routes into the IG configuration.

- Depending on the scanning interval, periodically scan the routes directory and updates the IG configuration when routes are added, removed, or changed.

- Route requests to the first route in the IG configuration whose condition is satisfied.

Routes are configuration files that you add to IG to manage requests. They are flat files in JSON format. You can add routes in the following ways:

- Manually into the filesystem.

- Through Common REST commands. For information, see " Creating and Editing Routes Through Common REST ".

- Through Studio. For information, see "*Configuring Routes With Studio*" in the *Getting Started Guide*.

Every route must call a handler to process the request and produce a response to a request.

When a route has a condition, it can handle only requests that meet the condition. When a route has no condition, it can handle any request.

Routes inherit settings from their parent configurations. 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.

For examples of route configurations see "*Configuring Routers and Routes*". For information about the parameters for routers and routes, see Router(5) in the *Configuration Reference* and Route(5) in the *Configuration Reference*.

1.5. Handlers, Filters, and Chains

Handlers and filters are chained together to modify a request, the response, or the context:

- *Handler*: Either delegates to another handler, or produces a response.

  One way to produce a response is to send a request to and receive a response from an external service. In this case, IG acts as a client of the service, often on behalf of the client whose request initiated the request.

  Another way to produce a response is to build a response either statically or based on something in the context. In this case, IG plays the role of server, generating a response to return to the client.

  For more information, see Handlers in the *Configuration Reference*.

- *Filter*: Either transforms data in the request, response, or context, or performs an action when the request or response passes through the filter.

A filter can leave the request, response, and contexts unchanged. For example, it can log the context as it passes through the filter. Alternatively, it can change request or response. For example, it can generate a static request to replace the client request, add a header to the request, or remove a header from a response.

For more information, see Filters in the *Configuration Reference*.

- *Chain*: A type of handler that dispatches processing to an ordered list of filters, and then to the handler.

  A `Chain` can be placed anywhere in a configuration that a handler can be placed. Filters process the incoming request, pass it on to the next filter, and then to the handler. After the handler produces a response, the filters process the outgoing response 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.

  For more information, see Chain(5) in the *Configuration Reference*.

- *Chain of Filters*: A type of filter that dispatches processing to an ordered list of filters without then dispatching the request to a handler. 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 anywhere in a configuration that a filter can be placed.

  For more information, see ChainOfFilters(5) in the *Configuration Reference*.

"Flow Inside a Chain" shows the flow inside a `Chain`, where a request filter transforms the request, a handler sends the request to a protected application, and then a response filter transforms the response. Notice how the flow traverses the filters in reverse order when the response comes back from the handler.

Flow Inside a Chain
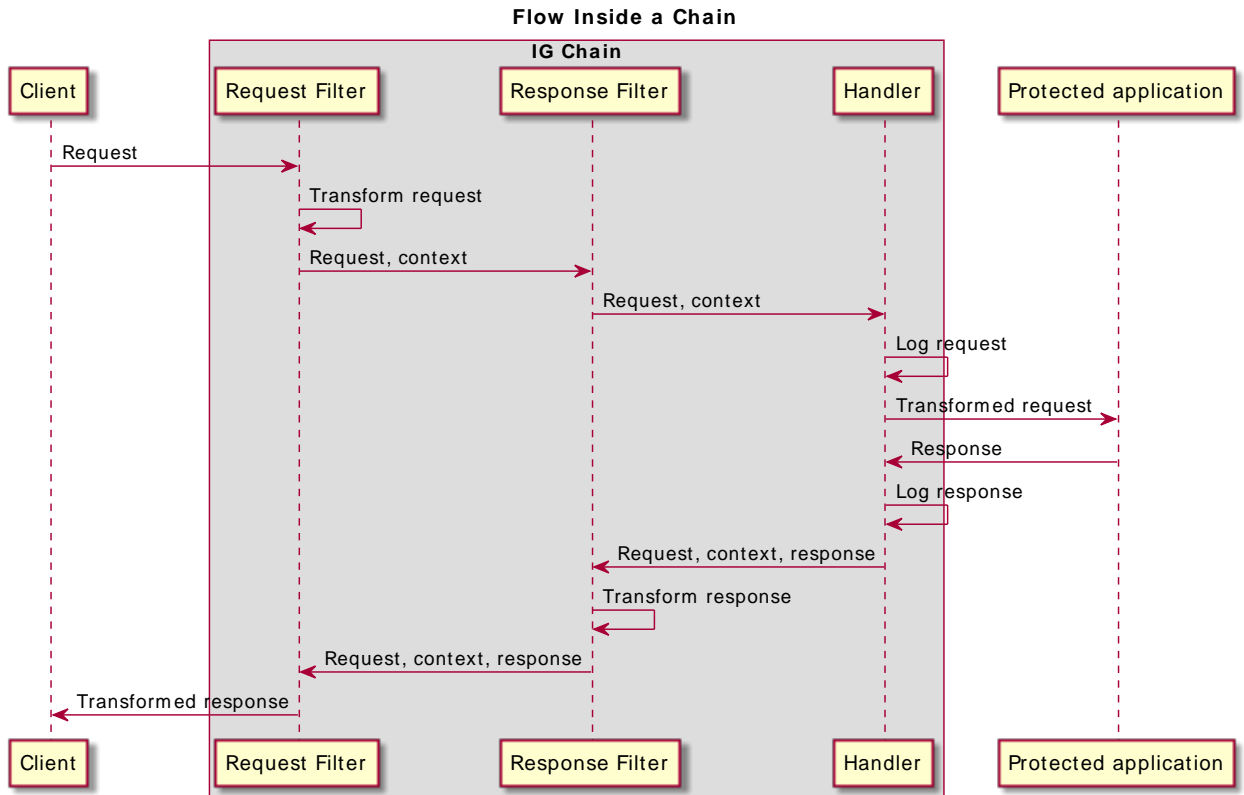
The route configuration in "Chain to a Protected Application" demonstrates the flow through a chain to a protected application. With IG and the sample application set up as described in "*First Steps*" in the *Getting Started Guide*, access this route on http://openig.example.com:8080/home/chain.

Chain to a Protected Application

```
{
  "condition": "${matches(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` passes 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.

"Requests and Responses in a Chain" list 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`.

Requests and Responses in a Chain

```
### Original request from user-agent
GET http://openig.example.com:8080/home/chain HTTP/1.1
Accept: */*
Host: openig.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: openig.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
```

1.6. Configuration Objects

Configuration objects have the following parts:

- *Name*: a unique string in the list of objects. When you declare inline objects, the name is not required.

- *Type*: the type name of the configuration object. IG defines many object types for different purposes.

- *Config*: additional configuration settings. The content of the configuration object depends on its type.

  If all of the configuration settings for the type are optional, the config field is also optional. The following configurations signify that the object uses default settings:

  - Omitting the config field

  - Setting the config field to an empty object, `"config": {}`

  - Setting `"config": null`

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.

1.7. Decorators

Decorators are additional heap objects to extend what another object can do. For example, a *CaptureDecorator* extends the capability of filters and handlers to log requests and responses. A *TimerDecorator* logs processing times. Decorate configuration objects with decorator names as field names.

IG defines the following decorators: `audit`, `baseURI`, `capture`, and `timer`. You can use these decorators without configuring them explicitly.

You can log requests, responses, and processing times by adding decorations as shown in the following example:

```
{
  "handler": {
  "type": "Router",
  "capture": [ "request", "response" ],
  "timer": true
  }
}
```

For more information, see Decorators in the *Configuration Reference*.

1.8. Configuration Parameters Declared as Property Variables

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.

For more information, see Properties(5) in the *Configuration Reference*.

1.9. Using Comments in IG Configuration Files

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

Use the following conventions when commenting to ensure your configuration files are easier to read:

• Use `comment` fields to add text comments. "Using a Comment Field" illustrates a CaptureDecorator configuration that includes a text comment.

Using a Comment Field

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

- Use an underscore (_) to comment a field temporarily. "Using an Underscore" illustrates a `CaptureDecorator` that has `"captureEntity": true` commented out. As a result, it uses the default setting (`"captureEntity": false`).

Using an Underscore

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

1.10. Understanding IG APIs With 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 help you discover and understand APIs, 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 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`. 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 cannot 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, see "Common REST API Documentation" in the *Configuration Reference*.

Retrieving API Descriptors for a Router

With IG running as described in "*First Steps*" in the *Getting Started Guide*, run the following query to generate a JSON that describes the router operations supported by the endpoint:

```
http://openig.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.

Retrieving API Descriptors for the UMA Service

With the UMA tutorial running as described in "*Supporting UMA Resource Servers*", run the following query to generate a JSON that describes the UMA share API:

```
http://openig.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.

Retrieving API Descriptors for the Monitoring Endpoint of a Route

With monitoring enabled for a route, run a query to generate a JSON that exposes monitoring information for the route. For example:

```
http://openig.example.com:8080/openig/api/system/objects/_router/routes/00-uma/monitoring?_api
{
    "swagger": "2.0",
    "info": {
    "version": "IG version",
    "title": "IG"
    },
    "host": "openig.example.com:8080",
    "basePath": "/openig/api/system/objects/_router/routes/00-uma/monitoring",
    "tags": [{
    "name": "Monitoring endpoint"
    }],
    . . .
```

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

For information about how to set up monitoring for a route, see "To Monitor the IG Route Monitoring Endpoint".

Retrieving 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:

```
http://openig.example.com:8080/openig/api/system/objects/_router?_api
{
    "swagger": "2.0",
    "info": {
    "version": "IG version",
    "title": "IG"
    },
    "host": "openig.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.

Retrieving API Descriptors for an IG Instance

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

```
http://openig.example.com:8080/openig/api?_api
{
    "swagger": "2.0",
    "info": {
    "version": "IG version",
    "title": "IG"
    },
    "host": "openig.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 are not included in the returned JSON.

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

Chapter 2
Installation in Detail

For information about how to quickly install and configure IG, see "*First Steps*" in the *Getting Started Guide*. This chapter contains information about how to do the following tasks:

- Prepare a deployment container for use with IG ("Configuring Deployment Containers").

- Prepare the network so that traffic passes through IG ("Preparing the Network").

- Download, deploy, and configure IG ("Installing IG").

- Change the locations of the configuration files ("Changing the Default Location of the Configuration Folders").

- Prepare for load balancing with IG ("Preparing For Load Balancing and Failover").

- Secure connections to and from IG ("Configuring IG For HTTPS (Client-Side)").

- Use IG JSON Web Token (JWT) Session cookies across multiple servers ("Setting Up Keys For JWT Encryption").

- Prevent further updates to the configuration ("Making the Configuration Immutable").

Before you begin to install or configure IG, make sure that you are using a supported container and version of Java. For information about requirements for running IG, see "*Before You Install*" in the *Release Notes*.

2.1. Configuring Deployment Containers

This section provides installation and configuration tips that you need to run IG in supported containers.

For the full list of supported containers see "Web Application Containers" in the *Release Notes*.

For information about advanced configuration for a container, see the container documentation.

2.1.1. About Securing Connections

IG is often deployed to replay credentials or other security information. In a real world deployment, that information must be communicated over a secure connection using HTTPS, meaning in effect

HTTP over encrypted Transport Layer Security (TLS). Never send real credentials, bearer tokens, or other security information unprotected over HTTP.

When IG is acting as a server, the web application container where IG runs is responsible for setting up TLS connections with client applications that connect to IG. For details, see "Configuring Jetty For HTTPS (Server-Side)" or "Configuring Tomcat For HTTPS (Server-Side)".

When IG is acting as a client, the ReverseProxyHandler configuration governs TLS connections from IG to other servers. For details, see "Configuring IG For HTTPS (Client-Side)" and ReverseProxyHandler(5) in the *Configuration Reference*.

TLS depends on the use of digital certificates (public keys). In typical use of TLS, the client authenticates the server by its X.509 digital certificate as the first step to establishing communication. Once trust is established, then the client and server can set up a symmetric key to encrypt communications.

In order for the client to trust the server certificate, the client needs first to trust the certificate of the party who signed the server's certificate. This means that either the client has a trusted copy of the signer's certificate, or the client has a trusted copy of the certificate of the party who signed the signer's certificate.

Certificate Authorities (CAs) are trusted signers with well-known certificates. Browsers generally ship with many well-known CA certificates. Java distributions also ship with many well-known CA certificates. Getting a certificate signed by a well-known CA is often expensive.

It is also possible for you to self-sign certificates. The trade-off is that although there is no monetary expense, the certificate is not trusted by any clients until they have a copy. Whereas it is often enough to install a certificate signed by a well-known CA in the server keystore as the basis of trust for HTTPS connections, self-signed certificates must also be installed in all clients.

Like self-signed certificates, the signing certificates of less well-known CAs are also unlikely to be found in the default truststore. You might therefore need to install those signing certificates on the client side as well.

This guide describes how to install self-signed certificates, which are certainly fine for trying out the software and okay for deployments where you manage all clients that access IG. If you need a well-known CA-signed certificate instead, see the documentation for your container for details on requesting a CA signature and installing the CA-signed certificate.

Once certificates are properly installed to allow client-server trust, also consider the cipher suites configured for use. The cipher suite used determines the security settings for the communication. Initial TLS negotiations bring the client and server to agreement on which cipher suite to use. Basically the client and server share their preferred cipher suites to compare and to choose. If you therefore have a preference concerning the cipher suites to use, you must set up your container to use only your preferred cipher suites. Otherwise the container is likely to inherit the list of cipher suites from the underlying Java environment.

The Java Secure Socket Extension (JSSE), part of the Java environment, provides security services that IG uses to secure connections. You can set security and system properties to configure the JSSE.

For a list of properties you can use to customize the JSSE in Oracle Java, see the *Customization* section of the *JSSE Reference Guide*.

2.1.2. Apache Tomcat For IG

This section describes essential Tomcat configuration that you need in order to run IG.

Download and install a supported version of Tomcat from http://tomcat.apache.org/.

> **Important**
>
> If you use startup scripts to bootstrap the IG web container, the scripts can start the container process with a different user. To prevent errors, make sure that the location of the IG configuration is correct. Alternatively, adapt the startup scripts to specify the `IG_INSTANCE_DIR` env variable or `ig.instance.dir` system properties, taking care to set file permissions correctly.
>
> If you start and stop the IG web container yourself, the default location of the IG configuration files is correct. By default, IG configuration files are located under `$HOME/.openig` on Linux, Mac, and UNIX systems, and under `%appdata%\OpenIG` on Windows.

Configure Tomcat to use the same protocol as the application you are protecting with IG. If the protected application is on a remote system, configure Tomcat to use the same port as well. If your application listens on both an HTTP and an HTTPS port, then you must configure Tomcat to do so, too.

To configure Tomcat to use an HTTP port other than 8080, modify the defaults in `/path/to/tomcat/conf/server.xml`. Search for the default value of 8080 and replace it with the new port number.

2.1.2.1. Configuring Tomcat Cookie Domains

To use IG for multiple protected applications running on different hosts, set a cookie domain in Tomcat or `JwtSession`.

- To set a cookie domain for an HTTP session (the default if you're not using a JWT session), add a context element to `/path/to/conf/Catalina/server/root.xml`, as in the following example, and then restart Tomcat to read the configuration changes:

  ```
  <Context sessionCookieDomain=".example.com" />
  ```

- To set a cookie domain for a JWT session, set the `cookieDomain` parameter in `JwtSession`. For information, see JwtSession(5) in the *Configuration Reference*. When the domain is set, a JWT cookie can be accessed from different hosts in that domain. When the domain is not set, the JWT cookie can be accessed only from the host where the cookie was created.

2.1.2.2. Configuring Tomcat For HTTPS (Server-Side)

To get Tomcat up quickly on an SSL port, add an entry similar to the following in `/path/to/tomcat/conf/server.xml`:

```
<Connector
 port="8443"
 protocol="HTTP/1.1"
 SSLEnabled="true"
 maxThreads="150"
 scheme="https"
 secure="true"
 address="127.0.0.1"
 clientAuth="false"
 sslProtocol="TLS"
 keystoreFile="/path/to/tomcat/conf/keystore"
 keystorePass="password"
/>
```

Also create a keystore holding a self-signed certificate:

```
$ keytool \
 -genkey \
 -alias tomcat \
 -keyalg RSA \
 -keystore /path/to/tomcat/conf/keystore \
 -storepass password \
 -keypass password \
 -dname "CN=openig.example.com,O=Example Corp,C=FR"
```

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

Notice the keystore file location and the keystore password both match the configuration. By default, Tomcat looks for a certificate with alias `tomcat`.

Restart Tomcat to read the configuration changes.

Browsers generally do not trust self-signed certificates. To work with a certificate signed instead by a trusted CA, see the Tomcat documentation on configuring HTTPS.

2.1.2.3. Configuring Tomcat to Access MySQL Over JNDI

If IG accesses an SQL database, then you must configure Tomcat to access the database using Java Naming and Directory Interface (JNDI). To do so, you must add the driver .jar for the database, set up a JNDI data source, and set up a reference to that data source.

The following steps are for MySQL Connector/J:

1. Download the MySQL JDBC Driver Connector/J from http://dev.mysql.com/downloads/connector/j.

2. Copy the driver .jar to `/path/to/tomcat/lib/` so that it is on Tomcat's class path.

3. Add a JNDI data source for your MySQL server and database in `/path/to/tomcat/conf/context.xml`:

```
<Resource
 name="jdbc/forgerock"
 auth="Container"
 type="javax.sql.DataSource"
 maxActive="100"
 maxIdle="30"
 maxWait="10000"
 username="mysqladmin"
 password="password"
 driverClassName="com.mysql.jdbc.Driver"
 url="jdbc:mysql://localhost:3306/databasename"
/>
```

4. Add a resource reference to the data source in `/path/to/tomcat/conf/web.xml`:

```
<resource-ref>
    <description>MySQL Connection</description>
    <res-ref-name>jdbc/forgerock</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

5. Restart Tomcat to read the configuration changes.

2.1.3. Jetty For IG

This section describes essential Jetty configuration that you need in order to run IG.

Download and install a supported version of Jetty from https://www.eclipse.org/jetty/download.html.

Configure Jetty to use the same protocol as the application you are protecting with IG. If the protected application is on a remote system, configure Jetty to use the same port as well. If your application listens on both an HTTP and an HTTPS port, then you must configure Jetty to do so as well.

To configure Jetty to use an HTTP port other than 8080, modify the defaults in `/path/to/jetty/etc/jetty.xml`. Search for the default value of 8080 and replace it with the new port number.

2.1.3.1. Configuring Jetty Cookie Domains

To use IG for multiple protected applications running on different hosts, set a cookie domain in Jetty or `JwtSession`:

• To set a cookie domain in Jetty, add a session domain handler element that specifies the domain to `/path/to/jetty/etc/webdefault.xml`, as in the following example:

```
<context-param>
  <param-name>org.eclipse.jetty.servlet.SessionDomain</param-name>
  <param-value>.example.com</param-value>
</context-param>
```

Restart Jetty to read the configuration changes.

- To set a cookie domain for a JWT session, set the `cookieDomain` parameter in `JwtSession`. For information, see JwtSession(5) in the *Configuration Reference*. When the domain is set, a JWT cookie can be accessed from different hosts in that domain. When the domain is not set, the JWT cookie can be accessed only from the host where the cookie was created.

2.1.3.2. Configuring Jetty For HTTPS (Server-Side)

This section describes how to set up Jetty to run IG over HTTPS.

These instructions are for Jetty 9.4.7, and are not compatible with earlier versions of Jetty. For more information about Jetty and HTTPS, see http://www.eclipse.org/jetty/documentation/current/configuring-ssl.html#configuring-sslcontextfactory.

To Configure Jetty for HTTPS

1. Remove the built-in keystore:

   ```
   $ rm /path/to/jetty/modules/ssl/keystore
   ```

2. Generate a new key pair with self-signed certificate in the keystore:

   ```
   $ keytool \
   -genkey
    \
   -alias jetty
    \
   -keyalg RSA
    \
   -keystore /path/to/jetty/modules/ssl/keystore
    \
   -storepass password
    \
   -keypass password
    \
   -dname "CN=openig.example.com,O=Example Corp,C=FR"
   ```

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

3. From `/path/to/jetty/`, create the `start.d` folder and its scripts:

```
$ java -jar start.jar --create-startd
MKDIR : ${jetty.base}/start.d
INFO  : Base directory was modified
```

The file /path/to/jetty/start.d/start.ini is created.

4. Enable the SSL module, and copy the keystore to etc:

```
$ java -jar start.jar --add-to-start=ssl
INFO  : ssl initialized in ${jetty.base}/start.d/ssl.ini
COPY  : ${jetty.base}/modules/ssl/keystore to ${jetty.base}/etc/keystore
INFO  : Base directory was modified
```

The file /path/to/jetty/start.d/ssl.ini is created.

5. Update the SSL module:

    a. Find the obfuscated form of the keystore password:

    ```
    $ java \
     -cp /path/to/jetty/lib/jetty-util-*.jar \
     org.eclipse.jetty.util.security.Password \
     password
    password
    OBF:1v2j1uum1xtv1zej1zer1xtn1uvk1v1v
    MD5:5f4dcc3b5aa765d61d8327deb882cf99
    ```

    b. In /path/to/jetty/start.d/ssl.ini, uncomment the following lines and update the passwords with the OBF password returned in the previous step:

    ```
    ## Connector port to listen on
    jetty.ssl.port=8443

    ## Keystore file path (relative to $jetty.base)
    jetty.sslContext.keyStorePath=etc/keystore

    ## Keystore password
    jetty.sslContext.keyStorePassword=OBF:1v2j1uum1xtv1zej1zer1xtn1uvk1v1v

    ## KeyManager password
    jetty.sslContext.keyManagerPassword=OBF:1v2j1uum1xtv1zej1zer1xtn1uvk1v1v
    ```

6. Enable the HTTPS module:

```
$ java -jar start.jar --add-to-start=https
INFO  : https initialized in ${jetty.base}/start.d/https.ini
INFO  : Base directory was modified
```

The file /path/to/jetty/start.d/https.ini is created.

7. Restart Jetty:

```
$ java -jar start.jar
. . .
2017-09-26 16:13:22.783:INFO:oejs.Server:main: Started @3782ms
```

FORGEROCK

Jetty starts up.

8. Copy the IG .war file to `/path/to/jetty/webapps/IG-6.1.0.war`, and make sure that you can access Studio on https://openig.example.com:8443/openig/studio.

If you see warnings that the site is not secure, or that the self-signed certificate is not valid, respond to the warnings to access the site.

2.1.3.3. Configuring Jetty to Access MySQL Over JNDI

If IG accesses an SQL database, then you must configure Jetty to access the database over JNDI. To do so, you must add the driver .jar for the database, set up a JNDI data source, and set up a reference to that data source.

The following steps are for MySQL Connector/J:

1. Download the MySQL JDBC Driver Connector/J from http://dev.mysql.com/downloads/connector/j.

2. Copy the driver .jar to `/path/to/jetty/lib/jndi/` so that it is on Jetty's class path.

3. Add a JNDI data source for your MySQL server and database in `/path/to/jetty/etc/jetty.xml`:

```
<New id="jdbc/forgerock" class="org.eclipse.jetty.plus.jndi.Resource">
  <Arg></Arg>
  <Arg>jdbc/forgerock</Arg>
  <Arg>
    <New class="com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSource">
      <Set name="Url">jdbc:mysql://localhost:3306/databasename</Set>
      <Set name="User">mysqladmin</Set>
      <Set name="Password">password</Set>
    </New>
  </Arg>
</New>
```

4. Add a resource reference to the data source in `/path/to/jetty/etc/webdefault.xml`:

```
<resource-ref>
    <description>MySQL Connection</description>
    <res-ref-name>jdbc/forgerock</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

5. Restart Jetty to read the configuration changes.

2.1.4. JBoss EAP For IG

This section describes a basic JBoss configuration to run IG.

To Install IG in JBoss

1. Download and install a supported version of JBoss from https://developers.redhat.com/products/
eap/download/.

   For information about supported versions, see "*Before You Install*" in the *Release Notes*.

2. Delete the JBoss welcome-content handler in the JBoss configuration file `/path/to/jboss/standalone/configuration/standalone.xml`:

   ```
   <server name="default-server">
     <host name="default-host" alias="localhost">
       <location name="/" handler="welcome-content"/> <!-- Delete this line -->
   ```

3. Download `IG-6.1.0.war` from the ForgeRock BackStage download site and move it to the JBoss deployment directory:

   ```
   $ cp IG-6.1.0.war /path/to/jboss/standalone/deployments/IG-6.1.0.war
   ```

4. (Optional) Add IG configuration:

   • To configure IG in the filesystem, edit `/path/to/jboss/bin/standalone.conf`. For example, add the IG base to the file:

   ```
   export IG_INSTANCE_DIR="/path/to/openig"
   ```

   • To configure IG at startup, add the configuration to the startup command. For example, add the IG base to the startup command:

   ```
   $ /path/to/jboss/bin/standalone.sh -Dig.instance.dir=/path/to/openig
   ```

5. Start JBoss as a standalone server:

   ```
   $ /path/to/jboss/bin/standalone.sh
   ```

   JBoss deploys IG in the root context.

6. Make sure that IG is running:

   • Make sure that you can see Studio at http://localhost:8080/openig/studio

   • Make sure that you can see the IG welcome page at http://localhost:8080

2.1.4.1. Configuring JBoss Cookie Domains

To use IG to protect multiple applications running on different hosts, set a cookie domain in JBoss or `JwtSession`:

• To set a cookie domain in JBoss, see the Redhat documentation about *Cookie Domain*.

• To set a cookie domain for a JWT session, set the `cookieDomain` parameter in `JwtSession`. For information, see JwtSession(5) in the *Configuration Reference*. When the domain is set, a JWT cookie can be accessed from different hosts in that domain. When the domain is not set, the JWT cookie can be accessed only from the host where the cookie was created.

2.1.4.2. Configuring JBoss For HTTPS (Server-Side)

Configure JBoss for HTTPS, depending on the requirements of application you are protecting with IG.

If the protected application listens on both an HTTP and an HTTPS port, configure JBoss to listen on both ports.

2.2. Preparing the Network

Because IG uses reverse proxy architecture, you must configure the network so that that traffic from the browser to the protected application goes through IG.

Modify DNS or host file settings so that the host name of the protected application resolves to the IP address of IG on the system where the browser runs.

Restart the browser after making this change.

2.3. Installing IG

This section describes a basic installation in Jetty or Tomcat. For information about installing in JBoss, see see "JBoss EAP For IG".

Follow these steps to install IG:

1. Download `IG-6.1.0.war` from the ForgeRock BackStage download site .

2. Copy the .war file to the root context of the web application container:

   • For Tomcat, copy the file to `/path/to/tomcat/webapps/ROOT.war`.

   • For Jetty, copy the file to `/path/to/jetty/webapps/IG-6.1.0.war`. Jetty automatically deploys IG in the root context on startup.

   • For JBoss, see "JBoss EAP For IG".

   > **Important**
   >
   > If you use startup scripts to bootstrap the IG web container, the scripts can start the container process with a different user. To prevent errors, make sure that the location of the IG configuration is correct.

> Alternatively, adapt the startup scripts to specify the `IG_INSTANCE_DIR` env variable or `ig.instance.dir` system properties, taking care to set file permissions correctly.
>
> If you start and stop the IG web container yourself, the default location of the IG configuration files is correct. By default, IG configuration files are located under `$HOME/.openig` on Linux, Mac, and UNIX systems, and under `%appdata%\OpenIG` on Windows.

3. Prepare your IG configuration files.

   For information about configuration files, see "Configuration Directories and Files". For information about how to change the default location of the configuration files, see "Changing the Default Location of the Configuration Folders".

   If you have not yet prepared configuration files, then start with the configuration described in "Trying IG With a Simple Configuration" in the *Getting Started Guide*.

   Copy the template to `$HOME/.openig/config/config.json`. Replace the baseURI of the DispatchHandler with that of the protected application.

   On Windows, copy the template to `%appdata%\OpenIG\config\config.json`. To locate the `%appdata%` folder for your version of Windows, open Windows Explorer, type `%appdata%` as the file path, and press Enter. You must create the `%appdata%\OpenIG\config` folder, and then add the configuration file.

4. Start the web container where IG is deployed.

5. Browse to the protected application.

   IG should now proxy all traffic to the application.

6. Make sure the browser is going through IG.

   Verify this in one of the following ways:

   - Follow these steps:

     1. Stop the IG web container.

     2. Verify that you cannot browse to the protected application.

     3. Start the IG web container.

     4. Verify that you can now browse to the protected application again.

   - Check the logs to see that traffic is going through IG.

2.4. Changing the Default Location of the Configuration Folders

By default, the IG configuration files are in the directory `$HOME/.openig` (on Windows, `%appdata%\OpenIG`) Change the default location in the following ways:

- Set the `IG_INSTANCE_DIR` environment variable to the full path to the base location for IG files:

```
# On Linux, macOS, and UNIX using Bash
$ export IG_INSTANCE_DIR=/path/to/openig

# On Windows
C:>set IG_INSTANCE_DIR=c:\path\to\openig
```

- When you start the web application container where IG runs, set the `ig.instance.dir` Java system property to the full path to the base location for IG files.

  The following example starts Jetty server in the foreground and sets the value of `ig.instance.dir`:

```
$ java -Dig.instance.dir=/path/to/openig -jar start.jar
```

2.5. Preparing For Load Balancing and Failover

For a high scale or highly available deployment, you can prepare a pool of IG servers with nearly identical configurations, and then load balance requests across the pool, routing around any servers that become unavailable. Load balancing allows the service to handle more load.

Before you spread requests across multiple servers, however, you must determine what to do with state information that IG saves in the context, or retrieves locally from the IG server system. If information is retrieved locally, then consider setting up failover. If one server becomes unavailable, another server in the pool can take its place. The benefit of failover is that a server failure can be invisible to client applications.

IG can save state information in several ways:

- Handlers including a `SamlFederationHandler` or a custom `ScriptableHandler` can store information in the context. Most handlers depend on information in the context, some of which is first stored by IG.

- Some filters, such as `AssignmentFilters`, `HeaderFilters`, `OAuth2ClientFilters`, `OAuth2ResourceServerFilters`, `ScriptableFilters`, `SqlAttributesFilters`, and `StaticRequestFilters`, 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.

IG can also retrieve information locally in several ways:

- Some filters and handlers, such as `FileAttributesFilters`, `ScriptableFilters`, `ScriptableHandlers`, and `SqlAttributesFilters`, can depend on local system files or container configuration.

By default, the context data resides in memory in the container where IG runs. This includes the default session implementation, which is backed by the HttpSession that the container handles. You can opt to store session data on the user-agent instead, however. For details and to consider whether your data fits, see JwtSession(5) in the *Configuration Reference*.

When you use the `JwtSession` implementation with a `cookieDomain` set, be sure to share the encryption keys and the `sharedSecret` across all IG configurations so that any server can read or update JWT cookies from any other server in the same `cookieDomain`.

If your data does not fit in an HTTP cookie, for example, because when encrypted it is larger than 4 KB, consider storing a reference in the cookie, and then retrieve the data by using another filter. IG logs warning messages if the `JwtSession` cookie is too large. Using a reference can also work when a server becomes unavailable, and the load balancer must fail requests over to another server in the pool.

If some data attached to a context must be stored on the server side, then you have additional configuration steps to perform for session stickiness and for session replication. Session stickiness means that the load balancer sends all requests from the same client session to the same server. Session stickiness helps to ensure that a client request goes to the server holding the original session data. Session replication involves writing session data either to other servers or to a data store, so that if one server goes down, other servers can read the session data and continue processing. Session replication helps when one server fails, allowing another server to take its place without having to start the session over again. If you set up session stickiness but not session replication, when a server crashes, the client session information for that server is lost, and the client must start again with a new session.

How you configure session stickiness and session replication depends on your load balancer and on your container.

Tomcat can help with session stickiness, and a Tomcat cluster can handle session replication:

- If you choose to use the Tomcat connector (mod_jk) on your web server to perform load balancing, then see the *LoadBalancer HowTo* for details.

  In the HowTo, you configure the `jvmRoute` attribute in the Tomcat server configuration, `/path/to/tomcat/conf/server.xml`, to identify the server. The connector can use this identifier to achieve session stickiness.

- A Tomcat cluster configuration can handle session replication. When setting up a cluster configuration, the ClusterManager defines the session replication implementation.

Jetty has provisions for session stickiness, and also for session replication through clustering:

- Jetty's persistent session mechanism appends a node ID to the session ID in the same way Tomcat appends the `jvmRoute` value to the session cookie. This can be useful for session stickiness if your load balancer examines the session ID.

- Session Clustering with a Database describes how to configure Jetty to persist sessions over JDBC, allowing session replication.

Unless it is set up to be highly available, the database can be a single point of failure in this case.

- Session Clustering with MongoDB describes how to configure Jetty to persist sessions in MongoDB, allowing session replication.

  The Jetty documentation recommends this implementation when session data is seldom written but often read.

2.6. Configuring IG For HTTPS (Client-Side)

For IG to connect to a server securely over HTTPS, IG must be able to trust the server. The default settings rely on the Java environment truststore to trust server certificates. The Java environment default truststore includes public key signing certificates from many well-known Certificate Authorities (CAs). If all servers present certificates signed by these CAs, then you have nothing to configure.

If, however, the server certificates are self-signed or signed by a CA whose certificate is not trusted out of the box, then you can configure the following objects:

- KeyStore, to hold the servers' certificates or the CA's signing certificate. For information, see KeyStore(5) in the *Configuration Reference*.

- TrustManager, to allow IG to handle the certificates in the KeyStore when deciding whether to trust a server certificate. For information, see TrustManager(5) in the *Configuration Reference*.

- KeyManager, an optional object to allow IG to present its certificate from the keystore when the server must authenticate IG as client. This object is optional. For information, see KeyManager(5) in the *Configuration Reference*.

- ReverseProxyHandler, to reference a configured TrustManager and KeyManager. For information, see ReverseProxyHandler(5) in the *Configuration Reference*.

You can configure each of these objects globally for the IG server, or locally for a particular ReverseProxyHandler configuration.

The Java KeyStore holds the peer servers' public key certificates (and optionally, the IG certificate and private key). For example, suppose you have a certificate file, `ca.crt`, that holds the trusted signer's certificate of the CA who signed the server certificates of the servers in your deployment. In that case, you could import the certificate into a Java Keystore file, `/path/to/keystore.jks`:

```
$ keytool \
 -import \
 -trustcacerts \
 -keystore /path/to/keystore \
 -file ca.crt \
 -alias ca-cert \
 -storepass changeit
```

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

You could then configure the following KeyStore for IG that holds the trusted certificate. Notice that the url field takes an expression that evaluates to a URL, starting with a scheme such as `file://`:

```
{
    "name": "MyKeyStore",
    "type": "KeyStore",
    "config": {
        "url": "file:///path/to/keystore",
        "password": "changeit"
    }
}
```

The TrustManager handles the certificates in the KeyStore when deciding whether to trust the server certificate. The TrustManager references your KeyStore:

```
{
    "name": "MyTrustManager",
    "type": "TrustManager",
    "config": {
        "keystore": "MyKeyStore"
    }
}
```

The `ReverseProxyHandler` configuration has the following security settings:

"trustManager"

References the `TrustManager`. Configure this when your server certificates are not trusted out of the box.

"hostnameVerifier"

Defines how the `ReverseProxyHandler` verifies host names in server certificates. By default, this setting is `STRICT`.

"keyManager"

References the optional `KeyManager`. Configure this if servers request that IG present its certificate as part of mutual authentication.

In that case, generate a key pair for IG, and have the certificate signed by a well-known CA. For instructions, see the documentation for the Java **keytool** command. You can use a different keystore for the `KeyManager` than you use for the `TrustManager`.

The following `ReverseProxyHandler` configuration references `MyTrustManager` and sets strict host name verification:

```
{
    "name": "ReverseProxyHandler",
    "type": "ReverseProxyHandler",
    "config": {
        "hostnameVerifier": "STRICT",
        "trustManager": "MyTrustManager"
    }
}
```

2.7. Setting Up Keys For JWT Encryption

You can use a JSON Web Token (JWT) session, JwtSession, to configure IG as described in JwtSession(5) in the *Configuration Reference*. A JwtSession stores session information in JWT cookies on the user-agent, rather than storing the information in the container where IG runs.

In order to encrypt the JWTs, IG needs cryptographic keys. IG can generate its own key pair in memory, but that key pair disappears on restart and cannot be shared across IG servers. Alternatively, IG can use keys from a keystore.

The following procedure prepares a keystore for JWT encryption in a deployment with one IG instance. For a deployment with more than one IG instance, also configure a `"sharedSecret"` as described in JwtSession(5) in the *Configuration Reference*.

To Prepare a Keystore for JWT Encryption

1. Generate the key pair in a new keystore file by using the Java **keytool** command.

   The following command generates a Java Keystore format file, `/path/to/keystore.jks`, holding a key pair with alias `jwe-key`. Notice that both the keystore and the private key have the same password:

   ```
   $ keytool \
    -genkey \
    -alias jwe-key \
    -keyalg rsa \
    -keystore /path/to/keystore.jks \
    -storepass changeit \
    -keypass changeit \
    -dname "CN=openig.example.com,O=Example Corp"
   ```

   Note

   Because KeyStore converts all characters in its key aliases to lower case, use only lowercase in alias definitions of a KeyStore.

2. Add a KeyStore to your configuration that references the keystore file:

```
{
    "name": "MyKeyStore",
    "type": "KeyStore",
    "config": {
        "url": "file:///path/to/keystore.jks",
        "password": "changeit"
    }
}
```

For details, see KeyStore(5) in the *Configuration Reference*.

3. Add a JwtSession to your configuration that references your KeyStore:

```
{
    "name": "MyJwtSession",
    "type": "JwtSession",
    "config": {
        "keystore": "MyKeyStore",
        "alias": "jwe-key",
        "password": "changeit",
        "cookieName": "IG",
        "cookieDomain": ".example.com"
    }
}
```

For a deployment with more than one IG instance, also configure a `"sharedSecret"` as described in JwtSession(5) in the *Configuration Reference*.

4. Specify your JwtSession object in the top-level configuration, or in the route configuration:

```
"session": "MyJwtSession"
```

2.8. Making the Configuration Immutable

IG operates in the following modes:

• **Development mode (mutable mode)**

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

In development mode, by default all endpoints are open and accessible. You can create, edit, and deploy routes through IG Studio, and manage routes through Common REST, without authentication or authorization.

To protect specific endpoints in development mode, configure an ApiProtectionFilter in `admin.json` and add it to the IG configuration.

• **Production mode (immutable mode)**

After you have developed your configuration, switch to production mode to test the configuration, to run the software in pre-production or production, or to run multiple instances of the software with the same configuration.

In production mode, the `/routes` endpoint is not exposed or accessible. Studio is effectively disabled, and you cannot manage, list, or even read routes through Common REST.

By default, other endpoints, such as `/monitoring`, `/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.

After installation, IG is by default in production mode. While you evaluate IG or develop routes, it can be helpful to switch to development mode as described in "Switching Between Production Mode and Development Mode" in the *Getting Started Guide*. This section describes options for switching back to production mode to run IG in a production environment.

To Make the Configuration Immutable

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, see AdminHttpApplication(5) in the *Configuration Reference*.

    The value set in `admin.json` overrides any value set by the configuration token `ig.run.mode` when it is used in an environment variable or system property. For information about `ig.run.mode`, see Configuration Tokens(5) in the *Configuration Reference*.

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 "Trying IG With a Simple Configuration" in the *Getting Started Guide*, and configure the `scanInterval` 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, see Router(5) in the *Configuration Reference*.

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://openig.example.com:8080/openig/studio.

Chapter 3
Getting Login Credentials From Data Sources

In "*First Steps*" in the *Getting Started Guide* you learned how to configure IG to proxy traffic and capture request and response data. You also learned how to configure IG to use a static request to log in with hard-coded credentials. In this chapter, you will learn to:

- Configure IG to look up credentials in a file

- Configure IG to look up credentials in a relational database

3.1. Before You Start

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*.

3.2. Log in With Credentials From a File

This sample shows you how to configure IG to get credentials from a file.

The sample uses a comma-separated value file, `userfile`:

```
username,password,fullname,email
george,costanza,George Costanza,george@example.com
kramer,newman,Kramer,kramer@example.com
bjensen,hifalutin,Babs Jensen,bjensen@example.com
demo,changeit,Demo User,demo@example.com
kvaughan,bribery,Kirsten Vaughan,kvaughan@example.com
scarter,sprain,Sam Carter,scarter@example.com
```

IG looks up the user credentials based on the user's email address. IG uses a `FileAttributesFilter` to look up the credentials.

Follow these steps to set up log in with credentials from a file:

1. Add the user file on your system:

```
$ vi /tmp/userfile
$ cat /tmp/userfile
username,password,fullname,email
george,costanza,George Costanza,george@example.com
kramer,newman,Kramer,kramer@example.com
bjensen,hifalutin,Babs Jensen,bjensen@example.com
demo,changeit,Demo User,demo@example.com
kvaughan,bribery,Kirsten Vaughan,kvaughan@example.com
scarter,sprain,Sam Carter,scarter@example.com
```

On Windows systems, use an appropriate path such as `C:\Temp\userfile`.

2. Add a new route to the IG configuration to obtain the credentials from the file.

   To add the route, add the following route configuration file as `$HOME/.openig/config/routes/02-file.json`:

```
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${true}",
            "credentials": {
              "type": "FileAttributesFilter",
              "config": {
                "file": "/tmp/userfile",
                "key": "email",
                "value": "george@example.com",
                "target": "${attributes.credentials}"
              }
            },
            "request": {
              "method": "POST",
              "uri": "http://app.example.com:8081/login",
              "form": {
                "username": [
                  "${attributes.credentials.username}"
                ],
                "password": [
                  "${attributes.credentials.password}"
                ]
              }
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
```

```
    "condition": "${matches(request.uri.path, '^/file')}"
}
```

On Windows, the file name should be `%appdata%\OpenIG\config\routes\02-file.json`.

Notice the following features of the new route:

- The `FileAttributesFilter` specifies the file to access, the key and value to look up to retrieve the user's record, and where to store the results in the request context attributes map.

- The `PasswordReplayFilter` creates a request by retrieving the username and password from the attributes map and replacing your browser's original HTTP GET request with an HTTP POST login request that contains the credentials to authenticate.

- The route matches requests to `/file`.

3. On Windows systems, edit the path name to the user file.

Now browse to http://openig.example.com:8080/file.

If everything is configured correctly, IG logs you in as George.

What's happening behind the scenes?

Log in With Credentials From a File

Log in With Credentials From a File
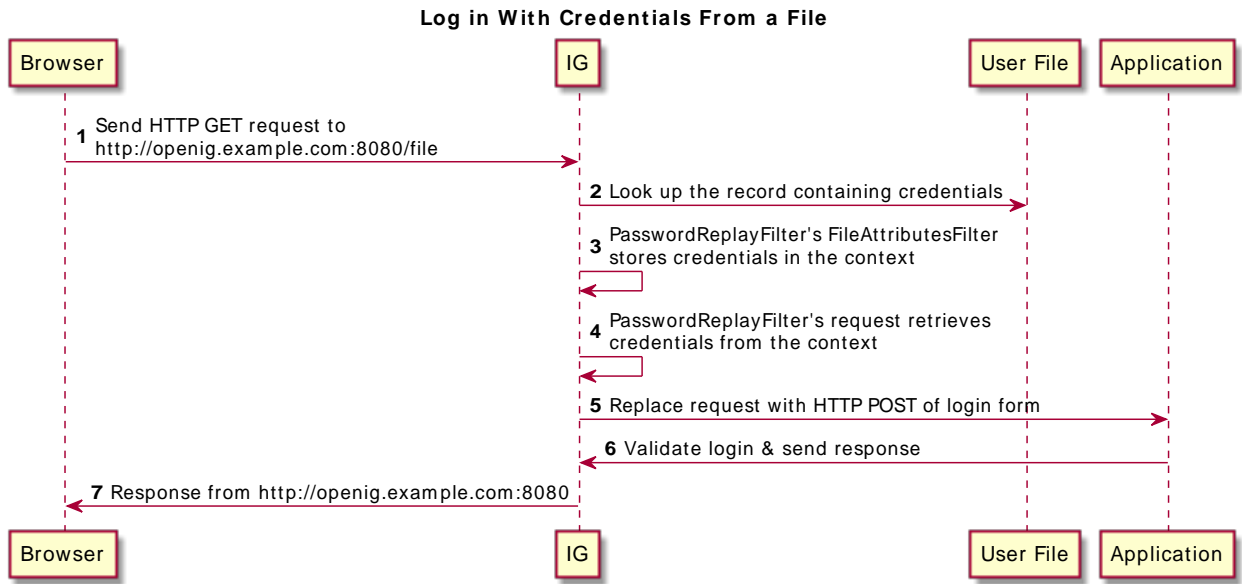
- IG intercepts the browser's HTTP GET request, which matches the route condition.

- The `PasswordReplayFilter`'s `FileAttributesFilter` looks up credentials in a file, and stores the credentials it finds in the request context attributes map.

- The `PasswordReplayFilter`'s request pulls the credentials out of the attributes map, builds the login form, and performs the HTTP POST request to the sample application.

- The sample application validates the credentials, and responds with a profile page. IG then passes the response from the sample application to your browser.

3.3. Log in With Credentials From a Database

This section describes how to configure IG to get credentials from H2. IG also works with other database software, but this example is tested with Jetty and H2 1.4.196.

IG relies on the application server where it runs to connect to the database. Configuring IG to retrieve data from a database is therefore a question of configuring the application server to connect to the database, and configuring IG to choose the appropriate data source, and to send the appropriate SQL request to the database. As a result, the IG configuration depends more on the data structure than on any particular database drivers or connection configuration.

The following flow chart illustrates the steps:

Log in With Credentials From a Database

Log in With Credentials From a Database
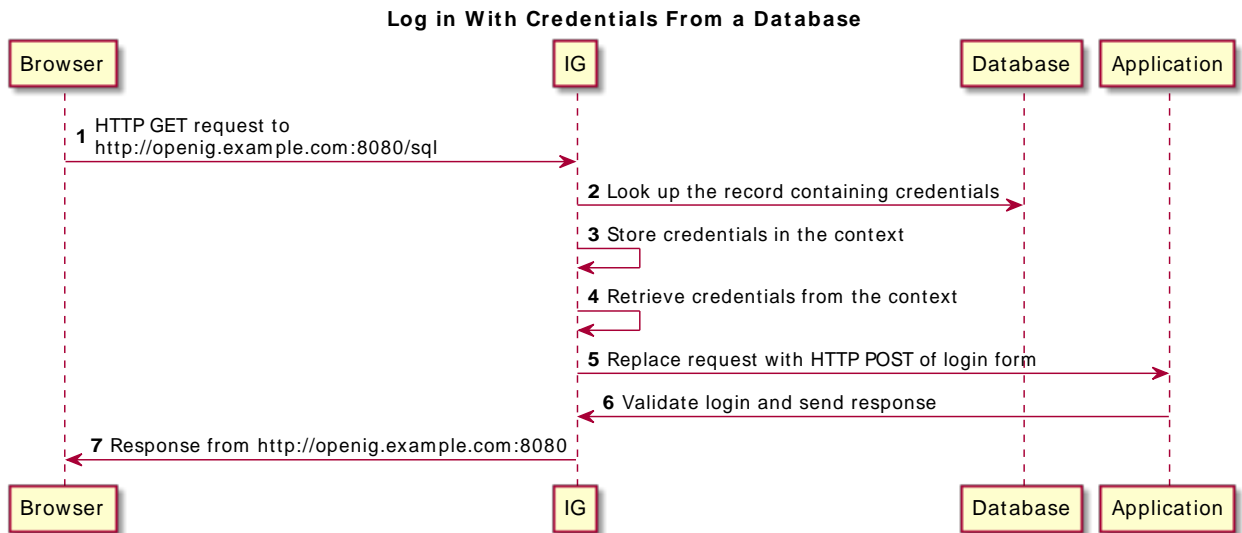
- IG intercepts the browser's HTTP GET request, which matches the route condition.

- The `SqlAttributesFilter` in `PasswordReplayFilter` looks up credentials in H2, and stores them in the request context attributes map.

- The `request` in `PasswordReplayFilter` pulls the credentials out of the attributes map, builds the login form, and performs the HTTP POST request to the sample application.

- The sample application validates the credentials, and responds with a profile page. IG then passes the response to the browser.

To Prepare the Database

1. Create the following user data in `/tmp/userfile`:

   ```
   username,password,fullname,email
   george,costanza,George Costanza,george@example.com
   kramer,newman,Kramer,kramer@example.com
   bjensen,hifalutin,Babs Jensen,bjensen@example.com
   demo,changeit,Demo User,demo@example.com
   kvaughan,bribery,Kirsten Vaughan,kvaughan@example.com
   scarter,sprain,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. Select the following options in the H2 Console, and then select Connect to access the console:

   - Saved Settings: `Generic H2 (Server)`

     This option sets the Driver Class, `org.h2.Driver`, the JDBC URL, `jdbc:h2:tcp://localhost/~/test`, and the User Name, `sa`.

   - Password: `password`

4. In the console, create the user table, using `/tmp/userfile`:

   ```
   DROP TABLE IF EXISTS USERS;
   CREATE TABLE USERS AS SELECT * FROM CSVREAD('/tmp/userfile');
   ```

5. Run the following command in the console to check that the table contains the same users as the file:

   ```
   SELECT * FROM users;
   ```

To Prepare Jetty's Connection to the Database

Follow these steps to enable Jetty to connect to the database:

1. Configure Jetty for JNDI.

Stop Jetty and add the following lines to `start.ini` in `/path/to/jetty/start.ini` to configure Jetty for JNDI:

```
# ========================================================
# Enable JNDI
# --------------------------------------------------------
OPTIONS=jndi

# ========================================================
# Enable additional webapp environment configurators
# --------------------------------------------------------
OPTIONS=plus
etc/jetty-plus.xml
```

For more information, see the Jetty documentation on *Configuring JNDI*.

2. Copy the H2 library to the classpath for Jetty:

```
$ cp /path/to/h2/bin/h2-*.jar /path/to/jetty/lib/ext/
```

3. Define a JNDI resource for H2 in `/path/to/jetty/etc/jetty.xml`:

```
<New id="jdbc/forgerock" class="org.eclipse.jetty.plus.jndi.Resource">
  <Arg></Arg>
  <Arg>jdbc/forgerock</Arg>
  <Arg>
    <New class="org.h2.jdbcx.JdbcDataSource">
      <Set name="Url">jdbc:h2:tcp://localhost/~/test</Set>
      <Set name="User">sa</Set>
      <Set name="Password">password</Set>
    </New>
  </Arg>
</New>
```

4. Add a resource reference to the data source in `/path/to/jetty/etc/webdefault.xml`:

```
<resource-ref>
    <res-ref-name>jdbc/forgerock</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

5. Restart Jetty to take the configuration changes into account.

To Prepare the IG Configuration

Add a new route to the IG configuration to look up credentials in the database:

1. Add the following route configuration file as `$HOME/.openig/config/routes/03-sql.json`:

```
{
  "handler": {
    "type": "Chain",
```

```
  "config": {
    "filters": [
      {
        "type": "PasswordReplayFilter",
        "config": {
          "loginPage": "${true}",
          "credentials": {
            "type": "SqlAttributesFilter",
            "config": {
              "dataSource": "java:comp/env/jdbc/forgerock",
              "preparedStatement":
              "SELECT username, password FROM users WHERE email = ?;",
              "parameters": [
                "george@example.com"
              ],
              "target": "${attributes.sql}"
            }
          },
          "request": {
            "method": "POST",
            "uri": "http://app.example.com:8081/login",
            "form": {
              "username": [
                "${attributes.sql.USERNAME}"
              ],
              "password": [
                "${attributes.sql.PASSWORD}"
              ]
            }
          }
        }
      }
    ],
    "handler": "ReverseProxyHandler"
  }
},
"condition": "${matches(request.uri.path, '^/sql')}"
}
```

On Windows, add the file as `%appdata%\OpenIG\config\routes\03-sql.json`.

2. Notice the following features of the route:

- The route matches requests to `/sql`.

- The `SqlAttributesFilter` in `PasswordReplayFilter` 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 in `PasswordReplayFilter` retrieves the username and password from the attributes map and replaces the browser's original HTTP GET request with an HTTP POST login request that contains the credentials to authenticate.

  The request is for `username, password`, but H2 returns the fields as `USERNAME` and `PASSWORD`. The configuration reflects this difference.

- The sample application validates the credentials, and responds with a profile page, which IG then passes to your browser.

To Test the Setup

Before you start, set up H2, Jetty, and IG as described in the previous procedures, make sure that sample application is running, and log out of AM.

- Access the route on http://openig.example.com:8080/sql.

  IG logs you in to the sample application as George.

Chapter 4

Getting Login Credentials From AM

Use IG with AM's password capture and replay feature 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, such as SharePoint, using the same user identity.

AM's password capture and replay feature captures the password during authentication, encrypts it, and adds to the session. IG's CapturedUserPasswordFilter then decrypts the password and uses it for authentication to a legacy web application.

For an alternative configuration using an AM policy agent instead of IG's CapturedUserPasswordFilter, see the documentation for earlier versions of IG.

4.1. About Password Capture and Replay

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.

Data Flow to Log In To a Protect Application With AM Credentials
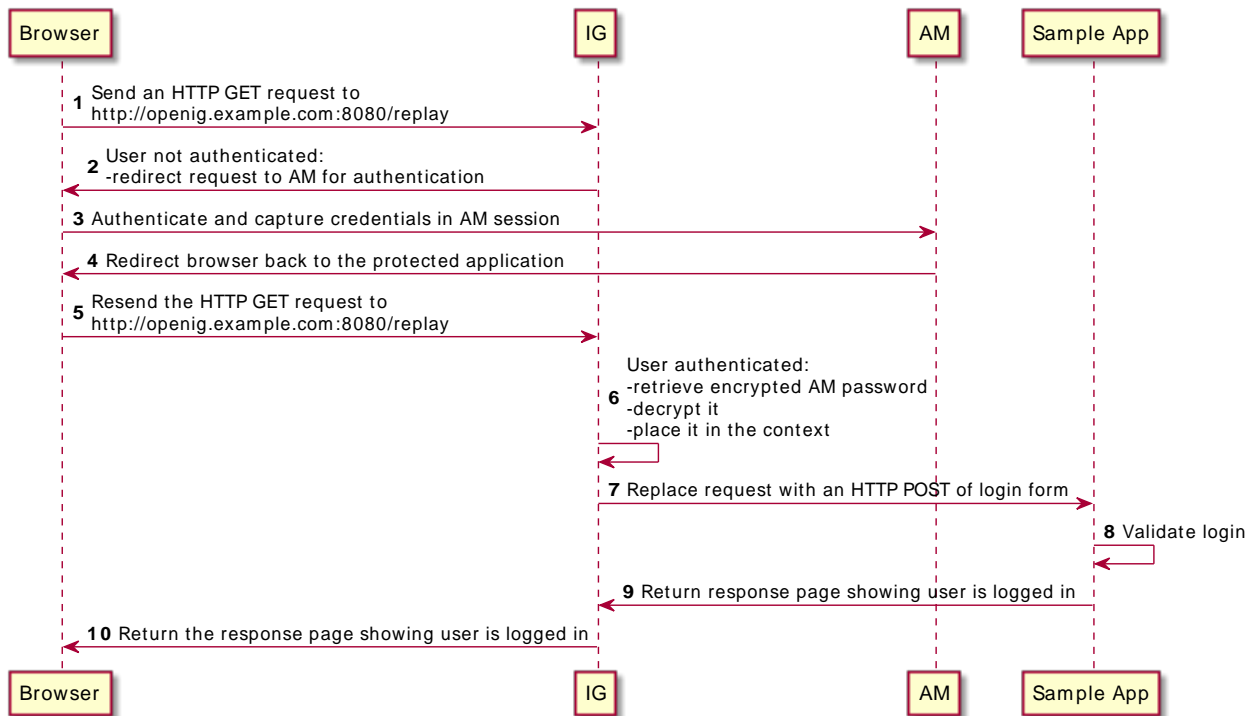
- IG intercepts the browser's HTTP GET request.

- 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.

4.2. Setting Up the Example

Before you start:

- Prepare IG and the sample app as described in "*First Steps*" in the *Getting Started Guide*

- Install and configure AM on http://openam.example.com:8088/openam, using the default configuration. If you use a different configuration, substitute in the tutorial accordingly.

The following table summarizes the steps to set up this example.

Tasks for Configuring This Example

Task	See Section(s)
Create a key to decrypt the password shared by AM and IG. This example uses DES as the decryption algorithm. From AM 6, CapturedUserPasswordFilter can use the stronger algorithm AES to decrypt the AM password.	"To Create a DES Shared Key In IG"
Configure AM password capture, and add the DES key to the AM configuration.	"To Configure Password Capture in AM"
Create a route in IG to handle the requests.	"To Configure Password Replay in IG"
Test the setup.	"To Test the Setup"

To Create a DES Shared Key In IG

Before AM communicates passwords to IG, it encrypts them with a key. IG then uses the key to decrypt the shared passwords.

This example uses DES as the decryption algorithm. For more information about DES shared keys, see DesKeyGenHandler(5) in the *Configuration Reference*.

The shared key is sensitive information. If it is possible for others to inspect the response, make sure you use HTTPS to protect the communication.

Use this procedure to generate a DES shared key in IG that you can use later in the AM and IG configuration:

1. Add the following route to the IG configuration as `$HOME/.openig/config/routes/04-keygen.json`. On Windows, add the route as `%appdata%\OpenIG\config\routes\04-keygen.json`.

```
{
  "name": "04-keygen",
  "handler": {
    "type": "DesKeyGenHandler"
  },
  "condition": "${matches(request.uri.path, '^/keygen') and (matches(contexts.client.remoteAddress,
':1') or matches(contexts.client.remoteAddress, '127.0.0.1'))}"
}
```

2. Call the route to generate a key:

```
$ curl http://localhost:8080/keygen
{"key":"1A+BCdEfGhI="}
```

In this example, the key is `1A+BCdEfGhI=`.

3. Make a note of the key for later.

To Configure Password Capture in AM

1. Update the Authentication Post Processing Classes for password replay.

   a. In the AM console, select Authentication > Settings > Post Authentication Processing.

   b. Add the following class to Authentication Post Processing Classes: `com.sun.identity.authentication.spi.ReplayPasswd`.

2. Add the key you created in "To Create a DES Shared Key In IG" to the AM configuration:

   a. In the AM console, select DEPLOYMENT > Servers, and then select the AM server name.

   In some earlier releases, select Configuration > Servers and Sites.

   b. Select Advanced, and add the property `com.sun.am.replaypasswd.key` with the value of the DES shared key.

3. Add `sunIdentityUserPassword` to the session whitelist, so that the property can be read from the session:

   a. Select Services > Session Property Whitelist Service.

   b. Add `sunIdentityUserPassword` as a whitelisted session property name.

4. Restart AM.

To Configure Password Replay in IG

1. Add the following route to the IG configuration as `$HOME/.openig/config/routes/04-replay.json`. On Windows, add the route as `%appdata%\OpenIG\config\routes\04-replay.json`.

```
{
  "heap": [{
    "name": "AmService-1",
    "type": "AmService",
    "config": {
      "url": "http://openam.example.com:8088/openam/",
      "version": "6.0"
    }
  },
    {
      "name": "CapturedUserPasswordFilter",
      "type": "CapturedUserPasswordFilter",
      "config": {
        "ssoToken": "${contexts.ssoToken.value}",
        "key": "DESKEY",
        "amService": "AmService-1"
      }
    }
  ],
  "name": "04-replay",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [{
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      },
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${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"
    }
  },
  "condition": "${matches(request.uri.path, '^/replay')}"
}
```

2. If necessary, change the value of `version` for AmService to your version of AM.

3. Change `DESKEY` to the key value that you generated in "To Create a DES Shared Key In IG".

   Notice the following features of the route:

   • The route matches requests to `/replay`.

   • The first filter in the chain is the SingleSignOnFilter. For information, see SingleSignOnFilter(5) in the *Configuration Reference*.

     If the request does not have a valid iPlanetDirectoryPro cookie, the SingleSignOnFilter redirects the request to AM for authentication.

     If the request already has a valid iPlanetDirectoryPro cookie, or after authenticating with AM to get a valid iPlanetDirectoryPro cookie, the SingleSignOnFilter passes the request to the next filter. The SingleSignOnFilter stores the cookie value in an `SsoTokenContext`.

     See SingleSignOnFilter(5) in the *Configuration Reference*.

   • The next filter in the chain is the PasswordReplayFilter.

     The PasswordReplayFilter uses the CapturedUserPasswordFilter declared in the heap to retrieve the AM password from session properties. The CapturedUserPasswordFilter uses the DES shared key to decrypt the password, and then makes it available in a CapturedUserPasswordContext.

     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.

     See PasswordReplayFilter(5) in the *Configuration Reference* and CapturedUserPasswordFilter(5) in the *Configuration Reference*.

To Test the Setup

1. Log out of AM if you are logged in.

2. Access the route on http://openig.example.com:8080/replay.

   You should be redirected to the AM login page.

3. Log in with username `demo`, password `changeit`.

   The request is redirected to the sample application.

> **Tip**
>
> If requests are directed to AM instead of to the sample application, review the cookie domain in the AM configuration. For more information, see "Requests Redirected to Access Management Instead of to the Resource".

Chapter 5
Single Sign-On and Cross-Domain Single Sign-On

This chapter describes the support for single-sign on (SSO) and cross-domain single sign-on (CDSSO) provided by IG.

5.1. About SSO Using the SingleSignOnFilter

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.

For an example where SSO with the SingleSignOnFilter is used, see "Setting Up IG as a PEP".

The following sequence diagram shows the flow of information during SSO between IG and AM as the authentication provider.

Flow of Information for SSO
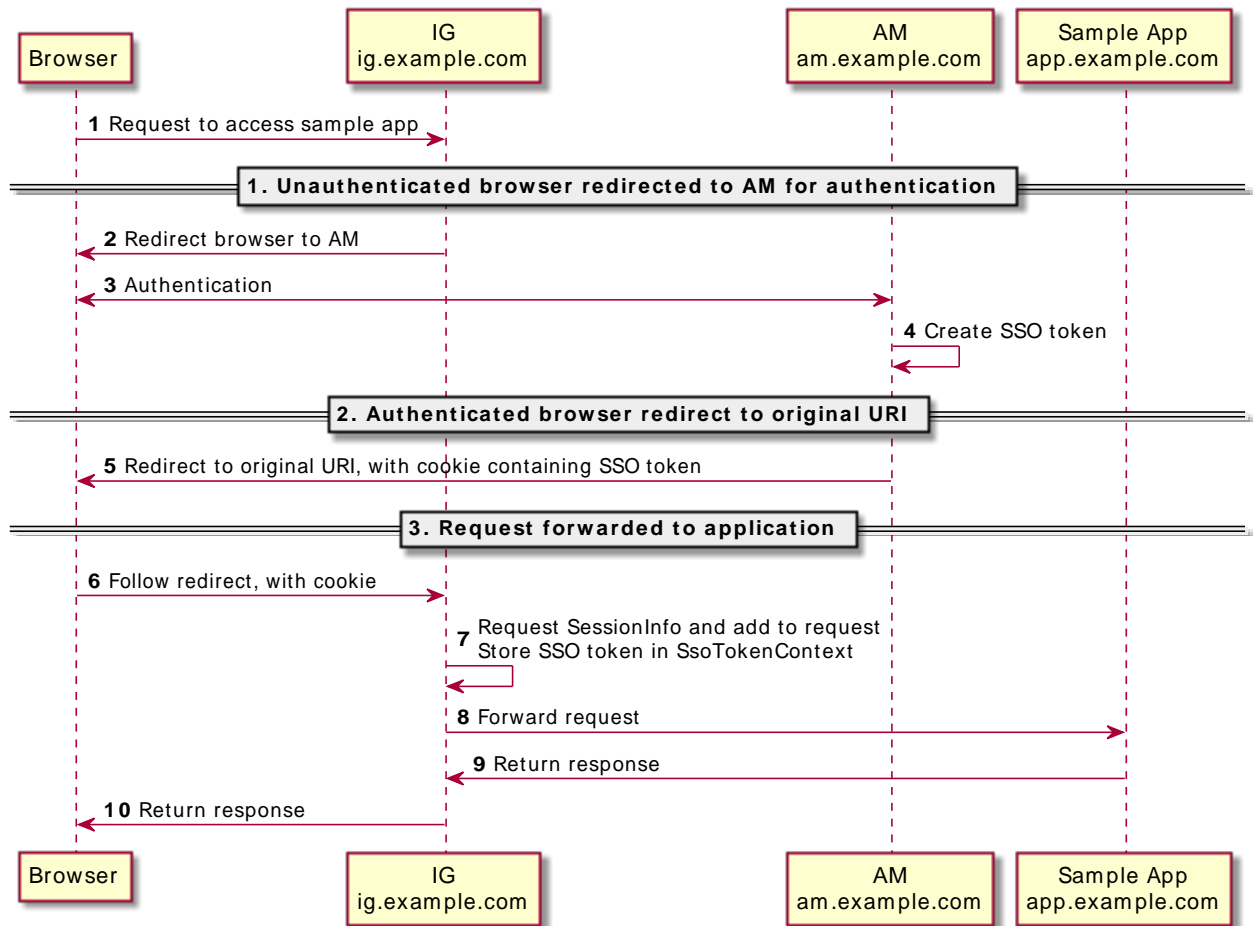
- The browser sends an unauthenticated request to access the sample app.

- 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 SessionInfo 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.

5.2. About CDSSO Using the CrossDomainSingleSignOnFilter

The SSO mechanism described in the previous section 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 app during CDSSO. In this example, AM is running on `am.example.com`, and IG is running on `ig.ext.com`.

Flow of Information for CDSSO
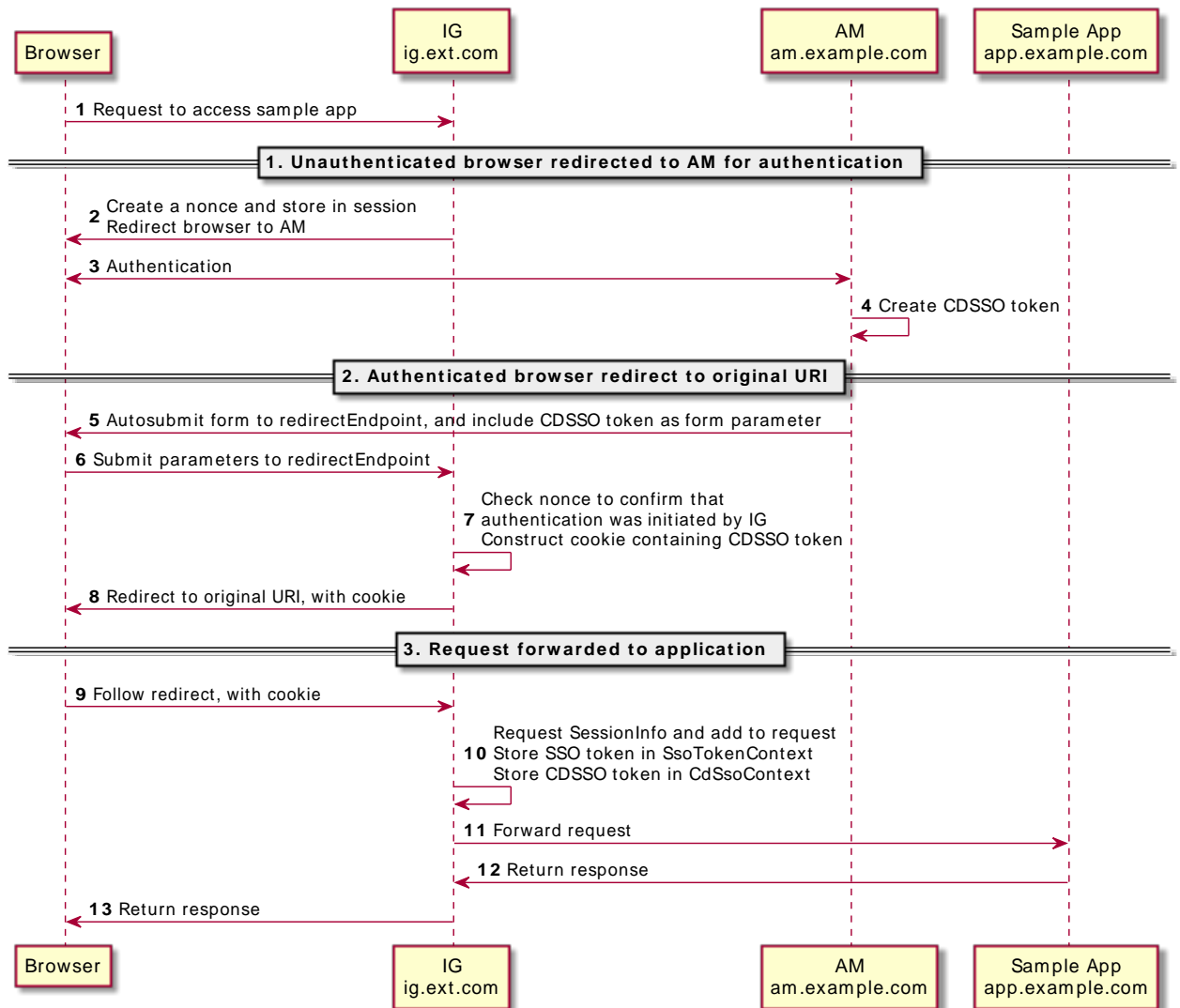
- The browser sends an unauthenticated request to access the sample app.

- IG intercepts the request, and redirects the browser to AM for authentication.

- AM authenticates the user and creates a CDSSO token.

- AM responds to a successful authentication with an HTML autosubmit form containing the issued token.

- The browser loads the HTML and autosubmit form parameters to the IG callback URL for the redirect endpoint.

- IG checks the nonce found inside the CDSSO token to confirm that the callback comes from an authentication initiated by IG. IG then 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 J2EE agent.

- IG redirects the request back to the original URI, with the cookie, and the browser follows the redirect back to IG.

- IG validates the token it gets from the cookie. It adds SessionInfo to the request, and stores the SSO token and CDSSO token in the contexts 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.

5.3. Setting Up CDSSO

This section describes how to set up CDSSO, where AM on `openam.example.com` authenticates users that are processed by IG on `openig.ext.com`.

CDSSO in AM must be configured as a Java EE policy agent. For more information, see Implementing Cross-Domain Single Sign-On in the Access Management *Authentication and Single Sign-On Guide*.

Before you start, install and configure AM on http://openam.example.com:8088/openam, with the default configuration. If you use a different configuration, substitute in the tutorial accordingly.

To Enable CDSSO in AM

1. In the top-level realm, select Applications > Agents > Java (or J2EE in earlier versions of AM).

2. Add a new agent with the following values:

   - Agent ID: `J2EE`

   - Agent URL: `http://openig.ext.com:8080/agentapp`

   - Server URL: `http://openam.example.com:8088/openam`

   - Password: `password`

3. Select the agent you just created, and on the SSO tab select set the following values:

   - Cross Domain SSO: Deselect this option

- CDSSO Redirect URI: `/home/cdsso/redirect`

To Set Up IG for CDSSO

Before you start, make sure that IG is running in development mode on `http://openig.ext.com:8080`, the sample application on `http://app.example.com:8081`, and AM on `http://openam.example.com:8088`.

1. Browse to http://openig.ext.com:8080/openig/studio, and select ⊕ Protect an Application.

2. Choose ☰ Structured to use the predefined menus and templates.

3. In the ✚ Create a route window, select Advanced Options, enter the following information, and then select Create route:

   - Base URI: `http://app.example.com:8081`

   - Condition: Path: `/home/cdsso`

   - Name: `cdsso`

4. Configure authentication:

   a. Select 👤 Authentication, and then enable it.

   b. Select Cross-Domain Single Sign-On, and enter the following information to reflect the configuration in "To Enable CDSSO in AM", and then save the settings:

      - AM service: Configure an AM service to use for authentication:

         - URI: `http://openam.example.com:8088/openam`

         - Version: The version of the AM instance, with at least one point. For example, `6.0`.

         - Agent: The credentials of the agent you created in AM.

            - Username: `J2EE`

            - Password: `password`

      - Redirect endpoint: `/home/cdsso/redirect`

      - Authentication cookie:

         - Path: `/home`

      Leave all other values as default.

5. On the top-right of the screen, select ⋮ and ⧉ Display to review the route.

   The following route should be displayed:

```
{
  "name": "cdsso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/cdsso')}",
  "monitor": false,
  "heap": [
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://openam.example.com:8088/openam",
        "realm": "/",
        "ssoTokenHeader": "iPlanetDirectoryPro",
        "agent": {
          "username": "J2EE",
          "password": "password"
        },
        "version": "6.0"
      }
    }
  ],
  "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"
    }
  }
}
```

Note

If necessary, change the value of `version` for AmService to your version of AM.

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.

To Test the Setup

1. Log out of AM if you are logged in, and clear any cookies.

2. Browse to http://openig.ext.com:8080/home/cdsso.

   The CrossDomainSingleSignOnFilter redirects the request to AM for authentication.

3. Log in to AM as user `demo`, password `changeit`.

   When you have authenticated, AM calls `/home/cdsso/redirect`, and includes the CDSSO token.

   The CrossDomainSingleSignOnFilter then passes the request to sample app, which returns the profile page.

> **Tip**
>
> If requests are directed to AM instead of to the sample application, review the cookie domain in the AM configuration. For more information, see "Requests Redirected to Access Management Instead of to the Resource".

Chapter 6

Enforcing Policy Decisions and Supporting Session Upgrade

This chapter describes how to set up IG as a policy enforcement point, with AM as a policy decision point. It provides an example of how to enforce a policy decision from AM, and an example of upgrading a session to a higher authentication level.

For more information about authentication and session upgrade, see AM's *Authentication and Single Sign-On Guide* .

6.1. About IG As a PEP With AM As PDP

The following terms are used in access management:

- *Policy Decision Point* (PDP): Entity that evaluates access rights and then issues authorization decisions.

- *Policy Enforcement Point* (PEP): Entity that intercepts a request for a resource and then enforces policy decisions from a PDP.

IG as a PEP intercepts requests for a resource, and provides information about the request to AM as a PDP.

AM 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.

Attributes and advices returned by the policy decision are stored in the `${contexts.policyDecision}` context.

After a policy decision, IG continues to process requests as follows:

- If the request is allowed, processing continues.

- If the request is denied with advice, IG checks whether it can respond to the advice. If IG can, it sends a redirect to AM, with information about how to meet the conditions in the advice.

- If the request is denied without advice, or if IG cannot respond to the advice, IG forwards the request to a `failureHandler` declared in the `PolicyEnforcementFilter`. If there is no `failureHandler`, IG returns a 403 Forbidden.

- If an error occurs during the process, IG returns 500 Internal Server Error.

In AM, administrators can maintain centralized, fine-grained, declarative policies to manage who can access what resources, and under what conditions. Policies can be managed separately by AM realm and by AM application.

AM provides a REST API for authorized users to request policy decisions. IG provides a PolicyEnforcementFilter that uses the REST API. For information, see PolicyEnforcementFilter(5) in the *Configuration Reference*.

6.2. Enforcing Policy Decisions From AM

This section gives an example of how to set up IG as a PEP, requesting policy decisions from AM as a PDP.

Tasks for Configuring Policy Enforcement

Task	See Section(s)
Set up a policy in AM to allow authenticated users to access the sample application.	"To Create a Policy in AM"
Create an AM agent. It will have all the necessary permissions to request policy decisions. When IG requests policy decisions, it must use these credentials.	"To Create a Java Agent in AM"
Configure IG as a PEP in Studio.	"To Set Up IG as a PEP"
Test the setup.	"To Test the Setup"

6.2.1. Setting Up AM As a PDP

This section describes how to create a policy in AM and configure an agent that can request policy decisions.

Before you start, install and configure AM on http://openam.example.com:8088/openam, with the default configuration. If you use a different configuration, substitute in the tutorial accordingly.

To Create a Policy in AM

1. Log in to the AM console as administrator.

2. In the top level realm, select Authorization > Policy Sets.

3. Add a new policy set with the following values, and then select Create:

   - Id: `PEP policy set`

- Resource Types: `URL`

4. In the new policy set, add a policy with the following values, and then select Create:

    - Name: `IG Policy`

    - Resource Type: `URL`

    - Resource pattern: `*://*:*/*`

    - Resource value: `http://app.example.com:8081/home/pep`

    This policy protects the home page of the sample application.

5. Select the Actions tab, add an action to allow HTTP `GET`, and then save your changes.

6. Select the Subjects tab, remove any default subject conditions, add a subject condition for all `Authenticated Users`, and then save your changes.

To Create a Java Agent in AM

1. In the top-level realm, select Applications > Agents > Java (or J2EE in earlier versions of AM).

2. Add a new agent with the following values:

    - Agent ID: `J2EE`

    - Agent URL: `http://openig.ext.com:8080/agentapp`

    - Server URL: `http://openam.example.com:8088/openam`

    - Password: `password`

3. Make sure all the changes are saved.

6.2.2. Setting Up IG as a PEP

This section describes how to set up IG to configure policy enforcement, where the user-agent is redirected to AM for authentication.

To configure IG without using Studio, add the route in the following procedure to the IG configuration as `$HOME/.openig/config/routes/04-pep.json`. On Windows, add the route as `%appdata%\OpenIG\config\routes\04-pep.json`.

To Set Up IG as a PEP

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*, and access Studio as described in "Accessing Studio" in the *Getting Started Guide*. IG must be running in development mode.

1. Browse to http://openig.example.com:8080/openig/studio, and select ⊕ Protect an Application.

2. Choose ☰ Structured to use the predefined menus and templates.

3. In the ✚ Create a route window, select Advanced Options, enter the following information, and then select Create route:

   - Base URI: `http://app.example.com:8081`

   - Condition: Path: `/home/pep`

   - Name: `04-pep`

4. Configure authentication:

   a. Select 👤 Authentication, and then enable it.

   b. Select Single Sign-On, and configure an AM server to use for authentication:

      - URI: `http://openam.example.com:8088/openam`

      - Version: The version of the AM server instance, with at least one point. For example, `6.0`.

      Leave the other fields with their default values.

5. Configure authorization:

   a. Select 🔍 Authorization, and then enable it.

   b. Select AM Policy Enforcement.

   c. Select the following options to reflect the configuration in "Setting Up AM As a PDP", and then save the settings:

      - AM configuration:

         - AM server: `http://openam.example.com:8088/openam (/)`.

           This is the AM server configured in the previous step. If you edit AM server, the change affects all filters that use it.

         - Policy administrator ID: `J2EE`

         - Policy administrator password: `password`

      - AM policy endpoint:

         - Policy set: `PEP policy set`

         - AM SSO token ID: `${contexts.ssoToken.value}`

      Leave the other fields with their default values, or empty if they are empty.

6. On the top-right of the screen, select ⋮ and ⬈ Display to review the route.

The following route should be displayed:

```
{
  "name": "04-pep",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/pep')}",
  "monitor": false,
  "heap": [
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://openam.example.com:8088/openam",
        "realm": "/",
        "ssoTokenHeader": "iPlanetDirectoryPro",
        "version": "6.0"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "PolicyEnforcementFilter-1",
          "type": "PolicyEnforcementFilter",
          "config": {
            "pepUsername": "J2EE",
            "pepPassword": "password",
            "pepRealm": "/",
            "application": "PEP policy set",
            "ssoTokenSubject": "${contexts.ssoToken.value}",
            "amService": "AmService-1"
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

> **Note**
>
> If necessary, change the value of `version` for AmService to your version of AM.

Notice the following features of the new route:

- The route matches requests to `/home/pep`.

- The first filter in the chain is the SingleSignOnFilter.

  If the request does not have a valid iPlanetDirectoryPro cookie, the SingleSignOnFilter redirects the request to authenticate with the AM server declared in the heap.

  If the request already has a valid iPlanetDirectoryPro cookie, or after authenticating with AM, the SingleSignOnFilter passes the request to the next filter, and stores the cookie value in an `SsoTokenContext`.

- The next filter in the chain is the PolicyEnforcementFilter.

  This filter retrieves the token value from the `SsoTokenContext` to identify the subject making the request. Using the AmService declared in the heap, the PolicyEnforcementFilter calls the AM policy endpoint for a policy decision.

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.

6.2.3. Testing the Setup

To Test the Setup

1. Log out of AM.

2. Browse to http://openig.example.com:8080/home/pep.

   Because you have not 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 `changeit`.

   When you have authenticated, AM redirects you back to the request URL, and IG requests a policy decision using the iPlanetDirectoryPro cookie value.

   AM returns a policy decision that grants access to the sample application.

> **Tip**
>
> If requests are directed to AM instead of to the sample application, review the cookie domain in the AM configuration. For more information, see "Requests Redirected to Access Management Instead of to the Resource".

6.3. Upgrading a Session

To protect sensitive resources, AM can configure authorization policies that require higher authentication levels. When a user-agent tries to access a sensitive resource at a lower authentication level, AM returns a policy decision that denies access, and returns an advice on how to upgrade the session to the higher level.

IG responds 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.

This section builds on the example in "Enforcing Policy Decisions From AM". The AM policy is edited to require the authentication level `1` to access the sample application. When the user tries to access the sample application, AM denies the request and provides an `AuthLevel` advice to increase the authentication level.

Tasks for Configuring Session Upgrade

Task	See Section(s)
Set up IG as a policy enforcement point and AM as a policy decision point, as described in the previous section.	"Enforcing Policy Decisions From AM"
Add an environment condition to the IG policy to require authentication level 1 to access the sample application.	"To Add an Environment Condition to the AM Policy"
Create a scripted authentication module and scripts in AM to deliver a token with authentication level 1.	"To Set Up an Authentication Module"
Try the setup.	"To Test the Setup"

6.3.1. Setting Up AM for Session Upgrade

This section provides an example of session upgrade. After authenticating with AM, the user must also provide a verification code to access the sample application.

To run this example, configure IG and AM as described in "Enforcing Policy Decisions From AM". No additional configuration of IG is required; the following procedures build on the AM configuration.

To Add an Environment Condition to the AM Policy

This section changes the policy you created in "To Create a Policy in AM".

1. Log in to the AM console as administrator.

2. In the top level realm, select Authorization > Policy Sets > PEP policy set.

3. In the IG policy, select Environments, and add an environment condition to require authentication level greater than or equal to 1.

To Set Up an Authentication Module

This section sets up an AM authentication module and scripts to upgrade a session's authentication level to 1.

1. In the top level realm, select Scripts > Scripted Module - Client Side, and replace the default script with one of the following scripts:

   - For AM 6 and later versions, use this script:

```
/*
 * Copyright 2018 ForgeRock AS. All Rights Reserved
 *
 * Use of this code requires a commercial software license with ForgeRock AS.
 * or with one of its affiliates. All use shall be exclusively subject
 * to such license between the licensee and ForgeRock AS.
 */
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 = "{}";
        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);
}
```

- For earlier versions of AM, use this script:

```
spinner.hideSpinner();
autoSubmitDelay = 60000;
$(document).ready(function() {
  fs = $(document.forms[0]).find("fieldset");
  strUI = '<div class="form-group"> \
          <label class="sr-only separator" for="answer"> \
          Verification Code</label><input onchange="s=$(\'#clientScriptOutputData\')[0]; \
          if (!s.value) s.value=\'{}\'; d=JSON.parse(s.value); d[\'answer\']=value; \
          s.value=JSON.stringify(d);" id="answer" class="form-control input-lg" type="text" \
          placeholder="Enter your verification code" value="" name="answer"></input></div>';
  $(fs).prepend(strUI);
});
```

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. In the top level realm, select Authentication > Modules, and add a module with the following characteristics:

   • Name: `VerificationCodeLevel1`

   • Type: `Scripted Module`

4. 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`

6.3.2. Testing the Setup

To Test the Setup

1. Log out of AM.

2. Browse to http://openig.example.com:8080/home/pep.

   If you have not previously authenticated to AM, the SingleSignOnFilter redirects the request to AM for authentication.

3. Log in to AM as user `demo`, password `changeit`.

   AM creates a session with an authentication level `0`, and IG requests a policy decision.

   The updated policy requires authentication level `1`, which is higher than the session's current authentication level. AM issues the following redirect with a `AuthLevelConditionAdvice` to authenticate at level `1`:

```
http://openam.example.com:8088/openam/?goto=http://openig.example.com:8080/home/pep&realm=
/&authIndexType=composite_advice&authIndexValue=<Advices><AttributeValuePair><Attribute
 name="AuthLevelConditionAdvice"/><Value>1</Value></AttributeValuePair></Advices>
```

4. In the session upgrade window, enter the verification code `123456`.

   AM returns a policy decision that grants access to the sample application.

> **Tip**
>
> If requests are directed to AM instead of to the sample application, review the cookie domain in the AM configuration. For more information, see "Requests Redirected to Access Management Instead of to the Resource".

Chapter 7
Acting As a SAML 2.0 Service Provider

The federation component of IG is a standards-based authentication service that validates users and logs them in to applications that IG protects. The federation component implements SAML 2.0. In this chapter, you will learn:

- How IG works as a SAML 2.0 service provider

- How to set up IG as a service provider for a single application

"*SAML 2.0 and Multiple Applications*" describes how to set up IG as a SAML 2.0 service provider for two applications, using AM as the identity provider.

7.1. About SAML 2.0 SSO and Federation

Federation allows organizations to share identities and services without giving away their identity information or the services they provide. Federation depends on standards to orchestrate interaction and exchange information between providers.

SAML 2.0 is a standard that describes the messages that providers exchange and the way that they exchange them. SAML 2.0 enables web single sign-on (SSO), for example, where the service managing the user's identity does not belong to the same organization and does not use the same software as the service that the user wants to access.

The following terms are used for federation and SAML:

- *Identity Provider* (IDP): The service that manages the user identity.

- *Service Provider* (SP): The service that a user wants to access.

- *Circle of trust* (CoT): An identity provider and service provider that participate in the federation.

When an identity provider and a service provider participate in federation, they agree on what security information to exchange, and mutually configure access to each other's services. The metadata that identity providers and service providers share is in an XML format defined by the SAML 2.0 standard.

7.1.1. Steps in SSO

SSO can be initiated by the SP (*SP initiated SSO*) or by the IDP (*IDP initiated SSO*).

Before SSO can be initiated by an IDP, the IDP must be configured with links that refer to the SP, and the user must be authenticated to the IDP. Instead of accessing app.example.com directly on the SP, the user accesses a link on the IDP that refers to the remote SP. The IDP provides SAML assertions for the user to the SP.

When SSO is initiated by the SP, the user attempts to access app.example.com directly on the SP. Because the user's federated identity is managed by the IDP, the SP sends a SAML authentication request to the IDP. After authenticating the user, the IDP provides SAML assertions for the user to the SP.

In both cases, the SAML assertion sent from the IDP to the SP attests which user is authenticated, when the authentication succeeded, how long the assertion is valid, and so on. The assertions can optionally contain additional and configurable attribute values, such as user meta-information or anything else provided by the IDP.

The SP uses the SAML assertions from the IDP to make authorization decisions, for example, to let an authenticated user complete a purchase that gets charged to the user's account at the identity provider.

SAML assertions can optionally be signed and encrypted.

7.1.2. IG As a SAML 2.0 SP

IG acts as a SAML 2.0 SP for SSO, providing users with an interface to applications that don't support SAML 2.0.

When SSO is initiated by the IDP, the IDP sends an unsolicited authentication statement to IG. When SSO is initiated by IG, IG calls the federation component to initiate SSO with the IDP. In both cases, the job of the federation component is to authenticate the user and to pass the required attributes to IG so that IG can log the user into protected applications.

7.2. Installation Overview

This tutorial assumes that you are familiar with SAML 2.0 federation and with the components involved, including AM. For information about AM, read the documentation for your version of AM.

This tutorial does not address PKI configuration for validation and encryption, although IG is capable of handling both, just as any AM Fedlet can handle both.

This tutorial takes you through the steps to set up AM as an IDP and IG as an SP to protect an application. To set up multiple SPs, read this chapter, work through the samples, and then consider the explanation in "*SAML 2.0 and Multiple Applications*".

Tasks for Configuring SAML 2.0 SSO and Federation

Task	See Section(s)
Prepare the network.	"Preparing the Network"

Task	See Section(s)
Configure AM As an IDP.	" Setting Up AM for This Tutorial "
	" Setting Up a Hosted Identity Provider "
	" Create a Fedlet Configuration "
Configuring IG as a SP.	" Retrieve the Fedlet Configuration Files "
	" Adding a Route for Credential Injection "
	" Adding a Route for SAML Federation "

Fedlet Configuration Files

File	Description
`fedlet.cot`	Circle of trust for IG and the identity provider.
`idp.xml`	Standard metadata (usually generated by the IDP).
`idp-extended.xml`	Metadata extensions (usually generated by the IDP).
`sp.xml`	Standard metadata for the IG SP (usually generated by the IDP).
`sp-extended.xml`	Metadata extensions for the IG SP (usually generated by the IDP).

For examples of the federation configuration files, see " Example Federation Configuration Files ". You can copy and edit these files to create new configurations.

7.3. Preparing the Network

Configure the network so that browser traffic to the application hosts is proxied through IG. The example in this chapter uses the host name `sp.example.com`.

Add `sp.example.com` to your `/etc/hosts` file:

```
127.0.0.1    localhost openam.example.com openig.example.com app.example.com sp.example.com
```

7.4. Configuring AM As an IDP

7.4.1. Setting Up AM for This Tutorial

To Set Up AM for This Tutorial

1. Install and configure AM on `http://openam.example.com:8088/openam`, with the default configuration. If you use a different configuration, make sure you substitute in the tutorial accordingly.

2. Click Add Identity and create a user with the following values:

   - ID: `george`

   - Password: `costanza`

3. In the User window, set the following values:

   - Email Address: `george`

   - Employee Number: `costanza`

   For simplicity, this tutorial uses `mail` to hold the username, and `employeenumber` to hold the password. Both attributes are in the standard user profile with the default AM configuration, and neither is needed for anything else in this tutorial. In a real deployment, you would use other attributes to represent real user profiles.

4. Test that you can log in to AM with this username and password.

7.4.2. Setting Up a Hosted Identity Provider

To Set Up a Hosted Identity Provider

1. In the top level realm select Configure SAMLv2 Provider > Create Hosted Identity Provider.

   A configuration page for the IDP is displayed.

2. In metadata > Name, change `http://openam.example.com:8088/openam` to `openam`.

   This makes it easier to refer to AM as the IDP.

3. In metadata > Signing Key, select `test`.

4. In Circle of Trust, select an existing circle of trust (CoT) or add one. In this example, the CoT is called `Circle of Trust`.

5. In Attribute Mapping, map the `mail` attribute to `mail`, and map the `employeenumber` attribute to `employeenumber`.

The SAML 2.0 attribute mapping indicates that IG (the SP) wants AM (the IDP) to get the value of these attributes from the user profile and send them to IG. IG can use the attribute values to log the user in to the application it protects.

6. Select Configure.

   A confirmation page is displayed. You can start to create a Fedlet from this page or go back to the top level realm, as described in the following procedure.

7.4.3. Create a Fedlet Configuration

A Fedlet is an example web application that acts as a lightweight SAML v2.0 SP. When you create a Fedlet, the federation configuration files are created in a directory similar to this: `$HOME/openam/myfedlets/`*`openig-fedlet`*`/Fedlet.zip`.

To Create a Fedlet Configuration

1. In the top level realm, browse to Create Fedlet Configuration.

2. In Name, enter a name for the Fedlet. In this tutorial, the Fedlet is named `sp`.

3. In Destination URL, enter the following URL for the SP: `http://sp.example.com:8080/saml`.

4. In Attribute Mapping, map the `mail` attribute to `mail`, and map the `employeenumber` attribute to `employeenumber`.

5. Select Create.

   After successfully creating the Fedlet, AM displays the location of the configuration files. Depending on your version of AM, the configuration files are in a `war` directory or .zip file.

   The .zip file is named something like `$HOME/openam/myfedlets/sp/Fedlet.zip` on the system where AM runs.

7.5. Configuring IG As an SP

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*.

7.5.1. Retrieve the Fedlet Configuration Files

To Retrieve the Fedlet Configuration Files

1. Unpack the configuration files from the Fedlet you created in " Create a Fedlet Configuration ". For example, unpack the .zip file as follows:

```
$ cd $HOME/openam/myfedlets/sp
$ unzip Fedlet.zip
```

2. Copy the files to the IG configuration:

```
$ mkdir $HOME/.openig/SAML
$ cp conf/* $HOME/.openig/SAML
$ ls -l $HOME/.openig/SAML

FederationConfig.properties
fedlet.cot
idp-extended.xml
idp.xml
sp-extended.xml
sp.xml
```

3. If the following fedlet adapter header is defined in `sp-extended.xml`, comment it out to prevent issues with timeout:

```
<!--
  <Attribute name="fedletAdapter">
    <Value>com.sun.identity.saml2.plugins.DefaultFedletAdapter</Value>
  </Attribute>
-->
```

4. Restart IG.

7.5.2. Adding a Route for Credential Injection

Create the configuration file `$HOME/.openig/config/routes/05-saml.json`.

On Windows, the file name should be `%appdata%\OpenIG\config\routes\05-saml.json`.

```
{
  "handler": {
    "type": "SamlFederationHandler",
    "config": {
      "assertionMapping": {
        "username": "mail",
        "password": "employeenumber"
      },
      "subjectMapping": "sp-subject-name",
      "redirectURI": "/federate"
    }
  },
  "condition": "${matches(request.uri.path, '^/saml')}",
  "session": "JwtSession"
}
```

The route injects credentials into the context, based on attribute values from the SAML assertion returned on successful authentication. Note the following features of the route:

- The route matches requests to `/saml`.

- The `SamlFederationHandler` extracts the mail and employee number from the SAML assertion and maps them to the session fields `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 then 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 details, see JwtSession(5) in the *Configuration Reference*.

7.5.3. Adding a Route for SAML Federation

Create the configuration file `$HOME/.openig/config/routes/05-federate.json`.

On Windows, the file name should be `%appdata%\OpenIG\config\routes\05-federate.json`.

```
{
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${empty session.username}",
          "handler": {
            "type": "StaticResponseHandler",
            "config": {
              "status": 302,
              "reason": "Found",
              "headers": {
                "Location": [
                  "http://sp.example.com:8080/saml/SPInitiatedSSO"
                ]
              }
            }
          }
        }
      ]
    }
  },
  {
    "handler": {
      "type": "Chain",
      "config": {
        "filters": [
          {
            "type": "StaticRequestFilter",
            "config": {
              "method": "POST",
              "uri": "http://app.example.com:8081/login",
              "form": {
                "username": [
```

```
                    "${session.username[0]}"
                  ],
                  "password": [
                    "${session.password[0]}"
                  ]
                }
              }
            }
          ],
          "handler": "ReverseProxyHandler"
        }
      }
    }
  ]
}
},
"condition": "${matches(request.uri.path, '^/federate')}",
"session": "JwtSession"
}
```

Notice the following features of the route:

- The route matches requests to `/federate`. This is the route you use to test the configuration.

- If the username has not been populated in the context, the user has not yet authenticated with the IDP. In this case,

  - The `DispatchHandler` dispatches requests to the `StaticResponseHandler`.

  - The `StaticResponseHandler` redirects to the SP-initiated SSO end point to initiate SAML 2.0 web browser SSO.

  - After authentication is successful, the `SamlFederationHandler` injects credentials into the session.

  If the credentials have been inserted into the context, or after successful authentication in the previous step, the `DispatchHandler` dispatches requests to the `Chain` to log the user in to the protected application.

- The `StaticRequestFilter` retrieves the first value for the username and password attributes of the SAML assertion.

> **Note**
>
> The attributes of a SAML assertion can contain one or more values, which are made available as a list of strings. Even if an attribute contains a single value, it is made available as a list of strings.

The `StaticRequestFilter` then replaces the browser's original HTTP GET request with an HTTP POST login request that contains the credentials to authenticate.

- 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 details, see JwtSession(5) in the *Configuration Reference*.

> **Tip**
>
> If more dynamic control is needed for the URL where the user agent is redirected, then use the `RelayState` query string parameter in the URL of the redirect `Location` header. The `RelayState` query string parameter specifies where to redirect the user when the SAML 2.0 web browser SSO process is complete. The `RelayState` overrides the redirectURI set in the `SamlFederationHandler`. The `RelayState` value must be URL-encoded. When using an expression, use the `urlEncode()` function to encode the value. For example: `${urlEncodeQueryParameterNameOrValue(contexts.router.originalUri)}`. In the following example, the user is finally redirected to the original URI from the request:

```
"headers": {
    "Location": [
        "http://openig.example.com:8080/saml/SPInitiatedSSO?RelayState=
${urlEncodeQueryParameterNameOrValue(contexts.router.originalUri)}"
    ]
}
```

7.6. Testing the Configuration

7.6.1. Testing IDP-initiated SSO

- Log out of the AM console and select this link for IDP-initiated SSO. The AM login page is displayed.

- Log in to AM with username `george` and password `costanza`. IG returns the response page showing that the user has logged in.

7.6.2. Testing SP-initiated SSO

- Log out of the AM console, and browse to the URL for the route at http://openig.example.com:8080/federate. The AM login page is displayed.

- Log in to AM with the username `george` and password `costanza`. IG returns the response page showing that the user has logged in.

The following sequence diagram shows what just happened.

SP-Initiated SSO

SP Initiated SSO

Browser	AM identity provider	IG service provider	Protected application

1. The user sends an HTTP GET request to the protected application, using the /federate route on IG.

2. Because the username and password are not available, the /federate route directs the request to AM for SPInitiatedSSO.

3. AM requests credentials, and the user logs in.

4. AM directs the request to the /saml route on IG, providing SAML assertions for the user.

5. On the /saml route, the SamlFederationHandler validates the assertions, sets the attributes, and redirects the request to the /federate route.

6. On the /federate route, the DispatchHandler directs the request to the StaticRequestFilter.

7. The StaticRequestFilter retrieves the credentials and replaces the original HTTP GET request with an HTTP POST login request that contains the credentials to authenticate to the protected application.

8. IG returns the response page showing that the user has logged in.

7.7. Example Federation Configuration Files

7.7.1. Circle of Trust

The following example of `$HOME/.openig/SAML/fedlet.cot` defines a CoT between AM as the IDP and an IG 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=
```

7.7.2. SAML Configuration File

The following example of `$HOME/.openig/SAML/sp.xml` defines a SAML configuration file for an IG service provider, `sp`:

```
<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"/>
    <AssertionConsumerService
      index="1"
      Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Artifact"
      Location="http://sp.example.com:8080/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>
```

7.7.3. Extended Configuration File

The following example of `$HOME/.openig/SAML/sp-extended.xml` defines a SAML configuration file for an IG service provider, `sp`:

```
<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>com.sun.identity.saml2.plugins.DefaultFedletAdapter</Value>
        </Attribute>
        -->
        <Attribute name="fedletAdapterEnv">
            <Value></Value>
        </Attribute>
        <Attribute name="spAccountMapper">
            <Value>com.sun.identity.saml2.plugins.DefaultLibrarySPAccountMapper</Value>
        </Attribute>
        <Attribute name="useNameIDAsSPUserID">
            <Value>false</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>employeenumber=employeenumber</Value>
            <Value>mail=mail</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://sp1.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="responseArtifactMessageEncoding">
     <Value>URI</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>
```

```xml
        </Attribute>
        <Attribute name="ECPRequestIDPList">
            <Value></Value>
        </Attribute>
        <Attribute name="ECPRequestIDPListGetComplete">
            <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>
        <Attribute name="spSessionSyncEnabled">
            <Value>false</Value>
        </Attribute>
        <Attribute name="relayStateUrlList">
        </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>false</Value>
        </Attribute>
        <Attribute name="wantAssertionEncrypted">
```
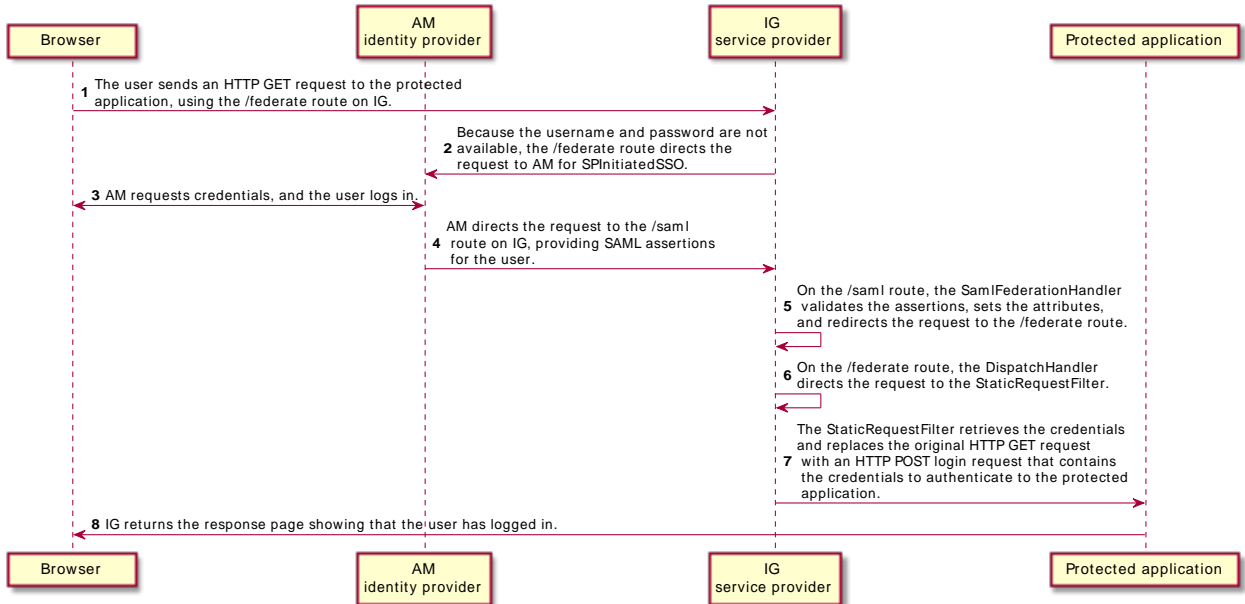
```
                <Value>false</Value>
        </Attribute>
        <Attribute name="cotlist">
                <Value>Circle of Trust</Value>
        </Attribute>
    </XACMLAuthzDecisionQueryConfig>
</EntityConfig>
```

Chapter 8

Acting as an OAuth 2.0 Resource Server

This chapter describes how IG acts as an OAuth 2.0 Resource Server, using the AM token info endpoint and the introspection endpoint.

8.1. About IG As an OAuth 2.0 Resource Server

The OAuth 2.0 Authorization Framework describes a way of allowing a third-party application to access a user's resources without having the user's credentials.

OAuth 2.0 includes the following entities:

- *Resource owner*: The user who owns protected resources on a resource server. For example, a resource owner has photos stored in a web service.

- *Resource server*: The service that gives authorized client applications access to the user's protected resources. In OAuth 2.0, an authorization server grants the client application authorization based on the resource owner's consent. For example, a web service that holds the user's photos.

- *Client*: The application that requests access to the protected resources. For example, a photo printing service requests access to the user's photos.

- *Authorization server*: The 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.

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.

Handling OAuth 2.0 Requests as an OAuth 2.0 Resource Server
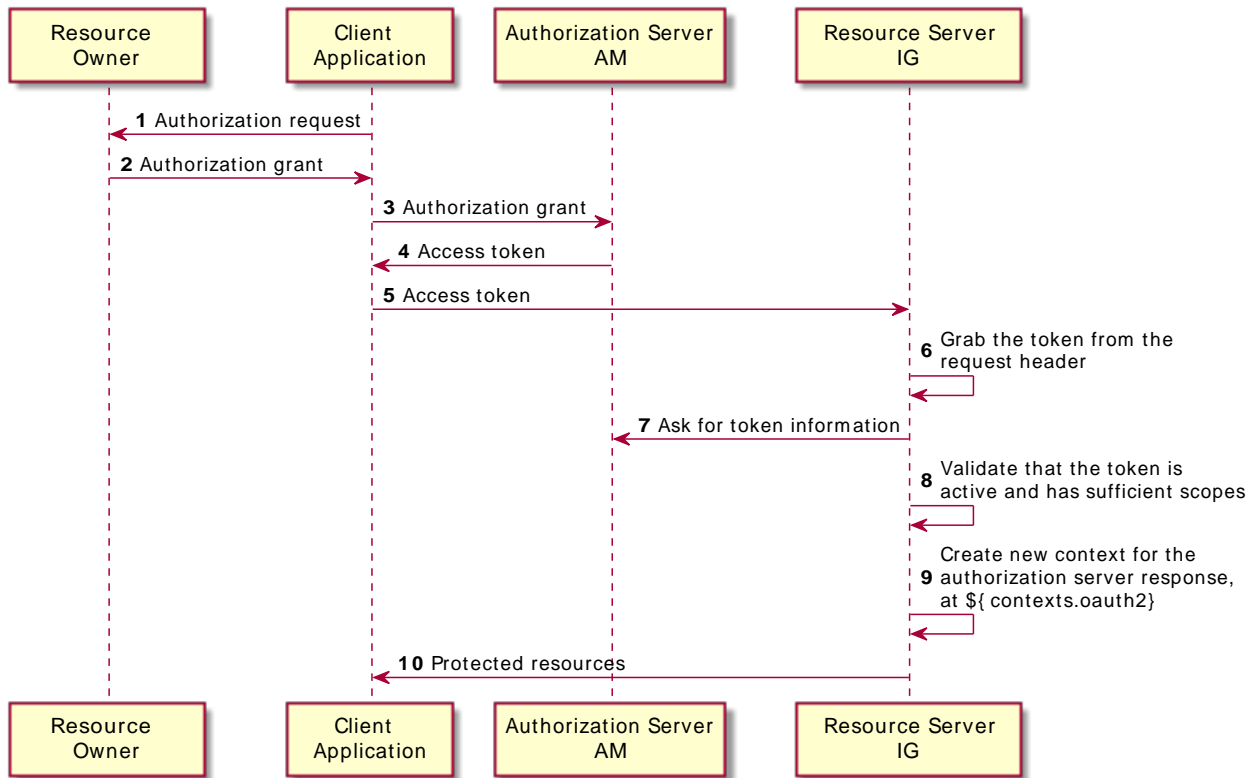
- The application obtains an *authorization grant* from the authorization server, representing the resource owner's consent.

  For information about the different OAuth 2.0 grant mechanisms supported by AM, see OAuth 2.0 Authorization Grant in the AM *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 given by the authorization server in the `Authorization` request header, like this:

```
Authorization: Bearer 7af41ddd-47a4-40dc-b530-a9aa9f7ceda9
```

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 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.

• The application supplies the access token to the resource server, which then validates the access token. If the access token is found to be valid, then the resource server can let the client have access to the resources.

IG validates the access token by submitting it to a token information endpoint or an introspection endpoint. The endpoint typically returns the time until the token expires, the OAuth 2.0 *scopes* associated with the token, and potentially other information.

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.

8.2. Preparing the Tutorial

Tasks for Configuring IG As an OAuth2 Resource Server

Task	See Section(s)
Set up a sample user in AM.	"To Set Up a Sample User in AM"
Set up AM as an authorization server.	"To Set Up AM As an Authorization Server"
Set up IG as a resource server to validate access tokens through the token info endpoint and introspection endpoint.	"To Set Up IG As a Resource Server Using the Token Info Endpoint" "To Set Up IG As a Resource Server Using the Introspection Endpoint"
Test the setup.	"To Test the Setup"
Add downstream filters that use information from the context to log you in to the sample application automatically.	"Using the Context to Log In To the Sample Application"

8.3. Setting Up AM As an Authorization Server

For more complete information about configuring AM as an OAuth 2.0 authorization service, see Configuring the OAuth 2.0 Authorization Service in the AM *OAuth 2.0 Guide*.

Before you start, install and configure AM on http://openam.example.com:8088/openam, with the default configuration. If you use a different configuration, substitute in the tutorial accordingly.

To Set Up a Sample User in AM

If you haven't set up the user George Costanza in a previous tutorial, do so now.

1. In the AM console, select the top level realm, and then select Identities.

2. Click Add Identity and create a user with the following values:

   - ID: `george`

   - Password: `costanza`

3. In the User window, set the following values:

   - Email Address: `george`

   - Employee number: `costanza`

To Set Up AM As an Authorization Server

1. In the AM console, configure an OAuth 2.0 Authorization Server:

   a. In the top level realm, select Configure OAuth Provider > Configure OAuth 2.0.

   b. Accept all of the default values and select Create.

2. Configure a profile for the third-party application to request OAuth 2.0 access tokens:

   a. In the top level realm, select Applications > OAuth 2.0.

   b. Add a client with the following values:

      - Client ID: `client-application`

      - Client secret: `password`

      - Scope(s): `mail employeenumber`

      This tutorial uses `mail` to hold the username, and `employeenumber` to hold the password. The attributes are part of a user's profile included with the default AM configuration, and are not needed for anything else in this tutorial. In a real deployment, you would no doubt use other attributes that depend on how the real user profiles are configured.

3. If you plan to use the token introspection endpoint to validate the access tokens, configure a profile for the resource server:

   a. In the top level realm, select Applications > OAuth 2.0.

   b. Add a client with the following values:

- Client ID: `resource-server`

- Client secret `password`

- Scope(s): `am-introspect-all-tokens`

    The client with this scope is authorized to examine (introspect) tokens.

4. Log out of AM.

8.4. Setting Up IG As a Resource Server

This section describes how to set up IG as an OAuth 2.0 resource server, using the AM token info endpoint and the introspection endpoint. For more information about the OAuth 2.0 resource server, see OAuth2ResourceServerFilter(5) in the *Configuration Reference*.

8.4.1. Validating Access Tokens Through the Token Info Endpoint

This section sets up IG as an OAuth 2.0 resource server, using the AM token info endpoint.

To configure IG without using Studio, add the route in the following procedure to the IG configuration as `$HOME/.openig/config/routes/rs-tokeninfo.json`. On Windows, add the route as `%appdata%\OpenIG\config\routes\rs-tokeninfo.json`.

After completing this procedure, test it as described in "Testing the Setup", or set up the example for the introspection endpoint as described in "Validating Access Tokens Through the Introspection Endpoint".

To Set Up IG As a Resource Server Using the Token Info Endpoint

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*, and access Studio as described in "Accessing Studio" in the *Getting Started Guide*. IG must be running in development mode.

1. Browse to http://openig.example.com:8080/openig/studio, and select ➕ Protect an Application.

2. Choose ☰ Structured to use the predefined menus and templates.

3. In the ➕ Create a route window, enter `http://app.example.com:8081/rs-tokeninfo`, and select Create route.

4. Configure authorization:

    a. Select 🔍 Authorization, and then enable it.

    b. Select OAuth 2.0 Resource Server.

c. Enter the following information to reflect the configuration in "To Set Up AM As an Authorization Server", and then save the settings:

- Scopes: `mail employeenumber`

- Access token resolver: `AM token info endpoint`

- AM server: Configure an AM server to use for the token info endpoint:

  - URI: `http://openam.example.com:8088/openam`

  - Version: The version of the AM server instance, with at least one point. For example, `6.0`.

  The endpoint is extrapolated from the URL of the AM server.

- Require HTTPS: Deselect this option

- Enable cache: Deselect this option

Leave all other values as default.

To view the route so far, on the top-right of the screen select ⋮ and ⬈ Display.

5. Add a StaticResponseHandler:

a. 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.

b. Replace the last ReverseProxyHandler in the route with the following StaticResponseHandler, and then save the route:

```
"handler": {
  "type": "StaticResponseHandler",
  "config": {
    "status": 200,
    "entity": "<html><body><h2>Decoded access token: ${contexts.oauth2.accessToken.info}</h2></
body></html>"
  }
}
```

6. On the top-right of the screen, select ⋮ and ⬈ Display to review the route.

Make sure that the following route is displayed:

```
{
```

```
    "name": "rs-tokeninfo",
    "baseURI": "http://app.example.com:8081",
    "condition": "${matches(request.uri.path, '^/rs-tokeninfo')}",
    "monitor": false,
    "heap": [
      {
        "name": "AmService-1",
        "type": "AmService",
        "config": {
          "url": "http://openam.example.com:8088/openam",
          "realm": "/",
          "ssoTokenHeader": "iPlanetDirectoryPro",
          "version": "6.0"
        }
      }
    ],
    "handler": {
      "type": "Chain",
      "config": {
        "filters": [
          {
            "name": "OAuth2ResourceServerFilter-1",
            "type": "OAuth2ResourceServerFilter",
            "config": {
              "scopes": [
                "mail",
                "employeenumber"
              ],
              "requireHttps": false,
              "realm": "OpenIG",
              "accessTokenResolver": {
                "name": "token-resolver-1",
                "type": "OpenAmAccessTokenResolver",
                "config": {
                  "amService": "AmService-1"
                }
              }
            }
          }
        ],
        "handler": {
          "type": "StaticResponseHandler",
          "config": {
            "status": 200,
            "entity": "<html><body><h2>Decoded access token: ${contexts.oauth2.accessToken.info}</h2></
body></html>"
          }
        }
      }
    }
}
```

> **Note**
>
> If necessary, change the value of `version` for AmService to your version of AM.

Notice the following features of the route:

- The route matches requests to `/rs-tokeninfo`.

- The `OAuth2ResourceServerFilter` expects an OAuth 2.0 access token in the header of the incoming authorization request, with the scopes `mail` and `employeenumber`.

  The `accessTokenResolver` uses the AM server declared in the heap. The token info 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 successfully 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. For an example that uses the scopes `email` and `employeenumber` to log the user in to the sample application, see "Using the Context to Log In To the Sample Application".

  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, OAuth 2.0 Bearer Token Usage.

- The StaticResponseHandler returns the content of the access token from the context `${contexts .oauth2.accessToken.info}`.

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.

8.4.2. Validating Access Tokens Through the Introspection Endpoint

This section sets up IG as an OAuth 2.0 resource server, using the introspection endpoint.

To configure IG without using Studio, add the route in the following procedure to the IG configuration as `$HOME/.openig/config/routes/rs-introspect.json`. On Windows, add the route as `%appdata%\OpenIG\config \routes\rs-introspect.json`.

To Set Up IG As a Resource Server Using the Introspection Endpoint

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*, and access Studio as described in "Accessing Studio" in the *Getting Started Guide*. IG must be running in development mode.

1. Browse to http://openig.example.com:8080/openig/studio, and select ⊕ Protect an Application.

2. Choose ☰ Structured to use the predefined menus and templates.

3. In the ✚ Create a route window, enter `http://app.example.com:8081/rs-introspect`, and select Create route.

4. Configure authorization:

   a. Select 🔑 Authorization, and then enable it.

   b. Select OAuth 2.0 Resource Server.

   c. Enter the following information to reflect the configuration in "To Set Up AM As an Authorization Server", and then save the settings:

      • Scopes: `mail employeenumber`

      • Access token resolver: `OAuth 2.0 introspection endpoint`

      • Introspection endpoint URI: `http://openam.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 at the end of "To Set Up AM As an Authorization Server".

      • Require HTTPS: Deselect this option

      • Enable cache: Deselect this option

      Leave all other values as default.

      To view the route so far, on the top-right of the screen select ⋮ and ↗ Display.

5. Add a StaticResponseHandler:

   a. 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.

   b. Replace the last ReverseProxyHandler in the route with the following StaticResponseHandler, and then save the route:

```
"handler": {
  "type": "StaticResponseHandler",
  "config": {
    "status": 200,
    "entity": "<html><body><h2>Decoded access token: ${contexts.oauth2.accessToken.info}</h2></
body></html>"
  }
}
```

6. On the top-right of the screen, select ⋮ and ⧉ Display to review the route.

Make sure that the following route is displayed:

```
{
  "name": "rs-introspect",
  "monitor": false,
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/rs-introspect')}",
  "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": {
                "endpoint": "http://openam.example.com:8088/openam/oauth2/introspect",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HeaderFilter",
                        "config": {
                          "messageType": "request",
                          "add": {
                            "Authorization": [
                              "Basic ${encodeBase64('resource-server:password')}"
                            ]
                          }
                        }
                      }
                    ],
                    "handler": "ForgeRockClientHandler"
                  }
                }
```

```
            }
          }
        }
      }
    ],
    "handler": {
      "type": "StaticResponseHandler",
      "config": {
        "status": 200,
        "entity": "<html><body><h2>Decoded access token: ${contexts.oauth2.accessToken.info}</h2></
body></html>"
      }
    }
  }
 }
}
```

Notice the following features of the route compared to `rs-tokeninfo.json`:

- The route matches requests to `/rs-introspect`.

- The `accessTokenResolver` uses the token introspection endpoint to validate the access token.

- The `providerHandler` is a chain that adds an authorization header to the request. The header
  contains the username and password of the OAuth 2.0 client with the scope to examine
  (introspect) access tokens.

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.

8.5. Testing the Setup

The following configurations get an access token from AM and use it to access IG, which then uses
the OAuth 2.0 resource owner password credentials authorization grant.

To Test the Setup

1. In a terminal window, use a **curl** command similar to the following to retrieve an access token:

```
$ curl \
--user "client-application:password"
 \
--data "grant_type=password&username=george&password=costanza&scope=mail%20employeenumber" \
http://openam.example.com:8088/openam/oauth2/access_token

{
"access_token":"aba19a55-468d-45e2-b1c4-decc7202faea",
"scope":"employeenumber mail",
"token_type":"Bearer",
"expires_in":3599
}
```

2. Validate the access token:

    • To use the `/oauth2/tokeninfo` endpoint, run the following command, replacing *<access_token>* with the access token returned by the previous step:

    ```
    $ curl \
    --header "Authorization: Bearer <access_token>" \
    http://openig.example.com:8080/rs-tokeninfo

    {
      access_token = f12c7d3e - 0183 - 4 f89 - 8 f5d - aa3055713a96,
      employeenumber = costanza,
      mail = george,
      grant_type = password,
      scope = [employeenumber, mail],
      realm = /,
      token_type = Bearer,
      expires_in = 32606,
      client_id = client-application
    }
    ```

    Note that the token info endpoint returns the scopes, `employeenumber` and `mail`.

    • To use the `/oauth2/introspect` endpoint, run the following command, replacing *<access_token>* with the access token returned by the previous step:

    ```
    $ curl \
    --header "Authorization: Bearer <access_token>" \
    http://openig.example.com:8080/rs-introspect

    {
      active = true,
      scope = employeenumber mail,
      client_id = client-application,
      user_id = george,
      token_type = access_token,
      exp = 1493721361,
      sub = george,
      iss = http: //openam.example.com:8088/openam/oauth2
    }
    ```

Note that the token introspection endpoint returns different information than the token info endpoint.

8.6. Using the Context to Log In To the Sample Application

The token info endpoint returns the scopes, `employeenumber` and `mail` in the context. This section contains an example route that retrieves the scopes and uses them to log the user directly in to the sample application.

To Log In To the Sample Application By Using the Token Info

1. Run the example in "To Set Up IG As a Resource Server Using the Token Info Endpoint".

2. On the top-right of the Studio screen, select ⋮ and ⟨⟩ Editor mode.

3. Remove the StaticResponseHandler, and add the following filters and handler:

   - An AssignmentFilter to access the token info through the context, and inject the credentials into `session`.

   - A StaticRequestFilter to retrieve 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.

   - A ReverseProxyHandler to submit the request to the sample application.

   An example route with a different name and condition is as follows:

```
{
  "name": "rs-pwreplay",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/rs-pwreplay')}",
  "monitor": false,
  "heap": [
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://openam.example.com:8088/openam",
        "realm": "/",
        "ssoTokenHeader": "iPlanetDirectoryPro",
        "version": "6.0"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
```

```
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "token-resolver-1",
              "type": "OpenAmAccessTokenResolver",
              "config": {
                "amService": "AmService-1"
              }
            }
          }
        },
        {
          "type": "AssignmentFilter",
          "config": {
            "onRequest": [{
              "target": "${session.username}",
              "value": "${contexts.oauth2.accessToken.info.mail}"
            },
              {
                "target": "${session.password}",
                "value": "${contexts.oauth2.accessToken.info.employeenumber}"
              }
            ]
          }
        },
        {
          "type": "StaticRequestFilter",
          "config": {
            "method": "POST",
            "uri": "http://app.example.com:8081/login",
            "form": {
              "username": [
                "${session.username}"
              ],
              "password": [
                "${session.password}"
              ]
            }
          }
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  }
}
```

> **Note**
>
> If necessary, change the value of `version` for AmService to your version of AM.

4. In a terminal window, use a **curl** command similar to the following to access the example route, replacing *<access_token>* with a new access token or the one returned in "To Test the Setup":

```
$ curl \
--header "Authorization: Bearer <access_token>" \
http://openig.example.com:8080/rs-pwreplay
```

HTML for the sample application should be displayed.

Chapter 9

Acting As an OAuth 2.0 Client or OpenID Connect Relying Party

IG helps integrate applications into OAuth 2.0 and OpenID Connect deployments. In this chapter, you will:

- Configure IG as an OpenID Connect 1.0 relying party

- Configure IG to use OpenID Connect discovery and dynamic client registration

9.1. About IG As an OAuth 2.0 Client

As described in "*Acting as an OAuth 2.0 Resource Server*", an OAuth 2.0 client is a third-party application that needs access to a user's protected resources.

IG can act as an OAuth 2.0 client when you configure an OAuth2ClientFilter as described in OAuth2ClientFilter(5) in the *Configuration Reference*. The OAuth2ClientFilter handles the process of allowing the user to select a provider, and redirecting the user through the authentication and authorization steps of an OAuth 2.0 authorization code grant. The code grant results in the authorization server returning an access token to the filter.

When an authorization grant succeeds, the OAuth2ClientFilter injects the access token data into a configurable target in the context so that subsequent filters and handlers have access to the access token. Subsequent requests can use the access token without reauthentication. If an authorization grant fails, the `failureHandler` is invoked.

If the protected application is an OAuth 2.0 resource server, then IG can send the access token with the resource request.

9.2. About IG As an OpenID Connect 1.0 Relying Party

The specifications available through the OpenID Connect site describe the OpenID Connect 1.0 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 aims to access.

OpenID Connect 1.0 has the following key 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 has the end user consent to provide the RP with 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 this as a key to its own user profile.

  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 aims 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 ultimate 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.

9.3. Installation Overview

This tutorial takes you through the steps to set up AM as an OpenID Connect provider and IG as an OpenID Connect relying party. For an example configuration with two identity providers, AM and Google, and a login handler, see "Example Configuration For Multiple Identity Providers" in the *Configuration Reference*.

Tasks for Configuring OpenID Connect

Task	See Section(s)
Set up AM as an OpenID Connect provider.	" Setting Up AM for OpenID Connect "
Set up IG as a relying party for browser requests to the home page of the sample application.	"Setting Up IG As a Relying Party"

Task	See Section(s)
Test the configuration.	"Testing the Configuration"

9.4. Setting Up AM for OpenID Connect

In this part, you create a sample user and set up AM as an OpenID Connect provider.

Before you start, install and configure AM on http://openam.example.com:8088/openam, with the default configuration. If you use a different configuration, substitute in the tutorial accordingly.

To Set Up a Sample User

If you haven't set up the user George Costanza in a previous tutorial, do so now.

1. In the AM console, select the top level realm, and then select Identities.

2. Click Add Identity and create a user with the following values:

   - ID: `george`

   - Password: `costanza`

3. In the User window, select the new user and set Email Address: `george@example.com`.

To Set Up AM as an OpenID Connect Provider

1. Configure AM as an OAuth 2.0 Authorization Server and OpenID Connect Provider.

   a. In the AM console, select the top level realm and browse to Configure OAuth Provider > Configure OpenID Connect.

   b. Accept the default values and click Create.

2. Register an OpenID Connect relying party. This step enables IG to communicate as an OAuth 2.0 relying party with AM.

   a. In the top level realm, select Applications > OAuth 2.0.

   b. Add a client with the following values:

      - Client ID: `oidc_client`

      - Client secret: `password`

      - Redirection URIs: `http://openig.example.com:8080/home/id_token/callback`

      - Scope(s): `openid`, `profile`, and `email`.

    c.   Click Add then on Signing and Encryption tab:

        • Change ID Token Signing Algorithm to `HS256`, `HS384`, or `HS512`.

        Because the algorithm must be HMAC, the value of `RS256` is not okay.

3.   Log out of AM.

9.5. Setting Up IG As a Relying Party

This section describes how to use Studio to configure IG as a relying party for browser requests to the home page of the sample application. The example refers to the provider configuration in " Setting Up AM for OpenID Connect ".

To configure IG without using Studio, add the route in "Route for IG As a Relying Party" to the IG configuration as `$HOME/.openig/config/routes/07-openid.json`. On Windows, add the route as `%appdata% \OpenIG\config\routes\07-openid.json`.

To Set Up IG As a Relying Party

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*, and access Studio as described in "Accessing Studio" in the *Getting Started Guide*. IG must be running in development mode.

1.   Browse to http://openig.example.com:8080/openig/studio, and select ⊕ Protect an Application.

2.   Choose ☰ Structured to use the predefined menus and templates.

3.   In the ✚ Create a route window, select Advanced Options, enter the following information, and then select Create route:

    • Base URI: `http://app.example.com:8081`

    • Condition: Path: `/home/id_token`

    • Name: `07-openid`

4.   Select Authentication, and then enable authentication.

5.   Select OpenID Connect, and then select the following options to reflect the configuration in " To Set Up AM as an OpenID Connect Provider ":

    • Client Filter:

        • Client Endpoint: `/home/id_token`

        • Require HTTPS: Deselect this option

- Client Registration:

  - Client ID: `oidc_client`

  - Client secret: `password`

  - Scopes: `openid profile email`

  - Basic authentication: Select this option

- Issuer:

  - Well-known Endpoint: `http://openam.example.com:8088/openam/oauth2/.well-known/openid-configuration`

Leave the other settings with their default values.

6. On the top-right of the screen, select ⋮ and ☑ Display to review the route. The route in "Route for IG As a Relying Party" should be displayed.

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.

Route for IG As a Relying Party

```
{
  "name": "07-openid",
  "baseURI": "http://app.example.com:8081",
  "monitor": false,
  "condition": "${matches(request.uri.path, '^/home/id_token')}",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "OAuth2ClientFilter",
          "name": "OAuth2Client",
          "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",
              "clientSecret": "password",
              "issuer": {
                "name": "Issuer",
                "type": "Issuer",
                "config": {
                  "wellKnownEndpoint": "http://openam.example.com:8088/openam/oauth2/.well-known/
openid-configuration"
                }
              },
              "scopes": [
                "openid",
                "profile",
                "email"
              ],
              "tokenEndpointAuthMethod": "client_secret_basic"
            }
          }
        ],
        "requireHttps": false,
        "cacheExpiration" : "disabled"
      }
    }
  ],
  "handler": "ReverseProxyHandler"
  }
 }
}
```

Notice the following features about the route:

- The route matches requests to `/home/id_token`.

- The `OAuth2ClientFilter` 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 " Setting Up AM for OpenID Connect ".

- 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 `http://openig.example.com:8080/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.

9.6. Testing the Configuration

To Test the Configuration

1. With IG running, access http://openig.example.com:8080/home/id_token.

   The AM login screen is displayed.

2. Log in to AM with the username `george` and password `costanza`.

   An OpenID Connect request to access private information is displayed.

3. Select Allow.

   The home page of the sample application is displayed.

   What's happening behind the scenes?

   - When IG gets the browser request, the OAuth2ClientFilter redirects you to authenticate with AM and consent to authorize access to user information. After you authorize access, AM returns an access token to the filter.

   - The OAuth2ClientFilter uses the access token to get the user information, and then injects the authorization state information into `attributes.openid`.

   - The ReverseProxyHandler redirects the request to the home page of the sample application.

9.7. Adapting the Configuration to Authenticate Automatically to the Sample Application

To authenticate automatically to the sample application, add a StaticRequestFilter like the following to the end of the chain in `07-openid.json`:

FORGEROCK

Acting As an OAuth 2.0 Client or OpenID Connect Relying Party
Using OpenID Connect Discovery and Dynamic Client Registration

```
{
  "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.

9.8. Using OpenID Connect Discovery and Dynamic Client Registration

OpenID Connect defines mechanisms for discovering and dynamically registering with an identity provider that is not known in advance. These mechanisms are specified in OpenID Connect Discovery and OpenID Connect Dynamic Client Registration. IG supports discovery and dynamic registration. In this section you will learn how to configure IG to try these features with AM.

Although this tutorial focuses on OpenID Connect dynamic registration, IG also supports dynamic registration as described in RFC 7591, *OAuth 2.0 Dynamic Client Registration Protocol*.

9.8.1. Preparing to Try Discovery and Dynamic Client Registration

This short tutorial builds on the previous tutorial in this chapter. If you have not already done so, start by performing the steps described in "Installation Overview". This tutorial requires a recent sample application, as the newer versions include a small WebFinger service that is used here.

When ready, complete preparations for OpenID Connect discovery and dynamic client registration:

- "Preparing AM for OpenID Connect Dynamic Registration"

- "Preparing IG for Discovery and Dynamic Registration"

Preparing AM for OpenID Connect Dynamic Registration

By default, AM does not allow dynamic registration without an access token.

After carrying out the steps described in " Setting Up AM for OpenID Connect ", also perform these steps:

FORGEROCK

Acting As an OAuth 2.0 Client or OpenID Connect Relying Party
Preparing to Try Discovery and Dynamic Client Registration

1. Log in to AM console as administrator.

2. In the top level realm, select Services, and then display the OAuth2 Provider configuration.

3. In the Client Dynamic Registration tab, enable Allow Open Dynamic Client Registration.

   In earlier versions of AM, this option is in Advanced OpenID Connect.

4. Log out of AM.

Preparing IG for Discovery and Dynamic Registration

Follow these steps to add a route demonstrating OpenID Connect discovery and dynamic client registration:

1. Add the following route to the IG configuration as `$HOME/.openig/config/routes/07-discovery.json` (on Windows, `%appdata%\OpenIG\config\routes\07-discovery.json`):

```
{
  "heap": [
    {
      "name": "DiscoveryPage",
      "type": "StaticResponseHandler",
      "config": {
        "status": 200,
        "reason": "OK",
        "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='george or george@example.com' />
            <input type='hidden' name='goto'
                   value='${contexts.router.originalUri}' />
            </form>
            <script>
            // The sample application handles the WebFinger request,
            // so make sure the request is sent to the 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>"
```

```
      }
    }
  ],
  "name": "07-discovery",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "DynamicallyRegisteredClient",
          "type": "OAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/discovery",
            "requireHttps": false,
            "requireLogin": true,
            "target": "${attributes.openid}",
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "comment": "Trivial failure handler for debugging only",
                "status": 500,
                "reason": "Error",
                "entity": "${attributes.openid}"
              }
            },
            "loginHandler": "DiscoveryPage",
            "metadata": {
              "client_name": "My Dynamically Registered Client",
              "redirect_uris": [
                "http://openig.example.com:8080/discovery/callback"
              ],
              "scopes": [
                "openid",
                "profile"
              ]
            }
          }
        }
      ],
      "handler": {
        "type": "Chain",
        "config": {
          "filters": [
            {
              "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}"
                  ]
                }
              }
            }
          ],
```

FORGEROCK

Acting As an OAuth 2.0 Client or OpenID Connect Relying Party
Preparing to Try Discovery and Dynamic Client Registration

```
              "handler": "ReverseProxyHandler"
          }
        }
      }
    }
  },
  "condition": "${matches(request.uri.path, '^/discovery')}",
  "baseURI": "http://openig.example.com:8080"
}
```

2. Consider the differences with `07-openid.json`:

    -   For discovery and dynamic client registration, no issuer or client registration is defined. Instead a `StaticResponseHandler` is used as a login handler for the client filter.

        The static response handler serves an HTML page that provides important pieces of information to IG:

        -   The value of a `discovery` parameter.

            IG uses the value to perform OpenID Connect discovery. Examples from the specification include `acct:joe@example.com`, `https://example.com:8080/`, and `https://example.com/joe`. First, IG extracts the domain host and port from the value, and attempts to find a match in the `supportedDomains` lists for any issuers that are already configured for the route. If it finds a match, then it can potentially use the issuer's registration end point and avoid an additional request to look up the user's issuer using the WebFinger protocol. If there is no match in the supported domains lists, IG uses the `discovery` value as the `resource` for a WebFinger request according to the OpenID Connect Discovery protocol.

            On success, IG has either found an appropriate issuer in the configuration, or found the issuer using the WebFinger protocol. IG can thus proceed to dynamic client registration.

            The small JavaScript function in the HTML page transforms user input into a useful `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.

        -   The value of a `goto` parameter.

            The `goto` parameter takes a URI that tells IG where to redirect the end user's browser once the process is complete and IG has injected the OpenID Connect user information into the context. In this case, the user is redirected back to this route so that the innermost chain of the configuration can log the user in to the protected application.

    -   The OAuth2ClientFilter specifies the following configuration:

        -   Login handler to point to the login page described above.

        -   Metadata that IG uses to prepare the dynamic registration request, including:

            -   Client name

            -   Redirection URIs

FORGEROCK

Acting As an OAuth 2.0 Client or OpenID Connect Relying Party
Trying OpenID Connect Discovery and Dynamic Client Registration

As set out in RFCs for OAuth2 and OpenID, redirection URIs are mandatory. One of the registered redirection URI values **must** exactly match the clientEndpoint/callback URI.

- Scopes:

  - `scope` is available for dynamic client registration with AM from version 5.5, and identity providers that support RFC 7591, *OAuth 2.0 Dynamic Client Registration Protocol*.

  - `scopes` is available for dynamic client registration with AM 5.5 and earlier versions only.

  For the option to dynamically register with a wider range of identity providers, you can use both `scope` and `scopes` at the same time.

- `07-discovery.json` uses the path `/discovery`, whereas `07-openid.json` uses `/home/id_token`.

  This distinction makes it easy to keep traffic separate on the two routes with a simple condition as in the following:

```
"condition": "${matches(request.uri.path, '^/discovery')}"
```

9.8.2. Trying OpenID Connect Discovery and Dynamic Client Registration

After following the steps described in "Preparing to Try Discovery and Dynamic Client Registration", test your configuration by browsing to IG at http://openig.example.com:8080/discovery.

Provide the following email address: `george@example.com`.

When redirected to the AM login page, log in as user `george`, password `costanza`, and then allow the application access to user information.

If successful, IG logs you in to the sample application as George Costanza, and the sample application returns George's page.

What is happening behind the scenes?

After IG gets the browser request, it returns the example page for discovery. You provide a user ID or email address, and the page transforms that into a `discovery` value. The value is tailored to let IG use the sample application as a WebFinger server. (In the real world the WebFinger server is more likely a service on the issuer's domain, not part of the protected application. For the purposes of this tutorial the WebFinger service has been embedded in the sample application to avoid leaving you with another server to manage during the tutorial.)

IG learns from the WebFinger service that AM is the issuer for the user. IG retrieves the OpenID Provider configuration from AM, and registers itself dynamically with AM, using the redirection URIs and scopes specified in the OAuth 2.0 client filter configuration.

Once the issuer and client registration are properly configured, the OAuth 2.0 client filter redirects the browser to AM for authentication and authorization to access to the user information. The rest is the same as the previous tutorial in this chapter. For details, see "Testing the Configuration".

FORGEROCK

Acting As an OAuth 2.0 Client or OpenID Connect Relying Party
Trying OpenID Connect Discovery and Dynamic Client Registration

IG reuses issuer and client registration configurations that it builds after discovery and dynamic registration. These dynamically generated configuration objects are held in memory, and do not persist when IG is restarted.

Chapter 10

Transforming OpenID Connect ID Tokens Into SAML Assertions

IG provides a token transformation filter to transform OpenID Connect ID tokens (id_tokens) issued by AM into SAML 2.0 assertions. The implementation uses the AM REST Security Token Service (STS) APIs, where the subject confirmation method is Bearer.

Many enterprises use existing or legacy, SAML 2.0-based SSO, but many mobile and social applications are managed by OAuth 2.0/OpenID. Use the IG token transformation filter to bridge OAuth 2.0 and SAML 2.0 frameworks.

10.1. About Token Transformation

The following figure illustrates the basic flow of information between a request, IG, the AM OAuth 2.0 and STS modules, and an application. For a more detailed view of the flow, see "Flow of Events".

Token Transformation
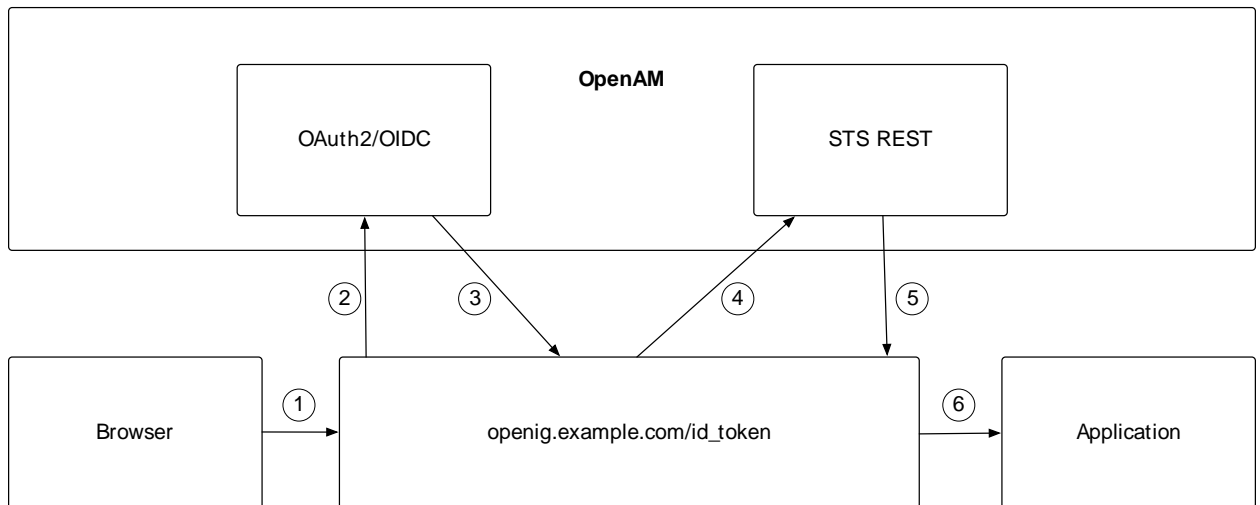
The basic process is as follows:

1. A user tries to access to a protected resource.

2. If the user is not authenticated, the OAuth2ClientFilter 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 OAuth2ClientFilter. 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.

10.2. Installation Overview

This tutorial takes you through the steps to set up AM as an OAuth 2.0 provider and IG as an OpenID Connect relying party.

When a user makes a request, the `OAuth2ClientFilter` returns an id_token. The TokenTransformationFilter references a REST STS instance that you set up in AM to transform the id_token into a SAML 2.0 assertion.

You need to be logged in to AM as administrator to set up the configuration for token transformation, but you do not need special privileges to request a token transformation. For more information, see TokenTransformationFilter(5) in the *Configuration Reference*.

Before you start this tutorial:

• Prepare IG as described in "*First Steps*" in the *Getting Started Guide*.

• Install and configure AM on http://openam.example.com:8088/openam, with the default configuration. If you use a different configuration, substitute in the tutorial accordingly.

Tasks for Configuring Token Transformation

Task	See Section(s)
Set up AM as an OpenID Connect provider and IG as an an OpenID Connect relying party.	" Setting Up AM for OpenID Connect "
Create a Bearer authentication module to validate the id_token and AM user.	" Creating a Bearer Module "
Create an STS REST instance to transform the id_token into a SAML assertion.	"Creating an Instance of STS REST"

Task	See Section(s)
Create the routes in IG to send the requests and test the setup.	" Setting Up IG Routes for Token Transformation "

10.3. Setting Up AM for Token Transformation

10.3.1. Setting Up AM as an OpenID Connect Provider

Set up AM as described in " Setting Up AM for OpenID Connect ". IG authenticates with AM and retrieves an OpenID Connect ID token (id_token) to be transformed by STS.

10.3.2. Creating a Bearer Module

The OpenID Connect ID token bearer module expects an id_token in an HTTP request header. It validates the id_token, and, if successful, looks up the AM user profile corresponding to the end user for whom the id_token was issued. Assuming the id_token is valid and the profile is found, the module authenticates the AM user.

You configure the token bearer module to specify how AM gets the information to validate the id_token, which request header contains the id_token, the identifier for the provider who issued the id_token, and how to map the id_token claims to an AM user profile.

To Create a Bearer Module for the id_token

1. Log in to the AM console as administrator.

2. In the top level realm, select Authentication > Modules.

3. Select Add Module and create a new bearer module with the following characteristics:

    • Module name: `oidc`

    • Type: `OpenID Connect id_token bearer`

4. In the configuration page, enter the following values and leave the other fields with the default 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://openam.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`

5. Select Save Changes.

10.3.3. Creating an Instance of STS REST

The REST STS instance exposes a preconfigured transformation under a specific REST endpoint.

For more information about setting up a REST STS instance, see the AM *Security Token Service Guide* .

For information about configuring keystores, see Configuring Keystores in the AM *Setup and Maintenance Guide*.

1. In the top level realm, select STS.

2. Create a Rest STS instance with the following characteristics:

   - Deployment Configuration

     - Deployment Url Element: `openig`

       This value identifies the STS instance and is used by the `instance` parameter in the TokenTransformationFilter.

   - SAML2 Token

     - SAML2 issuer Id: `OpenAM`

     - Service Provider Entity Id: `openig_sp`

     - NameIdFormat: Select `nameid:format:transient`

       > **Note**
       >
       > For STS, it isn't necessary to create a SAML SP configuration in AM.

   - OpenId Connect Token

     - OpenIdConnect Token Provider Issuer Id: `oidc`

     - Token signature algorithm: Enter a value that is consistent with " To Set Up AM as an OpenID Connect Provider ", for example, `HMAC SHA 256`

- Client secret (for HMAC-signed-tokens): `password`

- The audience for issued tokens: `oidc_client`.

3. Select Create and go to the SAML2 Token tab:

- Attribute Mappings: Add `password=mail` and `userName=uid`

4. Log out of AM.

10.4. Setting Up IG Routes for Token Transformation

The following sequence diagram shows what happens when you set up and access routes for token transformation.

Flow of Events
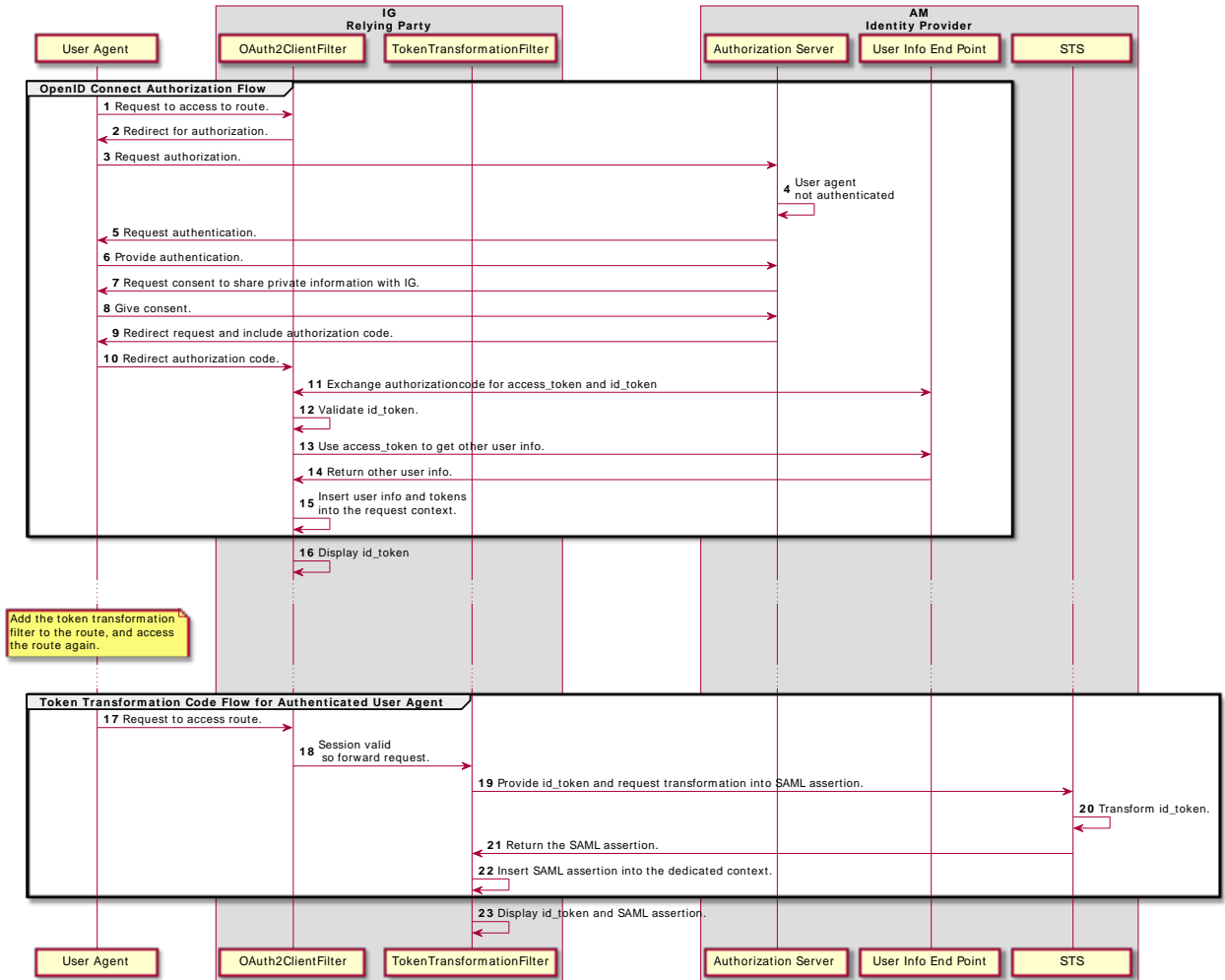
To Set Up IG For Token Transformation

The first steps of this procedure are the same as those for "To Set Up IG As a Relying Party". Adapt that route or follow this procedure to set up a new route.

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*, and access Studio as described in "Accessing Studio" in the *Getting Started Guide*. IG must be running in development mode.

1. Browse to http://openig.example.com:8080/openig/studio, and select ⊕ Protect an Application.

2. Choose ☰ Structured to use the predefined menus and templates.

3. In the ✚ Create a route window, select Advanced Options, enter the following information, and then select Create route:

   - Base URI: `http://app.example.com:8081`

   - Condition: Path: `/home/id_token`

   - Name: `07-openid`

4. Set up authentication:

   a. Select Authentication, and then enable authentication.

   b. Select OpenID Connect, and then select the following options to reflect the configuration in " To Set Up AM as an OpenID Connect Provider ":

      - Client Filter:

        - Client Endpoint: `/home/id_token`

        - Require HTTPS: Deselect this option

      - Client Registration:

        - Client ID: `oidc_client`

        - Client secret: `password`

        - Scopes: `openid profile email`

        - Basic authentication: Select this option

      - Issuer:

        - Well-known Endpoint: `http://openam.example.com:8088/openam/oauth2/.well-known/openid-configuration`

      Leave the other settings with their default values.

5. Set up token transformation:

   a. Select ⇄ Token transformation and enable it.

   b. In AM server, select Configure, and set up the following AM server:

- URI: `http://openam.example.com:8088/openam`

  The URI of the AM service to use for authentication and REST STS requests.

- Version: Enter the version of your AM server instance

Leave the other fields of AM server with their default values.

c. Enter the following values for token transformation, and then save the route:

- Username: `oidc_client`

- Password: `password`

- id_token: `${attributes.openid.id_token}`

- Instance: `openig`

6. Add a StaticResponseHandler:

a. 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.

b. Replace the last ReverseProxyHandler in the route with the following StaticResponseHandler, and then save the route:

```
"handler": {
  "type": "StaticResponseHandler",
  "config": {
    "entity": "{\"id_token\":\n\"${attributes.openid.id_token}\"} \n\n\n{\"saml_assertions\":\n
\"${contexts.sts.issuedToken}\"}",
      "reason": "Found",
      "status": 200
  }
}
```

7. On the top-right of the screen, select ⋮ and ⟳ Display to review the route.

Make sure that the following route is displayed:

```
{
  "name": "50-idtoken2",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/id_token')}",
  "monitor": false,
  "heap": [
    {
      "name": "AmService-1",
```

```
          "type": "AmService",
          "config": {
            "url": "http://openam.example.com:8088/openam",
            "realm": "/",
            "ssoTokenHeader": "iPlanetDirectoryPro",
            "version": "6.0"
          }
        }
    ],
    "handler": {
      "type": "Chain",
      "config": {
        "filters": [
          {
            "name": "OAuth2ClientFilter-1",
            "type": "OAuth2ClientFilter",
            "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",
                    "clientSecret": "password",
                    "issuer": {
                      "name": "Issuer",
                      "type": "Issuer",
                      "config": {
                        "wellKnownEndpoint": "http://openam.example.com:8088/openam/oauth2/.well-known/
openid-configuration"
                      }
                    },
                    "scopes": [
                      "openid",
                      "profile",
                      "email"
                    ],
                    "tokenEndpointAuthMethod": "client_secret_basic"
                  }
                }
              ],
              "requireHttps": false,
              "cacheExpiration": "disabled"
            }
          },
          {
```

```
        "name": "TokenTransformationFilter-1",
        "type": "TokenTransformationFilter",
        "config": {
          "username": "oidc_client",
          "password": "password",
          "idToken": "${attributes.openid.id_token}",
          "instance": "openig",
          "amService": "AmService-1"
        }
      }
    ],
    "handler": {
      "type": "StaticResponseHandler",
      "config": {
        "entity": "{\"id_token\":\n\"${attributes.openid.id_token}\"} \n\n\n{\"saml_assertions\":\n
\"${contexts.sts.issuedToken}\"}",
        "reason": "Found",
        "status": 200
      }
    }
  }
}
```

Note

If necessary, change the value of `version` for AmService to your version of AM.

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 OAuth2ClientFilter 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 " Setting Up AM for OpenID Connect ",
  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 `OAuth2ClientFilter`.

  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}`.

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.

To Test the Configuration

1. http://openig.example.com:8080/home/id_token.

   The AM login screen is displayed.

2. Log in to AM with the username `george` and password `costanza`.

   An OpenID Connect request to access private information is displayed.

3. Select Allow.

   The id_token and SAML assertions are displayed:

   ```
   {"id_token":
   "eyAidHlwIjogIkpXVCIsICJhbGciOiAiSFMyNTYiIH0.eyAiYXRfaGFzaCI6ICJ . . ."}

   {"saml_assertions":
   <"saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" Version= . . ."}
   ```

Chapter 11
Supporting UMA Resource Servers

This chapter describes the experimental support that IG provides for building a User-Managed Access (UMA) resource server.

IG 5.5 and AM 5.5 add support for the User-Managed Access (UMA) 2.0 Grant for OAuth 2.0 Authorization specifications. The text and examples in this chapter describe UMA 2.0, and are relevant for IG 5.5 and later versions used with AM 5.5 and later versions.

The examples in this chapter do not work for versions of IG or AM below 5.5. Refer to the documentation for those versions.

For information about UMA, see the AM *User-Managed Access (UMA) 2.0 Guide*

11.1. About IG As an UMA Resource Server

This section illustrates the position of IG as a resource server in an UMA environment, with AM as an authorization server.

For more information about the actions that form the UMA workflow, and the process flows for protecting a resource and performing an UMA 2.0 grant, see UMA 2.0 Actors and Actions in the AM *User-Managed Access (UMA) 2.0 Guide*.

UMA Architecture
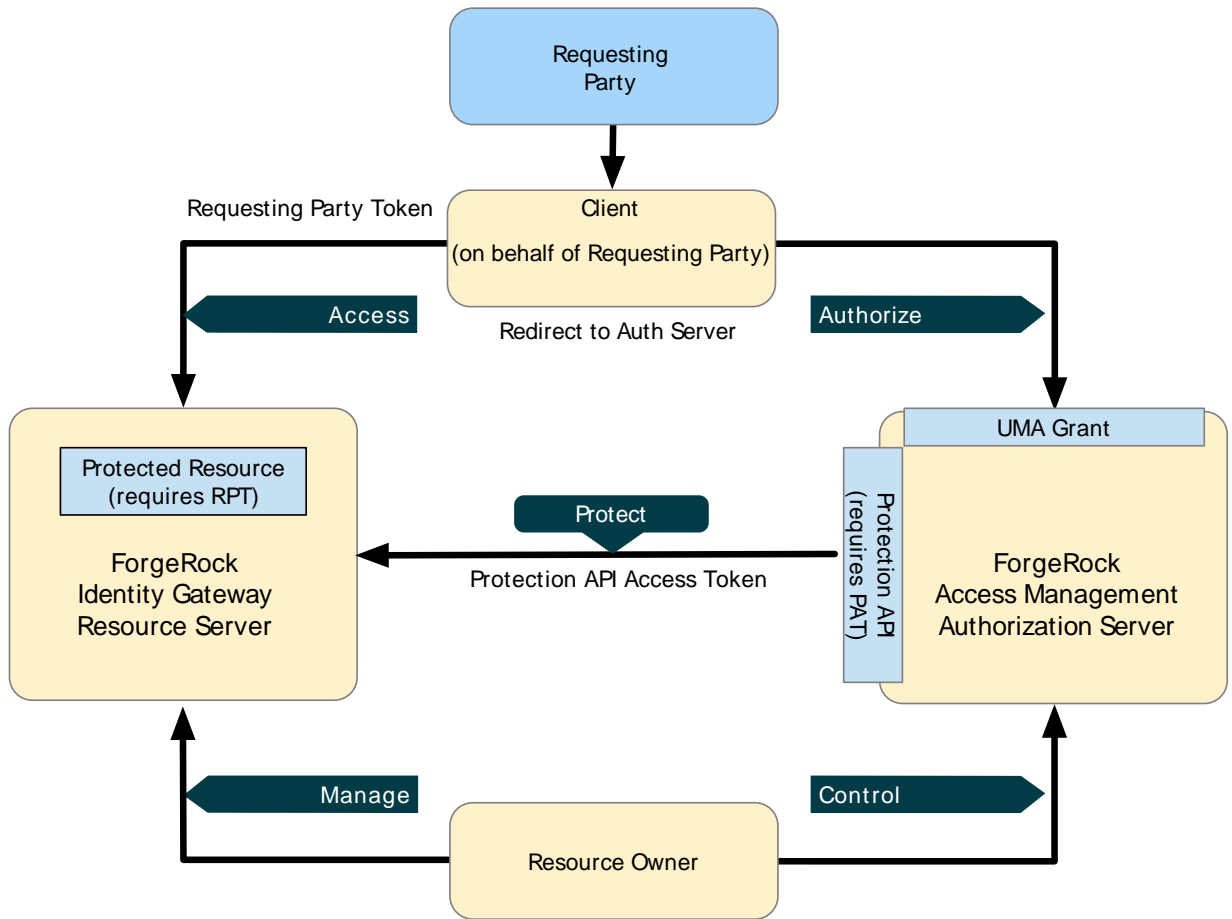
11.2. Sharing and Accessing Protected Resources

This section describes the data flows for registering protected resources and accessing protected resources with a Requesting Party Token (RPT).

For more information about the UMA 2.0 process flow, see UMA 2.0 Process Flow in the AM *User-Managed Access (UMA) 2.0 Guide*.

The following sequence diagram outlines the data flow for a successful registration of a protected resource:

UMA 2.0 Protecting a Resource Flow

UMA 2.0 Flow for Protecting a Resource
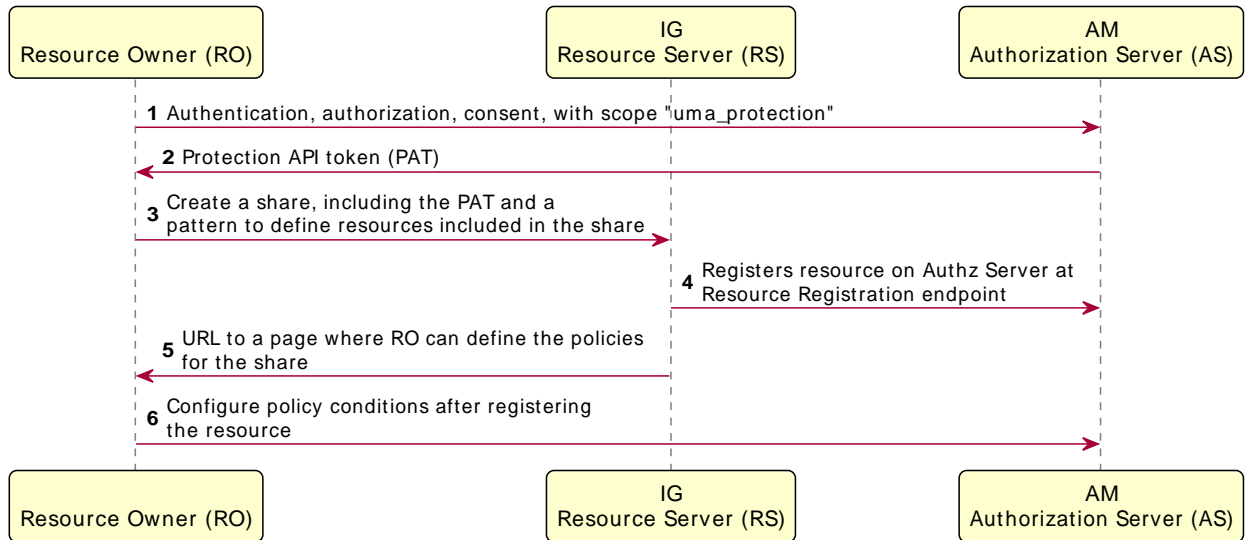
The following sequence diagram outlines the data flow for a successful UMA 2.0 grant flow, where the client accesses the protected resource:

UMA 2.0 Grant Flow Process

UMA 2.0 Grant Flow
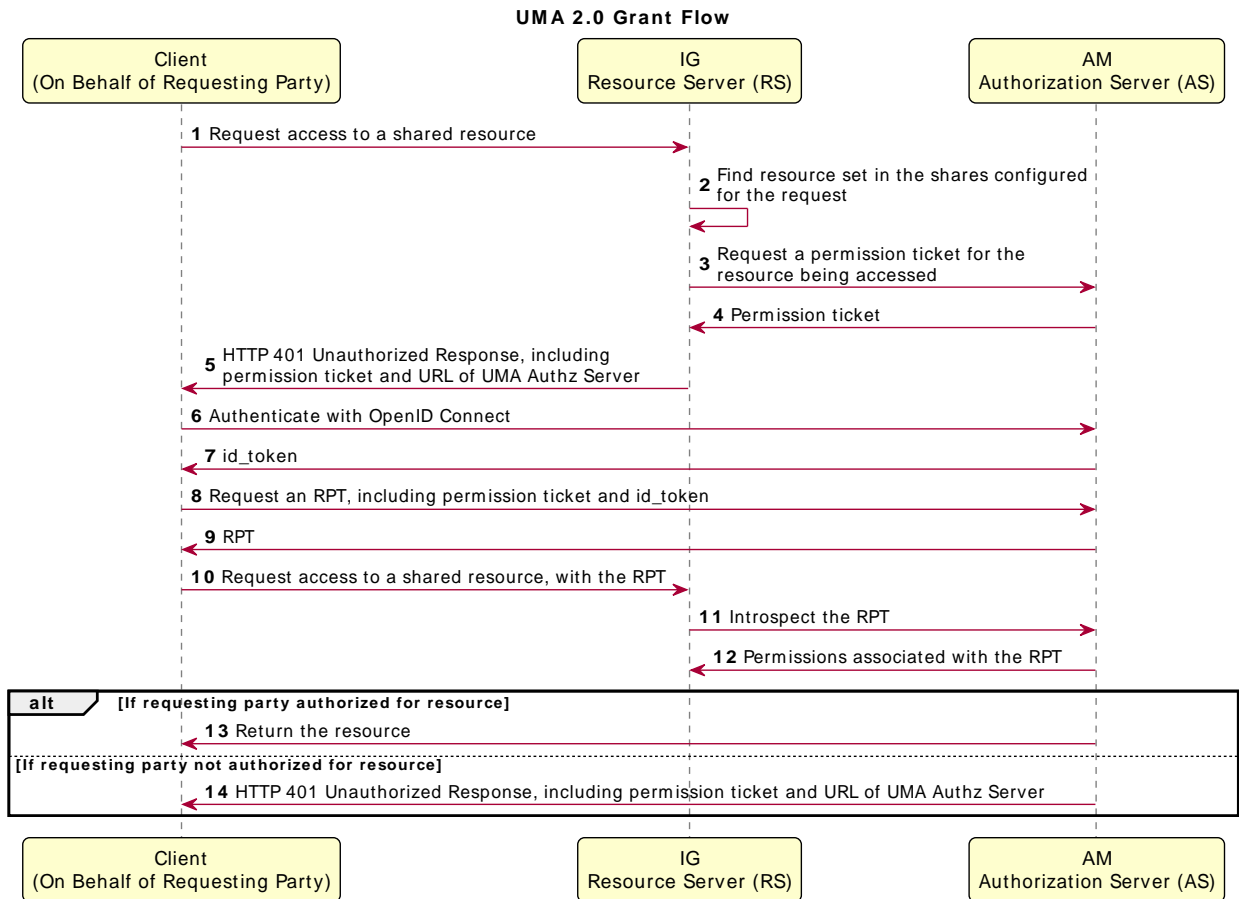
11.3. Limitations of This Implementation

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.

11.4. Preparing the Tutorial

This tutorial describes one way of setting up IG as an UMA resource server. It uses AM as an authorization server for OAuth 2.0 and for UMA, and uses the sample application as a resource to protect and for files that serve as a basic UMA client.

Tasks for Configuring UMA

Task	See Section(s)
Modify the AM configuration to allow cross-site access.	"To Enable CORS Support for AM"
Configure AM as an authorization server.	"To Configure AM As an OAuth 2.0 Authorization Server and UMA Authorization Server"
Register client profiles in AM for OAuth 2.0 and UMA.	"To Register Client Profiles in AM"
Create a subject to act as a resource owner and a subject to act as a requesting party.	"To Create a Resource Owner and Requesting Party"
Set up the IG configuration for an UMA resource server	"To Set Up IG As an UMA Resource Server"
If you use a configuration that is different from that described in this chapter, adjust the sample to your configuration.	"Editing the Example to Match Custom Settings"

11.5. Setting Up AM As an Authorization Server

This section describes the following 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)

Before you start, install and configure AM on http://openam.example.com:8088/openam, with the default configuration. If you use a different configuration, substitute in the tutorial accordingly.

To Enable CORS Support for AM

For information about CORS support, see the AM product documentation. This procedure describes how to modify the AM configuration to allow cross-site access.

> **Caution**
>
> The settings in this section are suggestions for this tutorial, and are not intended as documentation 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 header exposes AM to cross-site request forgery (CSRF) attacks.

1. In the `WEB-INF/web.xml` file of AM, edit the URL pattern for `CORSFilter` to match all endpoints:

    ```
    <filter-mapping>
        <filter-name>CORSFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ```

2. In the same file, edit the `CORSFilter` configuration to authorize cross-site access for origins, hosts, and headers, as shown in the following excerpt:

    ```
    <filter>
        <filter-name>CORSFilter</filter-name>
        <filter-class>org.forgerock.openam.cors.CORSFilter</filter-class>
        <init-param>
            <description>
                Accepted Methods (Required):
                A comma separated list of HTTP methods for which to accept CORS requests.
            </description>
            <param-name>methods</param-name>
            <param-value>POST,GET,PUT,DELETE,PATCH,OPTIONS</param-value>
        </init-param>
        <init-param>
            <description>
                Accepted Origins (Required):
                A comma separated list of origins from which to accept CORS requests.
            </description>
            <param-name>origins</param-name>
            <param-value>http://app.example.com:8081,http://openig.example.com:8080,http://
    openam.example.com:8088</param-value>
        </init-param>
    ```

```
    <init-param>
        <description>
            Allow Credentials (Optional):
            Whether to include the Vary (Origin)
            and Access-Control-Allow-Credentials headers in the response.
            Default: false
        </description>
        <param-name>allowCredentials</param-name>
        <param-value>true</param-value>
    </init-param>
    <init-param>
        <description>
            Allowed Headers (Optional):
            A comma separated list of HTTP headers
            which can be included in the requests.
        </description>
        <param-name>headers</param-name>
        <param-value>
            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
        </param-value>
    </init-param>
    <init-param>
        <description>
            Expected Hostname (Optional):
            The name of the host expected in the request Host header.
        </description>
        <param-name>expectedHostname</param-name>
        <param-value>openam.example.com:8088</param-value>
    </init-param>
    <init-param>
        <description>
            Exposed Headers (Optional):
            The comma separated list of headers
            which the user-agent can expose to its CORS client.
        </description>
        <param-name>exposeHeaders</param-name>
        <param-value>Access-Control-Allow-Origin,Access-Control-Allow-Credentials,Set-Cookie,WWW-
Authenticate</param-value>
    </init-param>
    <init-param>
        <description>
            Maximum Cache Age (Optional):
            The maximum time that the CORS client can cache
            the pre-flight response, in seconds.
            Default: 600
        </description>
        <param-name>maxAge</param-name>
        <param-value>600</param-value>
    </init-param>
</filter>
```

To Configure AM As an OAuth 2.0 Authorization Server and UMA Authorization Server

1. Log in to the AM console as administrator.

2. In the top level realm, select Configure OAuth Provider > Configure OAuth 2.0, accept the default values, and select Create.

   The AM service `OAuth2 Provider` is created for the authorization endpoint.

   The PAT is obtained through the OAuth 2.0 access token endpoint. The RPT is obtained through the UMA endpoint.

   If you plan to build your own examples or modify the sample clients, consider extending the default token lifetimes.

3. Select Configure OAuth Provider > Configure User Managed Access, accept the default values, and select Create.

   The AM service `UMA Provider` is created.

To Register Client Profiles in AM

Follow these steps to register client profiles for OAuth 2.0 and UMA:

1. In the top level realm, select Applications > OAuth 2.0.

2. Add a client with the following values:

   - Client ID: `OpenIG`

   - Client secret: `password`

   - Scope: `uma_protection`

3. Add a client to use when accessing resources, with the following values:

   - Client ID: `UmaClient`

   - Client secret: `password`

   - Scope: `openid`

To Create a Resource Owner and Requesting Party

Follow these steps to create identities in the top level realm:

1. In the top level realm, select Identities.

2. Add a new identity to act as a resource owner, with the following values:

   - ID: `alice`

   - Password: `password`

3. Add a new identity to act as a requesting party, with the following values:

- ID: bob

- Password: password

11.6. Setting Up IG As an UMA Resource Server

To Set Up IG As an UMA Resource Server

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*.

1. Add the following file as $HOME/.openig/config/admin.json.

   On Windows, add the route as %appdata%\OpenIG\config\admin.json:

   ```
   {
     "heap": [
       {
         "name": "ClientHandler",
         "type": "ClientHandler"
       },
       {
         "name": "ApiProtectionFilter",
         "type": "ScriptableFilter",
         "config": {
           "type": "application/x-groovy",
           "file": "CorsFilter.groovy"
         }
       }
     ],
     "prefix": "openig"
   }
   ```

   To allow CORS support for the UMA share API, this route overrides the default ApiProtectionFilter that protects the reserved administrative route. By default, the administrative route is under /openig. For information, see AdminHttpApplication(5) in the *Configuration Reference*.

2. Add the following route to the IG configuration as $HOME/.openig/config/routes/00-uma.json.

   On Windows, add the route as %appdata%\OpenIG\config\routes\00-uma.json.

   ```
   {
     "heap": [
       {
         "name": "UmaService",
         "type": "UmaService",
         "config": {
           "protectionApiHandler": "ClientHandler",
   ```

```
          "wellKnownEndpoint": "http://openam.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": "ScriptableFilter",
            "config": {
              "type": "application/x-groovy",
              "file": "CorsFilter.groovy"
            }
          },
          {
            "type": "UmaFilter",
            "config": {
              "protectionApiHandler": "ClientHandler",
              "umaService": "UmaService"
            }
          }
        ],
        "handler": "ClientHandler"
      }
    },
    "condition": "${request.uri.host == 'app.example.com'}"
}
```

Notice the following features of the route:

- The `UmaService` describes the resources that a resource owner can share, using AM as the authorization server.

  The `UmaService` provides a REST API to manage sharing of resource sets.

- The handler for the route chains together the CORS filter, the `UmaFilter`, and the default handler.

  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.

- The route matches requests to `app.example.com`.

3. Add the following script to the IG configuration as `$HOME/.openig/scripts/groovy/CorsFilter.groovy`.

   On Windows, add the script as `%appdata%\OpenIG\scripts\groovy\CorsFilter.groovy`.

```groovy
import org.forgerock.http.protocol.Response
import org.forgerock.http.protocol.Status

if (request.method == 'OPTIONS') {
    /**
     * Supplies a response to a CORS preflight request.
     *
     * Example response:
     *
     * HTTP/1.1 200 OK
     * Access-Control-Allow-Origin: http://app.example.com:8081
     * Access-Control-Allow-Methods: POST
     * Access-Control-Allow-Headers: Authorization
     * Access-Control-Allow-Credentials: true
     * Access-Control-Max-Age: 3600
     */

    def origin = request.headers['Origin']?.firstValue
    def response = new Response(Status.OK)

    // Browsers sending a cross-origin request from a file might have Origin: null.
    response.headers.put("Access-Control-Allow-Origin", origin)
    request.headers['Access-Control-Request-Method']?.values.each() {
        response.headers.add("Access-Control-Allow-Methods", it)
    }
    request.headers['Access-Control-Request-Headers']?.values.each() {
        response.headers.add("Access-Control-Allow-Headers", it)
    }
    response.headers.put("Access-Control-Allow-Credentials", "true")
    response.headers.put("Access-Control-Max-Age", "3600")

    return response
}

return next.handle(context, request)
/**
 * Adds headers to a CORS response.
 */
        .thenOnResult({ response ->
    if (response.status.isServerError()) {
        // Skip headers if the response is a server error.
    } else {
```

```
        def headers = [
                "Access-Control-Allow-Origin": request.headers['Origin']?.firstValue,
                "Access-Control-Allow-Credentials": "true",
                "Access-Control-Expose-Headers": "WWW-Authenticate"
        ]
        response.headers.addAll(headers)
    }
})
```

This script adds a CORS filter to include headers for cross-origin requests.

The tutorial involves JavaScript clients that are served by the sample application, and so not from the same origin as AM or IG. The route uses a CORS filter to include appropriate response headers for cross-origin requests.

The CORS filter handles pre-flight (HTTP OPTIONS) requests, and responses for all HTTP operations. The logic for the filter is provided through the script.

The filter adds the appropriate headers to CORS requests. Pre-flight requests are diverted to a dedicated handler, which returns the response directly to the user agent. For all other requests, the headers are added to the response.

For information about scripting filters and handlers, see "*Extending Identity Gateway*".

4. Restart IG to reload the configuration.

11.7. Test the Configuration

Follow these steps to test the configuration and demonstrate IG acting as an UMA resource server:

1. Log out of AM.

2. Browse to http://app.example.com:8081/uma/.

   If you used the settings described in this chapter and "Installing the Sample Application" in the *Getting Started Guide*, your configuration should match the displayed configuration. If you used other settings, you might need to edit the sample files to match your settings. For information, see "Editing the Example to Match Custom Settings".

3. Select the link to demonstrate Alice sharing resources.

4. On Alice's page, select Share with Bob to simulate Alice sharing resources as described in "Sharing and Accessing Protected Resources".

   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, get help in "Troubleshooting the UMA Example".

5. Go back to the first page, and select the link to demonstrate Bob accessing resources.

6. On Bob's page, select Get Alice's resources to simulate Bob accessing one of 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

11.8. Editing 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 "Installing the Sample Application" in the *Getting Started Guide* to temporary folder:

```
$ mkdir /tmp/uma
$ cd /tmp/uma
$ jar -xvf /path/to/IG-sample-application-6.1.0.jar uma
 created: uma/
inflated: uma/alice.html
inflated: uma/bob.html
inflated: uma/common.js
inflated: uma/index.html
inflated: 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-6.1.0.jar uma
adding: uma/(in = 0) (out= 0)(stored 0%)
adding: uma/index.html(in = 1698) (out= 880)(deflated 48%)
adding: uma/common.js(in = 4265) (out= 1319)(deflated 69%)
adding: uma/bob.html(in = 5427) (out= 1811)(deflated 66%)
adding: uma/style.css(in = 1403) (out= 696)(deflated 50%)
adding: uma/alice.html(in = 5494) (out= 1762)(deflated 67%)
```

4. If necessary, adjust the CORS settings for AM.

11.9. Understanding 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, see "Understanding IG APIs With API Descriptors".

Chapter 12
Configuring Routers and Routes

IG provides routers and routes to handle requests and their context. In this chapter, you will learn about:

• How routers and routes are configured

• How to read, create, edit, or delete routes through Common REST or Studio

• How to prevent changes to routes when IG is running

12.1. Configuring Routers

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
      }
    }
  ]
}
```

In this configuration, all requests are passed 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 are used to build the monitoring endpoints. For information about monitoring endpoints, see *"Auditing"*. For information about the parameters of a router, see Router(5) in the *Configuration Reference*.

12.2. Configuring Routes

Routes are JSON configuration files that handle requests and their context, and then hand off any request they accept to a handler. Another way to think of a route is like an independent dispatch handler, as described in DispatchHandler(5) in the *Configuration Reference*.

Routes can have a base URI to change the scheme, host, and port of the request.

For information about the parameters of routes, see Route(5) in the *Configuration Reference*.

12.2.1. Configuring Objects Inline or In the Heap

If you use an object only once in the configuration, you can declare it inline in the route and do not need to name it. However, when you need use an object multiple times, declare it in the heap, and then reference it by name in the route.

The following route shows an inline declaration for a handler. The handler is a router to route requests to separate route configurations:

```
{
    "handler": {
        "type": "Router"
    }
}
```

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

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

Notice that the heap takes an array. Because the heap holds all configuration objects at the same level, you can impose any hierarchy or order when referencing objects. Note that when you declare all objects in the heap and reference them by name, neither hierarchy nor ordering are obvious from the structure of the configuration file alone.

12.2.2. Setting Route Conditions

When a route has a condition, it can handle only requests that meet the condition. When a route has no condition, it can handle any request.

A condition can be based on almost any characteristic of the request, context, or IG runtime environment. Conditions are defined using IG expressions, as described in Expressions(5) in the *Configuration Reference*.

Because routes define the conditions on which they accept a request, the router does not have to know about specific routes in advance. In other words, you can configure the router first and then add routes while IG is running.

The following example shows a route with no condition. This route accepts any request:

```
{
  "name": "myroute",
  "handler": {
    "type": "ReverseProxyHandler"
  }
}
```

The following example shows the same route with a condition. This route accepts only requests whose path starts with `mycondition`:

```
{
  "name": "myroute",
  "handler": {
    "type": "ReverseProxyHandler"
  },
  "condition": "${matches(request.uri.path, '^/mycondition')}"
}
```

The following table lists some of the conditions used in routes in this guide:

Route Conditions

Route Condition	Some Example Requests That Meet the Condition
`"${matches(request.uri.path, '^/login')}"`	http://app.example.com/login, . . .
`"${request.uri.host == 'api.example.com'}"`	http://api.example.com/, https://api.example.com/home, http://api.example.com:8080/home, . . .
`"${matches(contexts.client.remoteAddress, '127.0.0.1')}"`	http://localhost:8080/keygen, http://127.0.0.1:8080/keygen, . . . Where /keygen is not mandatory and could be anything else.
`"${matches(request.uri.query, 'demo=simple')}"`	http://openig.example.com:8080/login?demo=simple, . . . For information about URI query, see query in URI(5) in the *Configuration Reference*.

Route Condition	Some Example Requests That Meet the Condition
`"${request.uri.scheme == 'http'}"`	`http://openig.example.com:8080`, ...
`"${matches(request.uri.path, '^/dispatch') or matches(request.uri.path, '^/mylogin')}"`	`http://openig.example.com:8080/dispatch`, `http://openig.example.com:8080/mylogin`, ...
`"${request.uri.host == 'sp1.example.com' and not matches(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`, ...

12.2.3. Configuring Route Names, IDs, and Filenames

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

The Router scans the routes folder for files with the `.json` extension, and uses the route's `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`, and the route's `name` property and route ID cannot be `default`.

12.3. Creating and Editing Routes Through Common REST

> **Note**
>
> When IG is in production mode, you cannot manage, list, or even read routes through Common REST. For more information, see "Making the Configuration Immutable".

> **Note**
>
> If an AM policy agent is configured in the same container as IG, by default the policy agent intercepts requests to manage routes. When you try to add a route through Common REST, the policy agent redirects the request to AM and the route is not added.

To override this behavior, add the URL pattern `/openig/api/*` to the list of not-enforced URI in the policy agent profile. For more information about configuring policy agents in AM, see the *Java Agents Guide* .

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 are 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, see "About ForgeRock Common REST" in the *Configuration Reference*.

To Manage Routes Through Common REST

Before you start, prepare IG as described in "*First Steps*" in the *Getting Started Guide*.

1. Add the following route to the IG configuration as `$HOME/.openig/config/routes/00-crest.json`.

   On Windows, add the route as `%appdata%\OpenIG\config\routes\00-crest.json`.

   ```
   {
     "name": "crest",
     "handler": {
       "type": "StaticResponseHandler",
       "config": {
         "status": 200,
         "reason": "OK",
         "entity": "Hello, world!"
       }
     },
     "condition": "${matches(request.uri.path, '^/crest')}"
   }
   ```

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

2. To read a route through Common REST:

   • Enter the following command in a terminal window:

   ```
   $ http GET http://openig.example.com:8080/openig/api/system/objects/_router/routes/00-crest
   ```

   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:

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

   • Check in `$HOME/.openig/logs/route-system.log` that the route has been removed from the configuration. To double check, access http://openig.example.com:8080/crest. You should get an HTTP 404.

   • Enter the following command in a terminal window:

```
$ http PUT http://openig.example.com:8080/openig/api/system/objects/_router/routes/00-crest < /tmp
/00-crest.json
```

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

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

4. To edit a route through Common REST:

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

- Enter the following command in a terminal window:

```
$ http PUT http://openig.example.com:8080/openig/api/system/objects/_router/routes/00-crest If-
Match:* < /tmp/00-crest.json
```

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`.

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

5. To delete a route through Common REST:

- Enter the following command in a terminal window:

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

- Check in `$HOME/.openig/logs/route-system.log` that the route has been removed from the configuration. To double-check, access http://openig.example.com:8080/crest. You should get an HTTP 404.

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

- Enter the following command in a terminal window:

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

The list of loaded routes is displayed.

12.4. Creating Routes Through Studio

> **Note**
>
> When IG is in production mode, Studio is effectively disabled. For more information, see "Making the Configuration Immutable".

Studio is a user interface to configure and deploy routes in IG. You can use Studio to create routes for tasks such as authenticating users, and authorizing access to APIs, throttling the rate of requests to protected applications, capturing messages, and collecting statistics. New features will be added as Studio evolves.

When IG is installed and running as described in this guide, access Studio on http://openig.example.com:8080/openig/studio.

For help to get started with Studio, see "*Configuring Routes With Studio*" in the *Getting Started Guide*. For examples of how to use Studio to configure routes, see the following sections of this guide:

- To configure IG to enforce AM policy decisions, see "To Set Up IG as a PEP".

- To configure IG as a resource server using the token info endpoint, see "To Set Up IG As a Resource Server Using the Token Info Endpoint".

- To configure IG as a relying party for OpenID Connect 1.0, see "To Set Up IG As a Relying Party".

- To configure a simple throttling filter, see "To Configure a Simple Throttling Filter".

12.5. Preventing the 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"
    }
}
```

12.6. Accessing Reserved Routes

IG uses an `ApiProtectionFilter` to protect the reserved routes. By default, the filter 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, see the CORS filter described in "To Set Up IG As an UMA Resource Server".

Chapter 13
Proxying WebSocket Traffic

This chapter describes how IG proxies WebSocket traffic.

When a user agent requests an upgrade from HTTP 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.

> **Important**
>
> IG cannot proxy WebSocket traffic for TLS connections, or when it is running in the Jetty application container.

The following sequence diagram shows the flow of information when IG proxies WebSocket traffic:

Flow of Information to Proxy WebSocket Traffic
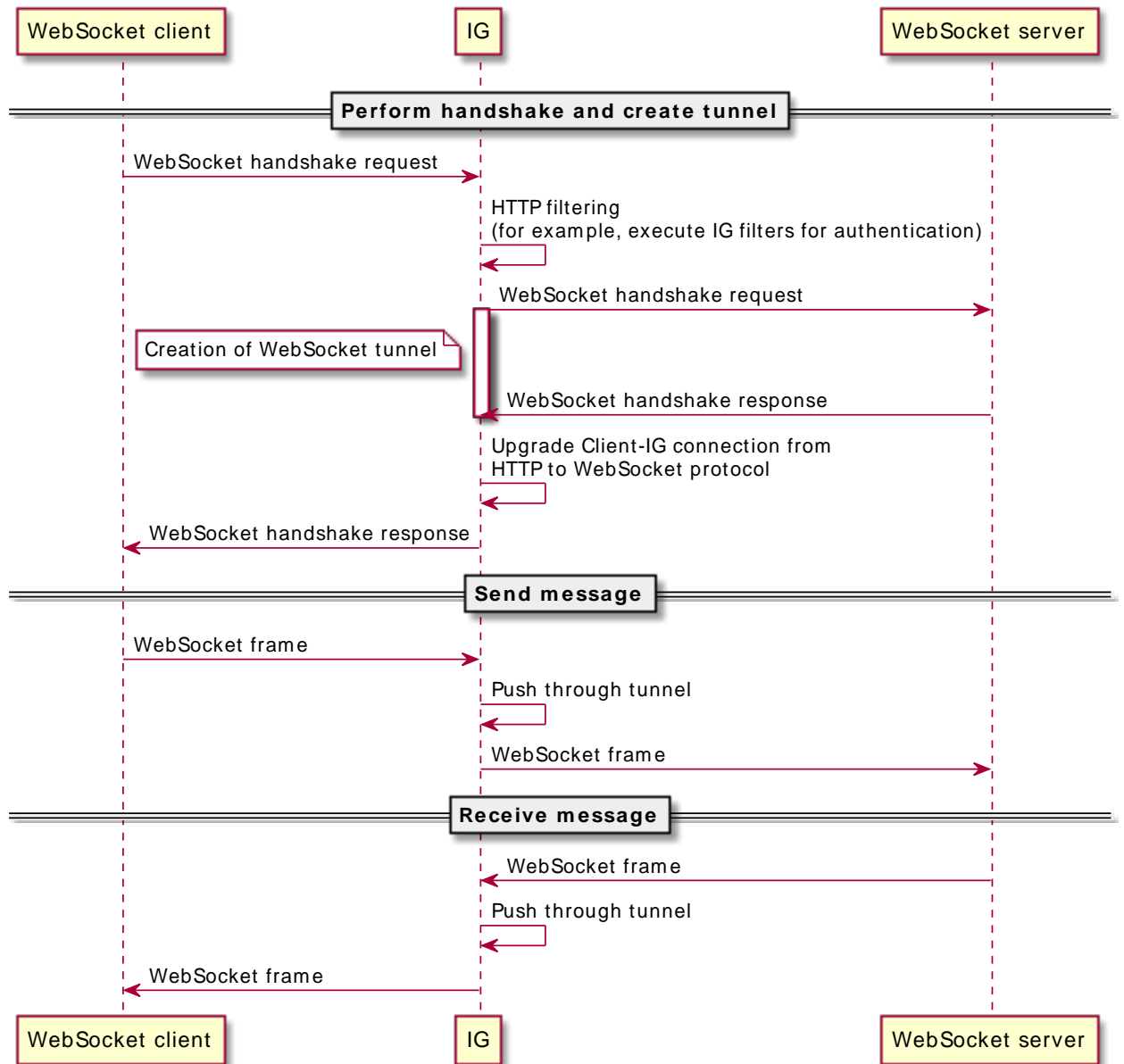
To set up IG to proxy WebSocket traffic, configure the `websocket` property of ClientHandler or ReverseProxyHandler. By default, IG does not proxy WebSocket traffic. For more information, see ClientHandler(5) in the *Configuration Reference* and ReverseProxyHandler(5) in the *Configuration Reference*.

The following procedures give an example of how to set up and test proxying for WebSocket traffic. The example uses the WebSocket server provided in the sample app, and uses the SingleSignOnFilter for authentication.

To Configure IG to Proxy WebSocket Traffic

Before you start:

- Prepare IG and the sample app as described in "*First Steps*" in the *Getting Started Guide*

- Install and configure AM on http://openam.example.com:8088/openam, using the default configuration. If you use a different configuration, substitute in the tutorial accordingly.

1. Add the following route to the IG configuration as `$HOME/.openig/config/routes/websocket.json` (on Windows, `%appdata%\OpenIG\config\routes\websocket.json`:

```
{
  "name" : "websocket",
  "baseURI" : "http://app.example.com:8081",
  "condition" : "${matches(request.uri.path, '^/websocket')}",
  "monitor" : false,
  "heap" : [ {
    "name" : "AmService-1",
    "type" : "AmService",
    "config" : {
      "url" : "http://openam.example.com:8088/openam",
      "realm" : "/",
      "ssoTokenHeader" : "iPlanetDirectoryPro",
      "version": "6.0"
    }
  }],
  "handler" : {
    "type" : "Chain",
    "config" : {
      "filters" : [ {
        "name" : "SingleSignOnFilter-1",
        "type" : "SingleSignOnFilter",
        "config" : {
          "amService" : "AmService-1"
        }
      }
      ],
      "handler" : {
        "type": "ReverseProxyHandler",
        "config": {
          "websocket": {
            "enabled": true
          }
        }
      }
    }
```

```
        }
      }
}
```

2. 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, and, after IG upgrades the HTTP connection to the WebSocket protocol, passes the messages to the WebSocket server.

To Test the Setup

Before you start:

• Prepare IG to run on Tomcat or JBoss EAP, as described in "Configuring Deployment Containers".

• Prepare the sample app as described in "*First Steps*" in the *Getting Started Guide*.

• Install and configure AM on http://openam.example.com:8088/openam, using the default configuration.

1. If necessary log out of AM, and then browse to http://openig.example.com:8080/websocket.

   The SingleSignOnFilter redirects you to AM for authentication.

2. Log in to AM as user `demo`, password `changeit`.

   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.

Chapter 14
Configuration Templates

This chapter contains template routes for common configurations.

Before you use these templates, install and configure IG with a router and default route as described in "*First Steps*" in the *Getting Started Guide*.

Next, take one of the templates and then modify it to suit your deployment. Read the summary of each template to find the right match for your application.

When you move to use IG in production, be sure to turn off DEBUG level logging, and to deactivate `CaptureDecorator` use to avoid filling up disk space. Also consider locking down the `Router` configuration.

14.1. Proxy and Capture

If you installed and configured IG with a router and default route as described in "*First Steps*" in the *Getting Started Guide*, 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:

Proxy and Capture

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert for HTTPS.",
      "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": "${matches(request.uri.query, 'demo=capture')}"
}
```

To try this example with the sample application:

1. Save the file as `$HOME/.openig/config/routes/20-capture.json`.

2. Browse to http://openig.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 ClientHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ClientHandler that references a truststore holding the certificate.

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

    In production, do not use `TrustAllManager` for TrustManager, or `ALLOW_ALL` for hostname verification. For information, see ClientHandler(5) in the *Configuration Reference*.

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.

14.2. 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:

Simple Login Form

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

To try this example with the sample application:

1. Save the file as `$HOME/.openig/config/routes/21-simple.json`.

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

3. Browse to http://openig.example.com:8080/login?demo=simple.

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

```
Method POST
URI /login
Cookies
Headers content-type: application/x-www-form-urlencoded
content-length: 31
host: app.example.com:8445
connection: Keep-Alive
user-agent: Apache-HttpAsyncClient/4.1.2 (Java/1.8.0_144)
```

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

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

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

   In production, do not use `TrustAllManager` for TrustManager, or `ALLOW_ALL` for hostname verification. For information, see ClientHandler(5) in the *Configuration Reference*.

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.

14.3. 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(5) in the *Configuration Reference*.

Login Form With Cookie From Login Page

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert for HTTPS.",
      "config": {
        "trustManager": {
          "type": "TrustAllManager"
        },
        "hostnameVerifier": "ALLOW_ALL"
      }
    }
  ],
```

```
      "handler": {
        "type": "Chain",
        "config": {
          "filters": [
            {
              "type": "PasswordReplayFilter",
              "config": {
                "loginPage": "${request.uri.path == '/login'}",
                "request": {
                  "method": "POST",
                  "uri": "https://app.example.com:8444/login",
                  "form": {
                    "username": [
                      "MY_USERNAME"
                    ],
                    "password": [
                      "MY_PASSWORD"
                    ]
                  }
                }
              }
            },
            {
              "type": "CookieFilter"
            }
          ],
          "handler": "ReverseProxyHandler"
        }
      },
      "condition": "${matches(request.uri.query, 'demo=cookie')}"
}
```

To try this example with the sample application:

1. Save the file as `$HOME/.openig/config/routes/22-cookie.json`.

2. Replace `MY_USERNAME` with `kramer`, and `MY_PASSWORD` with `newman`.

3. Browse to http://openig.example.com:8080/login?demo=cookie.

   The sample application profile page for the user `kramer` is displayed, as well as information about the session cookie.

   The sample application profile page for the user `kramer` displays the following information about the request:

```
Method POST
URI /login
Cookies session-cookie=5667511728176205275
Headers content-type: application/x-www-form-urlencoded
cookie: session-cookie=5667511728176205275
content-length: 31
host: app.example.com:8445
connection: Keep-Alive
user-agent: Apache-HttpAsyncClient/4.1.2 (Java/1.8.0_144)
```

Note that compared to the example in "Login Form With Cookie From Login Page", this example displays additional information about the session cookie

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

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

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

   In production, do not use `TrustAllManager` for TrustManager, or `ALLOW_ALL` for hostname verification. For information, see ClientHandler(5) in the *Configuration Reference*.

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.

14.4. 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.

The CookieFilter removes cookies from the response, except for the session cookie added by the container.

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(5) in the *Configuration Reference*.

Login Form With Password Replay and Cookie Filters

```
{
  "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": [
                  "changeit"
                ],
                "IDButton": [
                  "Log+In"
                ],
                "encoded": [
                  "false"
                ]
              },
              "headers": {
                "host": [
                  "app.example.com:8081"
                ]
              }
            }
          }
        },
        {
          "type": "CookieFilter"
        }
      ],
      "handler": "ReverseProxyHandler"
    }
  },
  "condition": "${matches(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://openig.example.com:8080/login?demo=classic
         HTTP/1.1 200 OK
   Server: Apache-Coyote/1.1
   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.

14.5. 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.

Login Which Requires a Hidden Value From the Login Page

```
{
  "properties": {
    "appBaseUri":  "https://app.example.com:8444"
  },
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert for HTTPS.",
      "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": "${matches(request.uri.query, 'demo=hidden')}",
"baseURI": "${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. Save the file as `$HOME/.openig/config/routes/24-hidden.json`.

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

3. Browse to http://openig.example.com:8080/login?demo=hidden.

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

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

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

    In production, do not use `TrustAllManager` for TrustManager, or `ALLOW_ALL` for hostname verification. For information, see ClientHandler(5) in the *Configuration Reference*.

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.

14.6. 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.

HTTP and HTTPS Application

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
```

```
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert for HTTPS.",
      "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": "${matches(request.uri.query, 'demo=https')}"
}
```

To try this example with the sample application:

1. Save the file as $HOME/.openig/config/routes/25-https.json.

2. Browse to openig.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 ClientHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ClientHandler that references a truststore holding the certificate.

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

In production, do not use `TrustAllManager` for TrustManager, or `ALLOW_ALL` for hostname verification. For information, see ClientHandler(5) in the *Configuration Reference*.

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.

14.7. 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.

AM Integration With Headers

```
{
  "heap": [
    {
      "name": "ReverseProxyHandler",
      "type": "ReverseProxyHandler",
      "comment": "Testing only: blindly trust the server cert for HTTPS.",
      "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": "${matches(request.uri.query, 'demo=headers')}"
}
```

To try this example with the sample application:

1. Save the file as `$HOME/.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: bribery" \
  http://openig.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 ClientHandler with one that is configured to trust the application's public key server certificate. Otherwise, use a ClientHandler that references a truststore holding the certificate.

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

   In production, do not use `TrustAllManager` for TrustManager, or `ALLOW_ALL` for hostname verification. For information, see ClientHandler(5) in the *Configuration Reference*.

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.

Chapter 15
Extending Identity Gateway

This chapter describes how to extend IG, taking you through the steps to:

• Write scripts to create custom filters and handlers

• Plug additional Java libraries into IG for further customization

For when you can't achieve complex server interactions or intensive data transformations with scripts or existing handlers, filters, or expressions, IG allows you to develop custom extensions in Java and provide them in additional libraries that you build into IG. The libraries allow you to develop custom extensions to IG.

> **Important**
>
> When you are writing scripts or Java extensions, never use a `Promise` blocking method, such as `get()`, `getOrThrow()`, or `getOrThrowUninterruptibly()`, to obtain the response.
>
> 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.

15.1. About Scripting

IG supports the Groovy dynamic scripting language through the use of the following scriptable objects:

• ScriptableFilter

• ScriptableHandler

• ScriptableThrottlingPolicy

• ScriptableAccessTokenResolver

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 or to an LDAP directory service, and access responses returned in promise callback methods.

Before trying the scripts in this chapter, install and configure IG as described in "*First Steps*" in the *Getting Started Guide*.

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 details, see CaptureDecorator(5) in the *Configuration Reference*.

For more information, see ScriptableFilter(5) in the *Configuration Reference*, ScriptableHandler(5) in the *Configuration Reference*, and ScriptableThrottlingPolicy(5) in the *Configuration Reference*.

15.1.1. Using a Reference File Script

The following example defines a ScriptableFilter written in Groovy, and stored in a file named `$HOME/.openig/scripts/groovy/SimpleFormLogin.groovy` (`%appdata%\OpenIG\scripts\groovy\SimpleFormLogin.groovy` on Windows):

```
{
  "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`.

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

15.1.2. Scripting in Studio

You can configure a ScriptableFilter or ScriptableThrottlingPolicy in Studio. 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 in a ScriptableFilter or ScriptableThrottlingPolicy, the script is added to the list of reference scripts.

- You can use a reference script in multiple filters in a route. If you edit a reference script, all filters that use it are updated with the change.

- If you delete a ScriptableFilter or ScriptableThrottlingPolicy, or remove it from the chain, the script that it references remains in the list of scripts.

- If a reference script is used in a filter, you can't rename or delete the script.

For an example of creating a ScriptableThrottlingPolicy in Studio, see " Configuring a Scriptable Throttling Filter ". For information about using Studio, see "Adding Filters to a Route" in the *Getting Started Guide*.

15.2. Scripting Dispatch

In order to route requests, especially when the conditions are complicated, you can use a `ScriptableHandler` instead of a `DispatchHandler` as described in DispatchHandler(5) in the *Configuration Reference*.

The following script demonstrates a simple dispatch handler:

```
/*
 * Copyright 2014-2017 ForgeRock AS. All Rights Reserved
 *
 * Use of this code requires a commercial software license with ForgeRock AS.
 * or with one of its affiliates. All use shall be exclusively subject
 * to such license between the licensee and ForgeRock AS.
 */


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

// Rather than return 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] == "hifalutin") {

            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
```

To try this handler, save the script as `$HOME/.openig/scripts/groovy/DispatchHandler.groovy` (`%appdata%\OpenIG\scripts\groovy\DispatchHandler.groovy` on Windows).

Next, add the following route to your configuration as `$HOME/.openig/config/routes/98-dispatch.json` (`%appdata%\OpenIG\config\routes\98-dispatch.json` on Windows):

```
{
  "heap": [
    {
      "name": "DispatchHandler",
      "type": "DispatchHandler",
      "config": {
        "bindings": [{
          "condition": "${matches(request.uri.path, '/mylogin')}",
          "handler": {
            "type": "Chain",
            "config": {
              "filters": [
                {
                  "type": "HeaderFilter",
                  "config": {
                    "messageType": "REQUEST",
                    "add": {
                      "Username": [
                        "bjensen"
                      ],
                      "Password": [
                        "hifalutin"
                      ]
                    }
                  }
                }
              ],
              "handler": "Dispatcher"
            }
          }
        },
        {
          "handler": "Dispatcher",
          "condition": "${matches(request.uri.path, '/dispatch')}"
        }
        ]
      }
    },
    {
      "name": "Dispatcher",
      "type": "ScriptableHandler",
```

```
      "config": {
        "type": "application/x-groovy",
        "file": "DispatchHandler.groovy"
      }
    }
  ],
  "handler": "DispatchHandler",
  "condition": "${matches(request.uri.path, '^/dispatch') or matches(request.uri.path, '^/mylogin')}"
}
```

The route sets up the headers required by the script when the user logs in.

To try it out, browse to http://openig.example.com:8080/dispatch.

The response from the script says, "Please log in." When you click the log in link, the `HeaderFilter` sets `Username` and `Password` headers in the request, and passes the request to the script.

The script then responds, `Welcome back, Babs!`

15.3. Scripting HTTP Basic Authentication

HTTP Basic authentication calls for the user agent such as a browser to send a user name and password to the server in an `Authorization` header. HTTP Basic authentication relies on an encrypted connection to protect the user name and password credentials, which are base64-encoded in the `Authorization` header, not encrypted.

The following script, for use in a `ScriptableFilter`, adds an `Authorization` header based on a username and password combination:

```
/*
 * Copyright 2014-2017 ForgeRock AS. All Rights Reserved
 *
 * Use of this code requires a commercial software license with ForgeRock AS.
 * or with one of its affiliates. All use shall be exclusively subject
 * to such license between the licensee and ForgeRock AS.
 */

/*
 * 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": "hifalutin"
 *             }
 *         }
 * }
```

```
 */

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)
```

To try this filter, save the script as `$HOME/.openig/scripts/groovy/BasicAuthFilter.groovy` (`%appdata%\OpenIG\scripts\groovy\BasicAuthFilter.groovy` on Windows).

Next, add the following route to your configuration as `$HOME/.openig/config/routes/09-basic.json` (`%appdata%\OpenIG\config\routes\09-basic.json` on Windows):

```
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ScriptableFilter",
          "config": {
            "type": "application/x-groovy",
            "file": "BasicAuthFilter.groovy",
            "args": {
              "username": "bjensen",
              "password": "hifalutin"
            }
          },
          "capture": "filtered_request"
        }
      ],
      "handler": {
        "type": "StaticResponseHandler",
        "config": {
          "status": 200,
          "reason": "OK",
          "entity": "Hello, Babs!"
        }
      }
```

```
      }
   },
   "condition": "${matches(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.

To try it out, browse to http://openig.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://openig.example.com:8080/basic HTTP/1.1
Authorization: Basic YmplbnNlbjpoaWZhbHV0aW4=
```

15.4. Scripting Authentication to LDAP and LDAPS-enabled Servers

Many organizations use an LDAP directory service to store user profiles including authentication credentials. The LDAP directory service securely stores user passwords in a highly-available, central service capable of handling thousands of authentications per second.

15.4.1. Scripting Authentication to an LDAP Server

The following script, for use in a `ScriptableFilter`, performs simple authentication against an LDAP server, based on request form fields `username` and `password`:

```
/*
 * Copyright 2014-2017 ForgeRock AS. All Rights Reserved
 *
 * Use of this code requires a commercial software license with ForgeRock AS.
 * or with one of its affiliates. All use shall be exclusively subject
 * to such license between the licensee and ForgeRock AS.
 */

import org.forgerock.opendj.ldap.*

/*
 * Perform LDAP authentication based on user credentials from a form.
 *
 * If LDAP authentication succeeds, then return a promise to handle the response.
 * If there is a failure, produce an error response and return it.
 */

username = request.form?.username[0]
password = request.form?.password[0]

// For testing purposes, the LDAP host and port are provided in the context's attributes.
```

```
// Edit as needed to match your directory service.
host = attributes.ldapHost ?: "localhost"
port = attributes.ldapPort ?: 1389

client = ldap.connect(host, port as Integer)
try {

    // Assume the username is an exact match of either
    // the user ID, the email address, or the user's full name.
    filter = "(|(uid=%s)(mail=%s)(cn=%s))"

    user = client.searchSingleEntry(
            "ou=people,dc=example,dc=com",
            ldap.scope.sub,
            ldap.filter(filter, username, username, username))

    client.bind(user.name as String, password?.toCharArray())

    // Authentication succeeded.

    // Set a header (or whatever else you want to do here).
    request.headers.add("Ldap-User-Dn", user.name.toString())

    // Most LDAP attributes are multi-valued.
    // When you read multi-valued attributes, use the parse() method,
    // with an AttributeParser method
    // that specifies the type of object to return.
    attributes.cn = user.cn?.parse().asSetOfString()

    // When you write attribute values, set them directly.
    user.description = "New description set by my script"

    // Here is how you might read a single value of a multi-valued attribute:
    attributes.description = user.description?.parse().asString()

    // Call the next handler. This returns when the request has been handled.
    return next.handle(context, request)

} catch (AuthenticationException e) {

    // LDAP authentication failed, so fail the response with
    // HTTP status code 403 Forbidden.

    response = new Response(Status.FORBIDDEN)
    response.entity = "<html><p>Authentication failed: " + e.message + "</p></html>"

} catch (Exception e) {

    // Something other than authentication failed on the server side,
    // so fail the response with HTTP 500 Internal Server Error.

    response = new Response(Status.INTERNAL_SERVER_ERROR)
    response.entity = "<html><p>Server error: " + e.message + "</p></html>"

} finally {
    client.close()
}

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

```
return response
```

For the list of methods to specify which type of objects to return, see the OpenDJ LDAP SDK Javadoc for `AttributeParser`.

To Authenticate to an LDAP Server

1. Install an LDAP directory server such as ForgeRock Directory Server, and generate or import some sample users who can authenticate over LDAP.

    For information about setting up DS and importing sample data, see To Set Up a Directory Server in the DS *Install Guide*.

2. Save the previous script as `$HOME/.openig/scripts/groovy/LdapAuthFilter.groovy` (`%appdata%\OpenIG\scripts\groovy\LdapAuthFilter.groovy` on Windows).

    If your directory server installation does not match the values in the script, adjust the script to match your installation.

3. Add the following route to your configuration as `$HOME/.openig/config/routes/10-ldap.json`, or `%appdata%\OpenIG\config\routes\10-ldap.json` on Windows:

```
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ScriptableFilter",
          "config": {
            "type": "application/x-groovy",
            "file": "LdapAuthFilter.groovy"
          }
        }
      ],
      "handler": {
        "type": "ScriptableHandler",
        "config": {
          "type": "application/x-groovy",
          "source": [
            "dn = request.headers['Ldap-User-Dn'].values[0]",
            "entity = '<html><body><p>Ldap-User-Dn: ' + dn + '</p></body></html>'",
            "",
            "response = new Response(Status.OK)",
            "response.entity = entity",
            "return response"
          ]
        }
      }
    }
  },
  "condition": "${matches(request.uri.path, '^/ldap')}"
}
```

The route calls the `LdapAuthFilter.groovy` script to authenticate the user over LDAP. On successful authentication, it responds with the the bind DN.

4. Browse to a URL where query string parameters specify a valid username and password.

   If you set up DS and imported the DS sample data as described in step 1, browse to the following link which specifies credentials for the sample user `abarnes`: http://openig.example.com:8080/ldap?username=abarnes&password=chevron.

   The script responds with the DN of the user:

   ```
   Ldap-User-Dn: uid=abarnes,ou=People,dc=example,dc=com
   ```

15.4.2. Scripting Authentication to an LDAPS-enabled Server

To authenticate to LDAPS, you must define the configuration for an SSL context builder, specify the enabled protocols, define a trust manager, and use the LDAPS port.

For convenience, this example uses a special TrustManager to blindly accept all certificates. In a production environment, use a TrustManager that is strictly configured to accept only appropriate certificates.

To test authentication to an LDAPS enabled server, follow the procedure in "Scripting Authentication to an LDAP Server", but save the following script as `LdapAuthFilter.groovy` instead of the script in that procedure:

```
import org.forgerock.opendj.ldap.*
import org.forgerock.opendj.security.SslContextBuilder;
import org.forgerock.opendj.security.SslOptions;
import org.forgerock.opendj.security.TrustManagers;

/* Perform LDAP authentication based on user credentials from a form,
 * connecting to an LDAPS enabled server.
 *
 * If LDAP authentication succeeds, then return a promise to handle the response.
 * If there is a failure, produce an error response and return it.
 */

username = request.form?.username[0]
password = request.form?.password[0]

// Update port number to match the LDAPS port of your directory service.
host = attributes.ldapHost ?: "localhost"
port = attributes.ldapPort ?: 1636

// Include options for SSL. In this example, the LDAP secure protocol is TLSv1.2,
// and the TrustAllManager blindly trusts all server certificates.
// In a production environment, replace TrustAllManager with a properly configured TrustManager.
ldapOptions = ldap.defaultOptions(context)
SslContextBuilder builder = new SslContextBuilder();
builder.trustManager(TrustManagers.trustAll());
SslOptions sslOptions = SslOptions.newSslOptions(builder.build()).enabledProtocols("TLSv1.2");
ldapOptions = ldapOptions.set(CommonLdapOptions.SSL_OPTIONS, sslOptions);
```

```
// Include SSL options in the LDAP connection
client = ldap.connect(host, port as Integer, ldapOptions)
try {

    // Assume the username is an exact match of either
    // the user ID, the email address, or the user's full name.
    filter = "(|(uid=%s)(mail=%s)(cn=%s))"

    user = client.searchSingleEntry(
            "ou=people,dc=example,dc=com",
            ldap.scope.sub,
            ldap.filter(filter, username, username, username))

    client.bind(user.name as String, password?.toCharArray())

    // Authentication succeeded.

    // Set a header (or whatever else you want to do here).
    request.headers.add("Ldap-User-Dn", user.name.toString())

    // Most LDAP attributes are multi-valued.
    // When you read multi-valued attributes, use the parse() method,
    // with an AttributeParser method
    // that specifies the type of object to return.
    attributes.cn = user.cn?.parse().asSetOfString()

    // When you write attribute values, set them directly.
    user.description = "New description set by my script"

    // Here is how you might read a single value of a multi-valued attribute:
    attributes.description = user.description?.parse().asString()

    // Call the next handler. This returns when the request has been handled.
    return next.handle(context, request)

} catch (AuthenticationException e) {
    // LDAP authentication failed, so fail the response with
    // HTTP status code 403 Forbidden.
    response = new Response(Status.FORBIDDEN)
    response.headers['Content-Type'] = "text/html; charset=utf-8"
    response.entity = "<html><p>Authentication failed: " + e.message + "</p></html>"

} catch (Exception e) {
    // Something other than authentication failed on the server side,
    // so fail the response with HTTP 500 Internal Server Error.
    response = new Response(Status.INTERNAL_SERVER_ERROR)
    response.headers['Content-Type'] = "text/html; charset=utf-8"
    response.entity = "<html><p>Server error: " + e.message + "</p></html>"

} finally {
    client.close()
}

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

15.5. Scripting SQL Queries

You can use a `ScriptableFilter` to look up information in a relational database and include the results in the request context.

The following filter looks up user credentials in a database given the user's email address, which is found in the form data of the request. The script then sets the credentials in headers, making sure the scheme is HTTPS to protect the request when it leaves IG:

```
/*
 * Copyright 2014-2017 ForgeRock AS. All Rights Reserved
 *
 * Use of this code requires a commercial software license with ForgeRock AS.
 * or with one of its affiliates. All use shall be exclusively subject
 * to such license between the licensee and ForgeRock AS.
 */

/*
 * 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()
def credentials = client.getCredentials(request.form?.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)
```

The previous script demonstrates a `ScriptableFilter` that uses a `SqlClient` class defined in another script. The following code listing shows the `SqlClient` class:

```
/*
 * Copyright 2014-2017 ForgeRock AS. All Rights Reserved
 *
 * Use of this code requires a commercial software license with ForgeRock AS.
 * or with one of its affiliates. All use shall be exclusively subject
 * to such license between the licensee and ForgeRock AS.
 */

import groovy.sql.Sql

import javax.naming.InitialContext
import javax.sql.DataSource

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

```
 */
class SqlClient {

    // Get a DataSource from the container.
    InitialContext context = new InitialContext()
    DataSource dataSource = context.lookup("jdbc/forgerock") as DataSource
    def 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
    }
}
```

To try the script, follow these steps:

1. Follow the tutorial in "Log in With Credentials From a Database".

   When everything in that tutorial works, you know that IG can connect to the database, look up users by email address, and successfully authenticate to the sample application.

2. Save the scripts as `$HOME/.openig/scripts/groovy/SqlAccessFilter.groovy` (`%appdata%\OpenIG\scripts\groovy\SqlAccessFilter.groovy` on Windows), and as `$HOME/.openig/scripts/groovy/SqlClient.groovy` (`%appdata%\OpenIG\scripts\groovy\SqlClient.groovy` on Windows).

3. Add the following route to your configuration as `$HOME/.openig/config/routes/11-db.json` (`%appdata%\OpenIG\config\routes\11-db.json` on Windows):

   ```
   {
   ```

```
    "handler": {
      "type": "Chain",
      "config": {
        "filters": [
          {
            "type": "ScriptableFilter",
            "config": {
              "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": "${matches(request.uri.path, '^/db')}"
}
```

The route calls the `ScriptableFilter` to look up credentials over SQL. It then uses calls a
`StaticRequestFilter` to build a login request. Although the script sets the scheme to HTTPS, the
`StaticRequestFilter` ignores that and resets the URI. This makes it easier to try the script without
additional steps to set up HTTPS.

To try the configuration, browse to a URL where a query string parameter specifies a valid email
address, such as http://openig.example.com:8080/db?mail=george@example.com.

If the lookup and authentication are successful, you see the profile page of the sample application.

15.6. Developing Custom Extensions

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(5) in the
Configuration Reference.

15.6.1. Key Extension Points

Interface Stability: Evolving (For information, see "ForgeRock Product Interface Stability" in the *Release Notes*.)

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 details, see 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. As described in "Embedding the Customization in IG", you must add your custom libraries to the `WEB-INF/lib/` directory of the IG .war file that you deploy.

Be sure to provide some 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.

15.6.2. Implementing 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:

```java
package org.forgerock.openig.doc;

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.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.
     */
    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.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 "Implementing a Class Alias Resolver".

15.6.3. Implementing 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;

import org.forgerock.openig.alias.ClassAliasResolver;

import java.util.HashMap;
```

```java
import java.util.Map;

/**
 * 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.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(String alias) {
        return ALIASES.get(alias);
    }
}
```

With this `ClassAliasResolver`, the filter configuration in "Implementing a Customized Sample Filter" can use the alias instead of the fully qualified class name, as follows:

```json
{
  "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.SampleClassAliasResolver
  ```

  If you have more than one resolver in your .jar file, add one line for each fully qualified class name.

15.6.4. Configuring 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.

15.6.5. Embedding the Customization in IG

After building your customizations into a .jar file, you can include them in the IG .war file for deployment. You do this by unpacking `IG-6.1.0.war`, including your .jar library in `WEB-INF/lib`, and then creating a new .war file.

For example, if your .jar file is in a project named `sample-filter`, and the development version is `1.0.0-SNAPSHOT`, you might include the file as in the following example:

```
$ mkdir root && cd root
$ jar -xf ~/Downloads/IG-6.1.0.war
$ cp ~/Documents/sample-filter/target/sample-filter-1.0.0-SNAPSHOT.jar WEB-INF/lib
$ jar -cf ../custom.war *
```

In this example, the resulting `custom.war` contains the custom sample filter. You can deploy the custom .war file as you would deploy `IG-6.1.0.war`.

Chapter 16
Auditing

The ForgeRock Common Audit Framework is a platform-wide infrastructure to handle audit events by using common audit event handlers. The handlers record events by logging them into files, relational databases, or syslog.

IG provides the following facilities to log audit events:

- JSON audit event handler, to log audit topics to JSON files. For more information, see JsonAuditEventHandler(5) in the *Configuration Reference*.

- The Elasticsearch search and analytics engine. For an example of how to record audit events in elasticsearch, see "To Record Audit Events in Elasticsearch".

- CSV files, with support for retention, rotation, and tamper-evident logs. For an example of how to record audit events in a CSV file, see "To Record Audit Events in a CSV File".

- Relational databases using JDBC.

- The UNIX system log (Syslog) facility.

- Java Message Service (JMS), to send asynchronous messages between clients. For an example of how to record audit events with a JMS audit event handler, see "To Record Audit Events With a JMS Audit Event Handler". For more information, see JmsAuditEventHandler(5) in the *Configuration Reference*.

- Splunk, to log IG events to a Splunk system. For an example of how to record audit events with a Splunk audit event handler, see "Recording Audit Events in Splunk".

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.

Transaction IDs from other services in the ForgeRock platform are sent as `X-ForgeRock-TransactionId` header values.

By default, IG does not trust transaction ID headers from client applications.

> **Note**
>
> If you trust transaction IDs sent by client applications, and want monitoring and reporting systems consuming the logs to allow correlation of requests as they traverse multiple servers, then set the boolean system property `org.forgerock.http.TrustTransactionHeader` to `true` in the Java command to start the container where IG runs.

To enable the audit framework for a route, you specify an audit service and configure an audit event handler. The following procedures describe how to record audit events in a CSV file and to the Elasticsearch search and analytics engine. For more information about recording audit events, see Audit Framework in the *Configuration Reference*.

16.1. Recording Audit Events in CSV

To Record Audit Events in a CSV File

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*.

1. Add the following route to the IG as `$HOME/.openig/config/routes/30-audit.json`.

   On Windows, add the route as `%appdata%\OpenIG\config\routes\30-audit.json`.

   ```
   {
     "handler": "ForgeRockClientHandler",
     "baseURI": "http://app.example.com:8081",
     "condition": "${matches(request.uri.path, '^/home/audit')}",
     "auditService": {
       "type": "AuditService",
       "config": {
         "config": {},
         "event-handlers": [
           {
             "class": "org.forgerock.audit.handlers.csv.CsvAuditEventHandler",
             "config": {
               "name": "csv",
               "logDirectory": "/tmp/logs",
               "buffering": {
                 "enabled": "true",
                 "autoFlush": "true"
               },
               "topics": [
                 "access"
               ]
             }
           }
         ]
       }
     }
   }
   ```

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.

2. Access the route on http://openig.example.com:8080/home/audit.

   The home page of the sample application should be displayed and the file `/tmp/logs/access.csv` should be updated.

16.2. Recording Audit Events in Elasticsearch

This section describes how to record audit events with an Elasticsearch audit event handler. For information about configuring the Elasticsearch event handler, see ElasticsearchAuditEventHandler(5) in the *Configuration Reference*.

To Record Audit Events in Elasticsearch

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*, and access Studio as described in "Accessing Studio" in the *Getting Started Guide*. IG must be running in development mode.

1. Make sure that Elasticsearch is installed and running.

   For Elasticsearch downloads and installation instructions, see the Elasticsearch *Getting Started* document. For information about configuring the Elasticsearch event handler, see ElasticsearchAuditEventHandler(5) in the *Configuration Reference*.

2. Browse to http://openig.example.com:8080/openig/studio, and select ⊕ Protect an Application.

3. Choose ☰ Structured to use the predefined menus and templates.

4. In the ✚ Create a route window, select Advanced Options, enter the following information, and then select Create route:

   • Base URI: `http://app.example.com:8081`

   • Condition: Path: `/home/audit`

   • Name: `30-elasticsearch`

5. Configure auditing:

   a. Select ➤ Audit, and then enable it.

   b. Select ✚ New event handler and then Elasticsearch event handler.

    c.   Enter the following information, and then save the settings:

        • Name: `elasticsearch`

       Leave the other fields with their default values and save.

    d.   In the event handlers frame, enable the event handler.

6. On the top-right of the screen, select ⋮ and ⤢ Display to review the route. The following route should be displayed:

```
{
  "name": "30-elasticsearch",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/audit')}",
  "monitor": false,
  "auditService": {
    "name": "AuditService",
    "type": "AuditService",
    "config": {
      "event-handlers": [
        {
          "class": "org.forgerock.audit.handlers.elasticsearch.ElasticsearchAuditEventHandler",
          "config": {
            "name": "elasticsearch",
            "indexMapping": {
              "indexName": "audit"
            },
            "connection": {
              "host": "localhost",
              "port": 9200,
              "useSSL": false
            },
            "topics": [
              "access"
            ]
          }
        }
      ]
    }
  },
  "handler": "ReverseProxyHandler"
}
```

7. (Optional) Select ⬚ Edit and set options to manage the connection, index mapping, and buffering. Use information in ElasticsearchAuditEventHandler(5) in the *Configuration Reference*.

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.

To Test the Setup

1. Access the route on http://openig.example.com:8080/home/audit.

The home page of the sample application is displayed and events are logged in Elasticsearch.

2. Access the ElasticSearch URL, `http://localhost:9200/audit/access/_search?q='"OPENIG-HTTP-ACCESS"'`

   The audit events logged in Elasticsearch are displayed.

3. Repeat the previous steps to confirm that each time you access the IG, Elasticsearch is updated.

16.3. Recording Audit Events in JMS

> **Important**
>
> This procedure is an example of how to record 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, see JmsAuditEventHandler(5) in the *Configuration Reference*.

To Record Audit Events With a JMS Audit Event Handler

Before you start, prepare IG as described in "*First Steps*" in the *Getting Started Guide*.

1. Add ActiveMQ client dependencies to IG:

   a. Download the following `.jar` files from :

      - `geronimo-j2ee-management_1.1_spec-1.0.1.jar`

      - `hawtbuf-1.11.jar`

      - `activemq-client-5.13.3.jar`

   b. Add the `.jar` files to the IG container, at `/path/to/jetty/webapps/ROOT/WEB-INF/lib`.

2. Download and install the ActiveMQ message broker from http://activemq.apache.org/. For help, see the the ActiveMQ documentation on the same site.

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. Add the following route to the IG as `$HOME/.openig/config/routes/30-jms.json`. On Windows, add the route as `%appdata%\OpenIG\config\routes\30-jms.json`.

   ```
   {
   ```

```
    "MyCapture" : "all",
    "auditService" : {
      "config" : {
        "event-handlers" : [
          {
            "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://openam.example.com:61616",
                  "topic.audit" : "audit"
                },
                "topicName" : "audit",
                "connectionFactoryName" : "ConnectionFactory"
              }
            }
          }
        ],
        "config" : { }
      },
      "type" : "AuditService"
    },
    "handler" : {
      "type" : "StaticResponseHandler",
      "config" : {
        "status" : 200,
        "headers" : {
          "content-type" : [ "text/plain" ]
        },
        "reason" : "found",
        "entity" : "Message from audited route"
      }
    },
    "monitor" : true,
    "condition" : "${request.uri.path == '/activemq_event_handler'}"
}
```

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. Access the route on http://openig.example.com:8080/activemq_event_handler.

Depending on how ActiveMQ is configured, audit events are displayed on the ActiveMQ console
or written to file. For example, the following log message can be written to a log file in the folder
where you installed ActiveMQ:

```
{
  "auditTopic": "access",
  "event": {
    "eventName": "OPENIG-HTTP-ACCESS",
```

```
      "timestamp": "2016-11-28T14:39:30.004Z",
      "transactionId": "882918f9-f7c3-47ee-9f87-5e3cfcfb98be-37",
      "server": {
        "ip": "0:0:0:0:0:0:0:1",
        "port": 8080
      },
      "client": {
        "ip": "0:0:0:0:0:0:0:1",
        "port": 56095
      },
      "http": {
        "request": {
          "secure": false,
          "method": "GET",
          "path": "http://openig.example.com:8080/activemq_event_handler",
          "queryParameters": {},
          "headers": {
            "accept": ["*/*"],
            "accept-encoding": ["gzip, deflate"],
            "Connection": ["keep-alive"],
            "host": ["openig.example.com:8080"],
            "user-agent": ["python-requests/2.9.1"]
          },
          "cookies": {}
        },
        "response": {
          "headers": {
            "Content-Length": ["26"],
            "Content-Type": ["text/plain"]
          }
        }
      },
      "response": {
        "status": "SUCCESSFUL",
        "statusCode": "200",
        "elapsedTime": 73,
        "elapsedTimeUnits": "MILLISECONDS"
      },
      "_id": "882918f9-f7c3-47ee-9f87-5e3cfcfb98be-38"
  }
}
```

16.4. Recording Audit Events in JSON

This section describes how to record audit events with a JSON audit event handler. For information about configuring the JSON event handler, see JsonAuditEventHandler(5) in the *Configuration Reference*.

To Record Audit Events With a JSON Audit Event Handler

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*, and access Studio as described in "Accessing Studio" in the *Getting Started Guide*. IG must be running in development mode.

1. Browse to http://openig.example.com:8080/openig/studio, and select ⊕ Protect an Application.

2. Choose ☰ Structured to use the predefined menus and templates.

3. In the ✚ Create a route window, select Advanced Options, enter the following information, and then select Create route:

   • Base URI: `http://app.example.com:8081`

   • Condition: Path: `/home/audit`

   • Name: `30-json`

4. Configure auditing:

   a. Select ➡ Audit, and then enable it.

   b. Select ✚ New event handler and then JSON audit event handler.

   c. Enter the following information, and then save the settings:

      • Name: `json`

      • Log directory: `/tmp/logs`

      Leave the other fields with their default values and save.

5. In the event handlers frame, enable the event handler

6. On the top-right of the screen, select ⋮ and ⬈ Display to review the route. The following route should be displayed:

```
{
  "name": "30-json",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/audit')}",
  "monitor": false,
  "auditService": {
    "name": "AuditService",
    "type": "AuditService",
    "config": {
      "event-handlers": [
        {
          "class": "org.forgerock.audit.handlers.json.JsonAuditEventHandler",
          "config": {
            "name": "json",
            "logDirectory": "/tmp/logs",
            "elasticsearchCompatible": false,
            "topics": [
              "access"
            ],
            "buffering": {
              "maxSize": 100000,
              "writeInterval": "100 ms"
            },
```

```
            "rotationRetentionCheckInterval": "1 m"
          }
        }
      ]
    }
  },
  "handler": "ReverseProxyHandler"
}
```

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.

7. (Optional) Select 🖉 Edit and set options to manage compatibility with the ElasticSearch JSON format, or file rotation, retention, and buffering. Use information in JsonAuditEventHandler(5) in the *Configuration Reference*.

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.

To Test the Setup

• Access the route on http://openig.example.com:8080/home/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:

```
{
  "eventName": "OPENIG-HTTP-ACCESS",
  "timestamp": "2016-11-08T15:39:59.128Z",
  "transactionId": "a386a21c-0ceb-4c6b-af77-167bd71f0161-1",
  "server": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 8080
  },
  "client": {
    "ip": "0:0:0:0:0:0:0:1",
    "port": 34066
  },
  "http": {
    "request": {
      "secure": false,
      "method": "GET",
      "path": "http://openig.example.com:8080/home/json-audit",
      "queryParameters": {},
      "headers": {
        "accept": ["text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8"],
        "accept-encoding": ["gzip, deflate"],
        "Accept-Language": ["en-US;q=1"],
```

```
          "cache-control": ["max-age=0"],
          "Connection": ["keep-alive"],
          "host": ["openig.example.com:8080"],
          "upgrade-insecure-requests": ["1"],
          "user-agent": ["Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_6)
                         AppleWebKit / 602.2 .14(KHTML, like Gecko)
                         Version / 10.0 .1 Safari / 602.2 .14 "]
        },
        "cookies": {
          "i18next": "en"
        }
      },
      "response": {
        "status": "SUCCESSFUL",
        "statusCode": "200",
        "elapsedTime": 104,
        "elapsedTimeUnits": "MILLISECONDS"
      },
      "_id": "a386a21c-0ceb-4c6b-af77-167bd71f0161-2"
    }
}
```

16.5. Recording Audit Events in Splunk

This section describes how to set up a Splunk audit event handler to log IG events to a Splunk system. For information about configuring the Splunk event handler, see SplunkAuditEvenHandler(5) in the *Configuration Reference*.

To Set Up Splunk

This procedure assumes a Splunk instance running on the same host as IG. Adjust the instructions for your Splunk system.

Before you start, prepare IG as described in "*First Steps*" in the *Getting Started Guide*.

1. Download Splunk from http://www.splunk.com, and install it with the default configuration. If you don't already have a Splunk account, create one.

    > **Tip**
    >
    > Splunk currently uses the following ports by default: 8000, 8065, 8088, 8089, and 8091. Before you install Splunk, make sure that these ports are free. Alternatively, change the Splunk installation and IG route to use other ports.
    >
    > To find port numbers and other settings used by Splunk, select Server settings > General settings in the Splunk web interface.

2. With Splunk running, create a new source type and associate it with log data from IG:

    a. In the Splunk web interface, select Settings > Source Types > New Source Type.

b. In the Create Source Type window, enter a name for the source type, for example, `openig`.

c. In the Event Breaks panel of the same window, select Regex... and enter `^{` to indicate how the bulk messages are separated.

d. Accept all of the other values as default and select Save.

3. Create an HTTP Event Collector to provide an authorization token so that IG can log events to Splunk:

a. Select Settings > Data Inputs > HTTP Event Collector > New Token.

b. Enter a Name for the token, for example, `openig`, leave the other fields with their default values, and select Next.

c. In the Input Settings screen, select Select > Select Source Type > Custom, and then select the source type you created in the previous step.

d. Select Review and then Submit.

    An authorization token is displayed. Make a note of the value or keep it on the screen so that you use it as the value of `authzToken` in "To Set Up IG for the Splunk Audit Event Handler".

4. In the same window, check that the Global Settings are configured correctly. For example, make sure that all tokens are enabled and that SSL is not enabled.

   The HTTP port number displayed in these global settings is used as the value of `port` in "To Set Up IG for the Splunk Audit Event Handler".

To Set Up IG for the Splunk Audit Event Handler

1. Add the following route to the IG as `$HOME/.openig/config/routes/30-splunk.json`.

   On Windows, add the route as `%appdata%\OpenIG\config\routes\30-splunk.json`.

```
{
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "entity": "Creating Splunk Audit Event"
    }
  },
  "condition": "${matches(request.uri.path, '^/splunk')}",
  "auditService": {
    "type": "AuditService",
    "config": {
      "config": {},
      "event-handlers": [
        {
          "class": "org.forgerock.audit.handlers.splunk.SplunkAuditEventHandler",
          "config": {
```

```
            "name": "splunk",
            "topics": [
              "access"
            ],
            "enabled": true,
            "connection": {
              "useSSL": false,
              "host": "localhost",
              "port": 8088
            },
            "buffering": {
              "maxSize": 10000,
              "writeInterval": "100 ms",
              "maxBatchedEvents": 500
            },
            "authzToken": "<splunk-authorization-token>"
          }
        }
      ]
    }
  }
}
```

2. Set the value of `"authzToken"` to the value of the authorization token returned in "To Set Up Splunk".

To Test the Setup

1. Access the route on http://openig.example.com:8080/splunk

2. Access the Splunk web interface on http://localhost:8000, and select Search & Reporting > Data Summary.

   Depending on how Splunk is configured, audit events are displayed on the web interface.

FORGEROCK

Chapter 17
Monitoring

IG can collect metrics on requests and responses, and expose metrics over the following HTTP endpoints:

- Prometheus Scrape Endpoint

- Forgerock Common REST Monitoring Endpoint

- IG Route Monitoring Endpoint

Metrics exposed by IG are listed in Monitoring Metrics(5) in the *Configuration Reference*.

17.1. Prometheus Scrape Endpoint

All ForgeRock products expose a monitoring endpoint where Prometheus can scrape metrics.

The endpoint is available in IG without any configuration. When IG is set up as described in "*First Steps*" in the *Getting Started Guide*, the endpoint is `http://openig.example.com:8080/openig/metrics/prometheus`.

You can run queries on the endpoint to access metrics only for certain routes or handlers, or for any other subset of metrics. For some examples, see "To Monitor the Prometheus Scrape Endpoint".

By default, everyone has read access to the Prometheus endpoint. No special credentials or privileges are required. For information about how to restrict access, see "Protecting the Forgerock Common REST Monitoring Endpoint".

The metrics are exposed in standard Prometheus format. For information about configuring Prometheus to scrape metrics, see the Prometheus web site.

Metrics are available for each router, subrouter, and route in the configuration. For a list of exposed metrics, see Monitoring Metrics(5) in the *Configuration Reference*.

All metrics are prefixed with `ig_`, and where applicable include the measurement unit.

To Monitor the Prometheus Scrape Endpoint

1. Add the following route to the IG configuration as `$HOME/.openig/config/routes/myroute1.json` (on Windows, `%appdata%\OpenIG\config\routes`):

```
{
  "name": "myroute1",
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "reason": "OK",
      "entity": "Hello, world from myroute1!"
    }
  },
  "condition": "${matches(request.uri.path, '^/myroute1')}"
}
```

The route contains a StaticResponseHandler to display a simple message.

2. Access the route a few times, on http://openig.example.com:8080/myroute1.

3. Query the Prometheus Scrape Endpoint at http://openig.example.com:8080/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._router",}
 0.0
...
```

17.2. Forgerock Common REST Monitoring Endpoint

All ForgeRock products expose a monitoring endpoint where metrics are exposed as a JSON format monitoring resource.

The endpoint is available in IG without any configuration. When IG is set up as described in "*First Steps*" in the *Getting Started Guide*, the endpoint is `http://openig.example.com:8080/openig/metrics/api?_queryFilter=true`.

You can query the endpoint to reveal metrics for certain routes or handlers, or for any other subset of metrics. For some examples, see "To Monitor the Forgerock Common REST Monitoring Endpoint".

By default, everyone has read access to the endpoint. No special credentials or privileges are required. For information about how to manage access, see "Protecting the Forgerock Common REST Monitoring Endpoint".

To Monitor the Forgerock Common REST Monitoring Endpoint

Before you start, prepare IG as described in "*First Steps*" in the *Getting Started Guide*.

1. Set up IG and some example routes, as described in the first few steps of "To Monitor the Prometheus Scrape Endpoint".

2. Query the Forgerock Common REST Monitoring Endpoint at http://openig.example.com:8080/openig/metrics/api?_prettyPrint=true&_sortKeys=_id&_queryFilter=true

    Metrics for `myroute1` 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
}
```

3. Change the query to access metrics only for `myroute1`: http://openig.example.com:8080/
   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.

17.3. IG Route Monitoring Endpoint (Deprecated)

When a route has `"monitor": true`, the IG Route Monitoring Endpoint exposes metrics for that route. Metrics are exposed as a JSON format monitoring resource.

You can read the metrics, set the percentiles in the distribution for which to maintain response time metrics, and post a reset to clear metrics for the route. When the IG container stops, IG discards the collected metrics.

When the log level is `INFO` or finer, monitoring endpoints are written in `$HOME/.openig/logs/route-system.log` when you start IG, or add or edit routes. Endpoints are constructed from:

• The base URL of IG.

• The object name of the main router configured in `config.json`:

  • When `config.json` is not provided, the 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.

- The subrouter name, when a route contains a subrouter and a main router.

- The route ID.

When IG is set up as described in "*First Steps*" in the *Getting Started Guide*, IG Route Monitoring Endpoint metrics are available at `http://openig.example.com:8080/openig/api/system/objects/_router/routes/`*route name*`/monitoring`

IG Route Monitoring Endpoint also serve API descriptors at runtime. When you retrieve an API descriptor, 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 about API descriptors for monitoring endpoints, see "Understanding IG APIs With API Descriptors". For information about Common REST, see "About ForgeRock Common REST" in the *Configuration Reference*.

To Monitor the IG Route Monitoring Endpoint

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*.

1. Browse to http://openig.example.com:8080/openig/studio, and select ⊕ Protect an Application.

2. In the Create a route window, select Advanced Options and enter the following information, and then select Create route:

   - Base URI: `http://app.example.com:8081`

   - Condition: Path: `/home/monitor`

   - Name: `00-monitor`

3. Select Statistics, and then enable statistics.

4. On the top-right of the screen, select ⋮ and ⤢ Display to review the route.

   ```
   {
     "name": "00-monitor",
     "baseURI": "http://app.example.com:8081",
     "condition": "${matches(request.uri.path, '^/home/monitor')}",
     "monitor": {
       "enabled": true
     },
     "handler": "ReverseProxyHandler"
   }
   ```

Notice the following features of the route:

- The route matches requests to `/home/monitor`.

- Each time the route is accessed, the ClientHandler passes the request to the sample application and IG collects metrics.

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.

6. Access the route a few times at http://openig.example.com:8080/home/monitor.

7. Go to the IG Route Monitoring Endpoint at http://openig.example.com:8080/openig/api/system/objects/_router/routes/00-monitor/monitoring.

   The monitoring resource displays metrics for requests and responses:

```
{
  "requests": {
    "total": 29,
    "active": 0
  },
  "responses": {
    "total": 29,
    "info": 0,
    "success": 29,
    "redirect": 0,
    "clientError": 0,
    "serverError": 0,
    "other": 0,
    "errors": 0,
    "null": 0
  },
  "throughput": {
    "mean": 2.1,
    "lastMinute": 0,
    "last5Minutes": 0,
    "last15Minutes": 0
  },
  "responseTime": {
    "mean": 4845.642,
    "median": 3552.415,
    "standardDeviation": 4236.919,
    "total": 140.836059,
    "percentiles": {
      "0.999": 25831.901,
      "0.9999": 25831.901,
      "0.99999": 25831.901
    }
  }
}
```

17.4. Protecting the Forgerock Common REST Monitoring Endpoint

By default, everyone has read access to the Prometheus Scrape Endpoint and Forgerock Common REST Monitoring Endpoint. No special credentials or privileges are required.

To protect these 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:

To Protect the Forgerock Common REST Monitoring Endpoint

1. Add the following script to the IG configuration as `$HOME/.openig/scripts/groovy/BasicAuthResourceServerFilter.groovy` (on Windows, `%appdata%\OpenIG\scripts\groovy\BasicAuthResourceServerFilter.groovy`): this script is a very simple implementation of the HTTP Basic Authentication mechanism.

   With such a filter, the endpoint `/openig/metrics` is not now protected through the standard HTTP Basic Authentication mechanism. It is possible to use here any kind of filter to protect that endpoint.

```groovy
/**
 * This scripts is a simple implementation of HTTP Basic 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
 *  - password: the expected password
 */

import static org.forgerock.util.promise.Promises.newResultPromise;

import java.nio.charset.Charset;
import org.forgerock.util.encode.Base64;

String authorizationHeader = request.getHeaders().getFirst("Authorization");
if (authorizationHeader == null) {
    // No credentials provided, reply that they are needed.
    Response response = new Response(Status.UNAUTHORIZED);
    response.getHeaders().put("WWW-Authenticate", "Basic realm=\"" + realm + "\"");
    return newResultPromise(response);
}

String expectedAuthorization = "Basic " + Base64.encode((username + ":" +
 password).getBytes(Charset.defaultCharset()))
if (!expectedAuthorization.equals(authorizationHeader)) {
    return newResultPromise(new Response(Status.FORBIDDEN));
}
// Credentials are as expected, let's continue
return next.handle(context, request);
```

For information about scripting filters and handlers, see "*Extending Identity Gateway*".

2. Add the following file as `$HOME/.openig/config/admin.json` (on Windows, `%appdata%\OpenIG\config\admin.json`):

```
{
  "heap": [{
    "name": "ClientHandler",
    "type": "ClientHandler"
  }, {
    "name": "MetricsProtectionFilter",
    "type": "ScriptableFilter",
    "config": {
      "type": "application/x-groovy",
      "file": "BasicAuthResourceServerFilter.groovy",
      "args": {
        "realm": "/",
        "username": "metric",
        "password": "password"
      }
    }
  }],
  "prefix": "openig"
}
```

3. Restart IG to reload the configuration.

For more information, see "Automatically Created Objects" in the *Configuration Reference*.

Chapter 18
Throttling the Rate of Requests to Protected Applications

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 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:
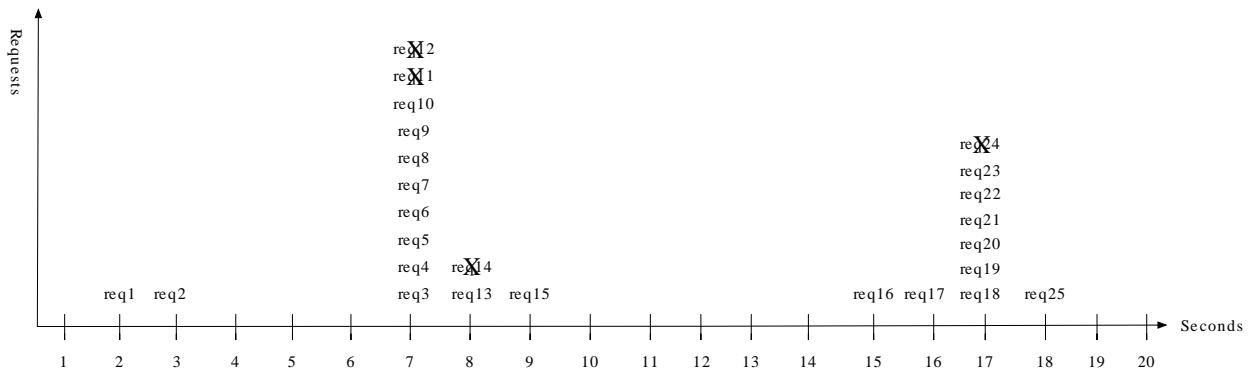
- At 7 seconds, 2 requests have previously passed when there is a burst of 9 requests. IG allows 8 requests, but disregards the 9th because the throttling rate for the 10-second throttling period has been reached.

- 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 http://openig.example.com:8080/home/throttle-scriptable HTTP/1.1

. . .

HTTP/1.1 429 Too Many Requests
Retry-After: 10
```

This section describes how to set up simple, mapped, and scriptable throttling filters. For more configuration options, see ThrottlingFilter(5) in the *Configuration Reference*

18.1. Configuring a Simple Throttling Filter

This section describes how to use Studio 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.
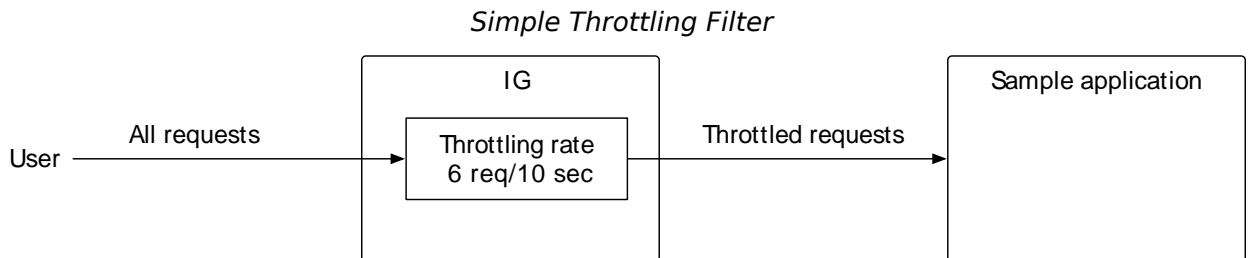
Simple Throttling Filter

To try this example without using Studio, add the route in the following procedure to the IG configuration as `$HOME/.openig/config/routes/simple-throttling.json`. On Windows, add the route as `%appdata%\OpenIG\config\routes\simple-throttling.json`.

To Configure a Simple Throttling Filter

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*, and access Studio as described in "Accessing Studio" in the *Getting Started Guide*. IG must be running in development mode.

1. Browse to http://openig.example.com:8080/openig/studio, and select ⊕ Protect an Application.

2. Choose ☰ Structured to use the predefined menus and templates.

3. In the ➕ Create a route window, select Advanced Options, enter the following information, and then select Create route:

    • Base URI: `http://app.example.com:8081`

- Condition: Path: `/home/throttle-simple`

- Name: `00-throttle-simple`

4. Select ▼ Throttling, and then enable throttling.

5. In GROUPING POLICY, select to 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.

6. In RATE POLICY, select Fixed, and 🔲 Edit, and then allow 6 requests each 10 seconds.

7. On the top-right of the screen, select ⋮ and ☑ Display to view the route. The following route should be displayed:

```
{
  "name": "00-throttle-simple",
  "monitor": false,
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(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"
    }
  }
}
```

   Notice the following features of the new 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.

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.

To Test the Setup

1. With IG and the sample application running, use **curl**, a bash script, or another tool to access the following route in a loop: http://openig.example.com:8080/home/simple-throttle.

   Accessing the route in a loop runs the request multiple times in quick succession, allowing you to test the throttling rate.

   ```
   $ curl -v http://openig.example.com:8080/home/throttle-simple/\[01-10\] \
   > /tmp/simple-throttle.txt 2>&1
   ```

2. Search the output file to see the result:

   ```
   $ grep "< HTTP/1.1" /tmp/simple-throttle.txt | sort | uniq -c

   6 < HTTP/1.1 200 OK
   4 < HTTP/1.1 429 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.

18.2. Configuring a Mapped Throttling Filter

This section describes how to configure a mapped throttling policy. The grouping policy defines criteria to group requests, and the rate policy defines the criteria by which rates are mapped.

In the following example, the grouping policy is based on the user ID, and the throttling rate is based on the user status. 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 and therefore has the default rate.

Mapped Throttling Filter

## To Configure a Mapped Throttling Filter

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*, and access Studio as described in "Accessing Studio" in the *Getting Started Guide*. IG must be running in development mode.

1. Add the following CSV file as `/tmp/userfile`:

   ```
   UserId,status
   Alice,gold
   Bob,gold
   Carol,silver
   Dave,bronze
   ```

2. Browse to http://openig.example.com:8080/openig/studio, and select ⊕ Protect an Application.

3. Choose ☰ Structured to use the predefined menus and templates.

4. In the ✚ Create a route window, select Advanced Options, enter the following information, and then select Create route:

   • Base URI: `http://app.example.com:8081`

   • Condition: Path: `/home/throttle-mapped`

- Name: `00-throttle-mapped`

5. Select ▼ Throttling, and then enable throttling.

6. Set up the grouping policy:

   a. In GROUPING POLICY, select to 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.

   b. Select to group requests by custom criteria.

   c. Enter the custom expression `${attributes.rate.UserId}`.

   d. Save the grouping policy.

7. Set up the rate policy:

   a. In RATE POLICY, select Mapped.

   b. Select to map requests by custom criteria.

   c. Enter the custom expression `${attributes.rate.status}`.

   d. In Default Rate, select ⊡ Edit and change default rate to 1 request each 10 seconds.

   e. In Mapped Rates, add the following rate for gold status:

      - Match Value: `gold`

      - Number of requests: `6`

      - Period: `10 seconds`

   f. Add a different rate for silver status:

      - Match Value: `sales.example.com`

      - Number of requests: `3`

      - Period: `10 seconds`

   g. Save the rate policy.

8. Add the FileAttributesFilter:

   a. 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.

b. Insert the following FileAttributesFilter at the beginning of the chain and then save the route:

```
{
   "type": "FileAttributesFilter",
   "config": {
      "file": "/tmp/userfile",
      "key": "UserId",
      "value": "${request.headers['UserId'][0]}",
      "target": "${attributes.rate}"
   }
}
```

9. Select ⋮ and ⌕ Display to view the route. The following route should be displayed:

```
{
  "name": "00-throttle-mapped",
  "monitor": false,
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/throttle-mapped')}",
  "capture": "all",
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "FileAttributesFilter",
          "config": {
            "file": "/tmp/userfile",
            "key": "UserId",
            "value": "${request.headers['UserId'][0]}",
            "target": "${attributes.rate}"
          }
        },
        {
          "type": "ThrottlingFilter",
          "name": "ThrottlingFilter-1",
          "config": {
            "requestGroupingPolicy": "${attributes.rate.UserId}",
            "throttlingRatePolicy": {
              "type": "MappedThrottlingPolicy",
              "name": "MappedPolicy",
              "config": {
                "throttlingRateMapper": "${attributes.rate.status}",
                "throttlingRatesMapping": {
                  "gold": {
                    "numberOfRequests": 6,
                    "duration": "10 s"
                  },
                  "silver": {
                    "numberOfRequests": 3,
```

```
                 "duration": "10 s"
               }
            },
            "defaultRate": {
               "numberOfRequests": 1,
               "duration": "10 s"
            }
          }
        }
      }
    }],
    "handler": "ReverseProxyHandler"
  }
 }
}
```

Notice the following features of the route:

- The route matches requests to `/home/throttle-mapped`.

- The FileAttributesFilter uses the request header `UserId` and the CSV file to expose the user IDs and statuses in the target `${attributes.rate}`.

  Whenever the CSV file is edited to change the user status, the throttling rate for the user is also changed.

- The request grouping policy of the ThrottlingFilter groups requests by user ID. This means that the throttling rate is applied independently to each user ID.

- The mapping policy applies a throttling rate to requests from users with gold or silver status. If the request does not have a mapped status, the mapping policy applies the default rate.

10. 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.

To Test the Setup

1. With IG and the sample application running, access the following route in a loop: http://openig.example.com:8080/home/throttle-mapped.

   Accessing the route in a loop runs the request multiple times in quick succession, allowing you to test the throttling rate. You can use a command-line tool such as **curl** to run the following commands in quick succession or in a bash script:

```
$ curl -v http://openig.example.com:8080/home/throttle-mapped/\[01-10\]
 \
--header "UserId:Alice" \
> /tmp/Alice.txt 2>&1

$ curl -v http://openig.example.com:8080/home/throttle-mapped/\[01-10\] \
 --header "UserId:Bob" \
> /tmp/Bob.txt 2>&1

$ curl -v http://openig.example.com:8080/home/throttle-mapped/\[01-10\]
 \
--header "UserId:Carol" \
> /tmp/Carol.txt 2>&1

$ curl -v http://openig.example.com:8080/home/throttle-mapped/\[01-10\]
 \
--header "UserId:Dave" \
> /tmp/Dave.txt 2>&1
```

2. Search the output files to see the result for each user and each department:

```
$ grep "< HTTP/1.1" /tmp/Alice.txt | sort | uniq -c

 6 < HTTP/1.1 200 OK
 4 < HTTP/1.1 429 Too Many Requests

$ grep "< HTTP/1.1" /tmp/Bob.txt | sort | uniq -c

 6 < HTTP/1.1 200 OK
 4 < HTTP/1.1 429 Too Many Requests

$ grep "< HTTP/1.1" /tmp/Carol.txt | sort | uniq -c

 3 < HTTP/1.1 200 OK
 7 < HTTP/1.1 429 Too Many Requests

$ grep "< HTTP/1.1" /tmp/Dave.txt | sort | uniq -c

 1 < HTTP/1.1 200 OK
 9 < HTTP/1.1 429 Too Many Requests
```

Notice that Alice and Bob, with gold status, are allowed six requests, and Carol, with silver status, is allowed three. This is consistent with the mapping in `00-throttle-mapped.json`. Because Dave has the bronze status, which is not mapped, he gets the default rate.

3. Edit the CSV file to change the status of Alice from gold to silver, and then run the previous steps again.

Notice that with a silver status, Alice is now allowed only three requests in 10 seconds.

18.2.1. Considerations for Dynamic Throttling

The following image illustrates what can happen when the throttling rate defined by `throttlingRateMapping` changes frequently or quickly:

Dynamic Throttling Rate
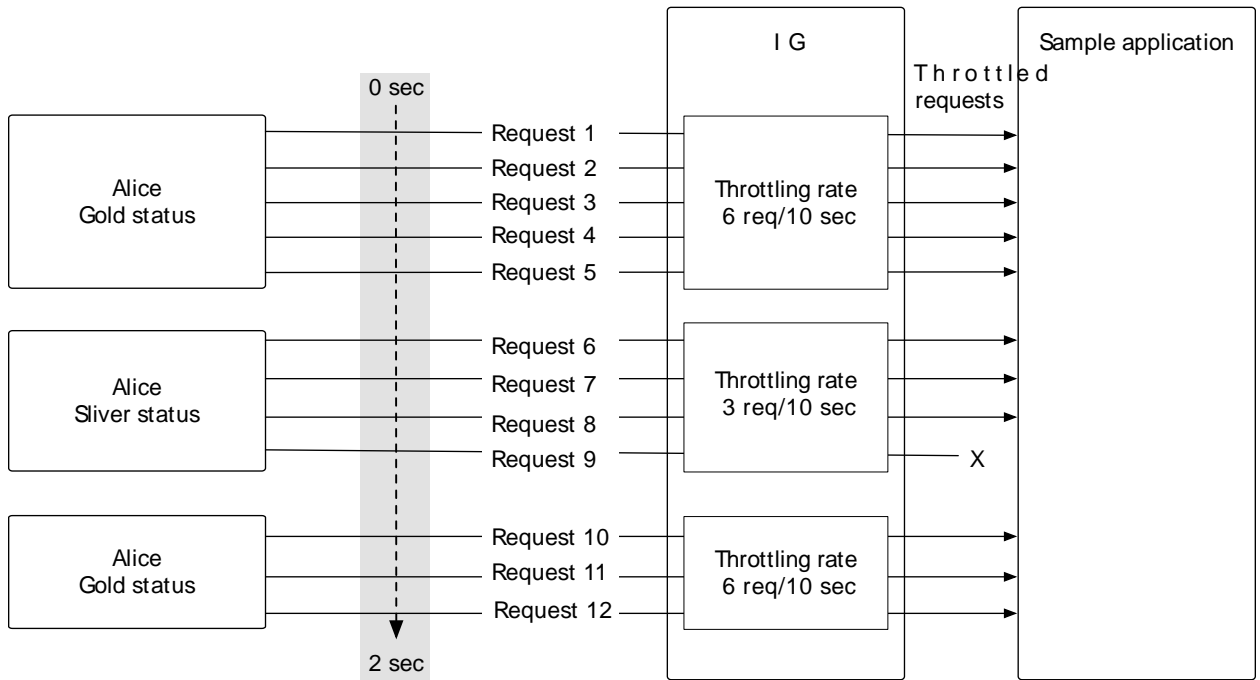
In the image, Alice starts out with a gold status. In a two second period, she sends five requests, is downgraded to silver, sends four requests, is upgraded back to gold, and then sends three more requests.

After making five requests with a gold status, Alice has almost reached her throttling rate. When her status is downgraded to silver, those requests are disregarded and the full throttling rate for silver is applied. Alice can now make three more requests even though she had nearly reached her throttling rate with a gold status.

After making three requests with a silver status, Alice has reached her throttling rate. When she makes a fourth request, the request is refused.

Alice is now upgraded back to gold and can now make six more requests even though she had reached her 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.

18.3. Configuring a Scriptable Throttling Filter

This section describes how to configure a scriptable throttling policy, with script applies a throttling rate of 6 requests/10 seconds to requests from gold status users. For all other requests, the script returns `null`. When the script returns `null`, the default rate of 1 request/10 seconds is applied.

For information, see "Scripting in Studio".

To try this example without using Studio, add the route in the following procedure to the IG configuration as `$HOME/.openig/config/routes/throttle-scriptable.json`. On Windows, add the route as `%appdata%\OpenIG\config\routes\throttle-scriptable.json`.

To Configure a Scriptable Throttling Filter

Before you start, prepare IG and the sample application as described in "*First Steps*" in the *Getting Started Guide*, and access Studio as described in "Accessing Studio" in the *Getting Started Guide*. IG must be running in development mode.

1. Add the following CSV file as `/tmp/userfile`:

    ```
    UserId,status
    Alice,gold
    Bob,gold
    Carol,silver
    Dave,bronze
    ```

2. Browse to http://openig.example.com:8080/openig/studio, and select ⊕ Protect an Application.

3. Choose ≣ Structured to use the predefined menus and templates.

4. In the ✚ Create a route window, select Advanced Options, enter the following information, and then select Create route:

    *   Base URI: `http://app.example.com:8081`

    *   Condition: Path: `/home/throttle-scriptable`

    *   Name: `throttle-scriptable`

5. Select ▼ Throttling, and then enable throttling.

6. Set up the grouping policy:

    a. In GROUPING POLICY, select to 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.

    b. Select to group requests by custom criteria.

c. Enter the custom expression `${attributes.rate.UserId}`.

d. Save the grouping policy.

7. Set up the rate policy:

a. In RATE POLICY, select Scripted.

b. 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.

c. Add the following argument/value pairs:

- argument: `status`, value: `"gold"`

- argument: `rate`, value: `6`

- argument: `duration`, value: `"10 seconds"`

d. Replace the default script with the content of a valid Groovy script. For example, enter the following script:

```
if (attributes.rate.status == status) {
  return new ThrottlingRate(rate, duration)
} else {
  return null
}
```

> **Tip**
>
> Alternatively, you can skip the step where you define arguments, and add the following script instead:
>
> ```
> if (attributes.rate.status == 'gold') {
>   return new ThrottlingRate(6, '10 seconds')
> } else {
>   return null
> }
> ```

> **Note**
>
> Studio does not check the validity of the Groovy script.

When you save, the script is added to the list of reference scripts available to use in scriptable throttling filters.

e. Enable the default rate, and set it to 1 request each 10 seconds.

f. Save the rate policy.

8. Add the FileAttributesFilter:

a. 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.

b. Insert the following FileAttributesFilter at the beginning of the chain and then save the route:

```
{
  "type": "FileAttributesFilter",
  "config": {
    "file": "/tmp/userfile",
    "key": "UserId",
    "value": "${request.headers['UserId'][0]}",
    "target": "${attributes.rate}"
  }
}
```

9. Select ⋮ and ⤢ Display to view the route. The following route should be displayed:

```
{
  "name": "throttle-scriptable",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/throttle-scriptable')}",
  "monitor": false,
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "FileAttributesFilter",
          "config": {
            "file": "/tmp/userfile",
            "key": "UserId",
            "value": "${request.headers['UserId'][0]}",
            "target": "${attributes.rate}"
          }
        },
        {
          "name": "ThrottlingFilter-1",
          "type": "ThrottlingFilter",
          "config": {
            "requestGroupingPolicy": "${attributes.rate.UserId}",
            "throttlingRatePolicy": {
              "type": "DefaultRateThrottlingPolicy",
              "config": {
                "delegateThrottlingRatePolicy": {
                  "name": "ScriptedPolicy",
                  "type": "ScriptableThrottlingPolicy",
                  "config": {
                    "type": "application/x-groovy",
                    "source": [
                      "if (attributes.rate.status == status) {",
```

```
                         "    return new ThrottlingRate(rate, duration)",
                         "} else {",
                         "    return null",
                         "}"
                       ],
                       "args": {
                         "status": "gold",
                         "rate": 10,
                         "duration": "10 seconds"
                       }
                     }
                   },
                   "defaultRate": {
                     "numberOfRequests": 1,
                     "duration": "10 s"
                   }
                 }
               }
             }
           }
         }
       ],
       "handler": "ReverseProxyHandler"
     }
   }
}
```

Notice the following features of the new route:

- 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.

10. 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.

To Test the Setup

1. With IG and the sample application running, access the following route in a loop: http://openig.example.com:8080/home/throttle-scriptable.

   Accessing the route in a loop runs the request multiple times in quick succession, allowing you to test the throttling rate. You can use a command-line tool such as **curl** to run the following commands in quick succession or in a bash script:

```
$ curl -v http://openig.example.com:8080/home/throttle-scriptable/\[01-10\]
 \
--header "UserId:Alice" \
> /tmp/Alice.txt 2>&1

$ curl -v http://openig.example.com:8080/home/throttle-scriptable/\[01-10\]
 \
--header "UserId:Bob" \
> /tmp/Bob.txt 2>&1

$ curl -v http://openig.example.com:8080/home/throttle-scriptable/\[01-10\]
 \
--header "UserId:Carol" \
> /tmp/Carol.txt 2>&1

$ curl -v http://openig.example.com:8080/home/throttle-scriptable/\[01-10\]
 \
--header "UserId:Carol" \
> /tmp/Carol.txt 2>&1
```

2. Search the output files to see the result for each user and each organization:

```
$ grep "< HTTP/1.1" /tmp/Alice.txt | sort | uniq -c

6 < HTTP/1.1 200 OK
4 < HTTP/1.1 429 Too Many Requests

$ grep "< HTTP/1.1" /tmp/Bob.txt | sort | uniq -c

6 < HTTP/1.1 200 OK
4 < HTTP/1.1 429 Too Many Requests

$ grep "< HTTP/1.1" /tmp/Carol.txt | sort | uniq -c

1 < HTTP/1.1 200 OK
9 < HTTP/1.1 429 Too Many Requests

$ grep "< HTTP/1.1" /tmp/Carol.txt | sort | uniq -c

1 < HTTP/1.1 200 OK
9 < HTTP/1.1 429 Too Many Requests
```

Notice that Alice and Bob, with gold status, are allowed 6 requests, consistent with the value in the script. Carol and Dave, with silver and bronze status, get the default throttling rate.

Chapter 19
Logging Events

Log messages in IG and third-party dependencies are recorded using the Logback implementation of the Simple Logging Facade for Java (SLF4J) API.

19.1. Default Logging Behavior

Log messages are recorded with the following default 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`.

- When a route is accessed, log messages for requests and responses passing through the route are written to a log file in `$HOME/.openig/logs`, and named by the route name or filename.

  For more information, see "Capturing Log Messages for Routes" and CaptureDecorator(5) in the *Configuration Reference*.

- 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 logging level.

19.2. Reference Logback Configuration

The content and format of logs 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 level to `INFO`, 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"?><!--
  Copyright 2016-2018 ForgeRock AS. All Rights Reserved
```

```
  Use of this code requires a commercial software license with ForgeRock AS.
  or with one of its affiliates. All use shall be exclusively subject
  to such license between the licensee and ForgeRock AS.
--><configuration>

  <!--
    Avoid logs to be flooded in case of repetitive messages.

    Configuration properties:
     * AllowedRepetitions (int): threshold above which same messages won't be logged anymore
     * CacheSize (int): Remove eldest entry when hitting max size
   -->
  <!--<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">
    <encoder>
      <pattern>%nopex[%thread] %highlight(%-5level) %boldWhite(%logger{35}) @%mdc{routeId:-system} -
%message%n%highlight(%rootException{short})</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-${routeId}" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${ig.instance.dir}/logs/route-${routeId}.log</file>

        <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
          <!-- Rotate files daily -->
          <fileNamePattern>${ig.instance.dir}/logs/route-${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>%date{"yyyy-MM-dd'T'HH:mm:ss,SSSXXX", UTC} | %-5level | %thread | %logger{20} | @
%mdc{routeId:-system} | %message%n%xException</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="INFO">
    <appender-ref ref="SIFT"/>
```

```
        <appender-ref ref="STDOUT"/>
    </root>
</configuration>
```

19.3. Capturing Log Messages for Routes

IG provides a CaptureDecorator to capture messages for requests and responses. The decorator can capture the context and entity of inbound and outbound messages for the route, or for the individual handlers and filters in the route.

Captured information is written to SLF4J logs. For more information about the decorator configuration, see CaptureDecorator(5) in the *Configuration Reference*.

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:

Capturing Log Messages for Routes
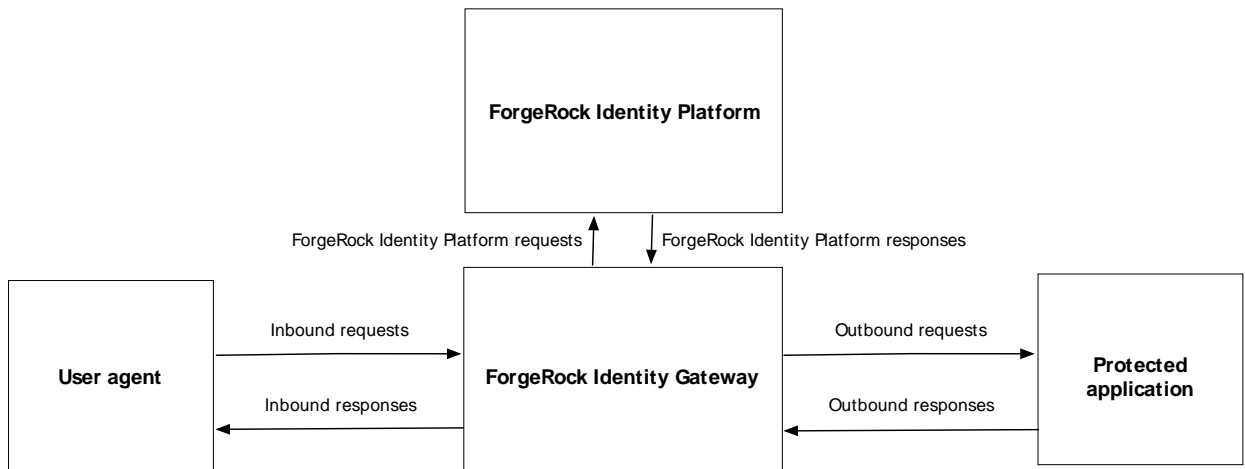
To Capture Messages on a Route in Studio

1. In Studio, select ROUTES and then select a route.

2. On the left side of the screen, select Q 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. The log file should contain the messages defined by your capture configuration.

19.4. Changing the Logging Behavior

The Logback configuration is very flexible, providing a wide range of options for logging. For a full description of its parameters, see the Logback website. The following examples show some simple changes that you can make.

To change the logging behavior, create a new Logback file at `$HOME/.openig/config/logback.xml`. This 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 interval:

```
<configuration scan="true" scanPeriod="5 seconds">
```

The configuration `scan="true"` requires `logback.xml` to be scanned for changes. The 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 scan period has elapsed, where the example specifies 5 seconds.

> **Note**
>
> If your 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 ...
> ```

19.4.1. Formatting Log Messages

You can format log messages in many ways. For example, to add the date to the log message, edit `logback.xml` to change the pattern of the log messages in the `encoder` part of the SIFT appender:

```
%d{yyyyMMdd-HH:mm:ss} | %-5level | %thread | %logger{20} | %message%n%xException
```

19.4.2. Logging for Different Object Types

You can change the log messages for a single object type without changing them for the rest of the configuration.

For example, to record log messages with the level `ERROR` or higher for the ClientHandler, edit `logback.xml` to add a logger defined by the fully qualified class name of the ClientHandler, and to set its logging level to `ERROR`:

```
<logger name="org.forgerock.openig.handler.ClientHandler" level="ERROR"/>
```

Log messages with a level lower than `ERROR` are no longer recorded for the ClientHandler but continue to be recorded for the rest of the configuration.

19.4.3. Logging for 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 logging 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
```

19.4.4. Logging for the BaseUriDecorator

During setup and configuration, it can be helpful to display log messages from the BaseUriDecorator.

For example, 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
```

19.4.5. Switching Off 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"/>
```

Chapter 20
Troubleshooting

This chapter covers common problems and their solutions.

Get the product version and build information for the running instance of IG from the `/api/info` endpoint. If IG is set up as described in "*First Steps*" in the *Getting Started Guide*, access the information at http://openig.example.com:8080/openig/api/info.

20.1. Timeout When Downloading Large Files

If `SocketTimeoutException` errors occur in the logs when you try to download large files, in your ReverseProxyHandler or ClientHandler, increase `soTimeout` and set `asyncBehavior` to `streaming`.

20.2. Requests Redirected to Access Management Instead of to the Resource

By default, ForgeRock Access Management 5 and later writes cookies to the fully qualified domain name of the server, for example, `openam.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 Access Management console, go to Configure > Global Services > Platform, and add the domain `example.com`.

20.3. Troubleshooting the UMA Example

You have set up and are testing the example in "About IG As an UMA Resource Server".

If you have problems creating shares for Alice, perform the following steps to see if you can get a PAT from AM:

1. With AM running, run the following command to get a tokenID:

```
$ curl --request POST
 \
--header "X-OpenAM-Username: alice"
 \
--header "X-OpenAM-Password: password"
 \
--header "Content-Type: application/json"
 \
--data "{}"  \
http://openam.example.com:8088/openam/json/authenticate

{"tokenId":"AQIC5wM2LY . . . Dg5AAJTMQAA*","successUrl":"/openam/console"}
```

2. In the following command, replace `tokenID` with the value you got in the previous step:

```
$ curl -X POST -H "Cache-Control: no-cache" -H "Cookie: iPlanetDirectoryPro=tokenID"
 \
-H "Content-Type: application/x-www-form-urlencoded"
 \
-d
 'grant_type=password&scope=uma_protection&username=alice&password=password&client_id=IG&client_secret=password
 \
http://openam.example.com:8088/openam/oauth2/access_token

{"access_token":"AQIC5wM2LY . . . Dg5AAJTMQAA*","scope":"uma_protection","token_type":"Bearer"
,"expires_in":3599}
```

An access token should be displayed. If you fail to get an access token, check that AM is configured as described in "Setting Up AM As an Authorization Server".

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/.

20.4. Can't Deploy Routes in 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 that it contains a main router named `_router`.

For information about Studio, see "Creating Routes Through Studio ".

20.5. Object not found in heap

```
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)
```

You have specified `"handler": "Router2"` in `config.json`, but no handler configuration object named Router2 exists. Make sure you have added an entry for the handler and that you have correctly spelled its name.

20.6. Extra or missing character / invalid JSON

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:

```
16:09:50 | ERROR | openig.example.com-startStop-1 | o.f.o.h.r.RouterHandler |
The file '/Users/me/.openig/config/routes/zz-default.json' is not a valid route configuration.
```

Use a JSON editor or JSON validation tool such as JSONLint to make sure that your JSON is valid.

20.7. The values in the flat file are incorrect

Ensure the flat file is readable by the user running the container for IG. Values are all characters including space and tabs between the separator, so make sure the values are not padded with spaces.

20.8. Problem accessing URL

```
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
```

This error is typically encountered when using an `AssignmentFilter` as described in AssignmentFilter(5) in the *Configuration Reference* and setting a string value for one of the headers. 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.

20.9. StaticResponseHandler results in a blank page

Define an entity for the response as in the following example:

```
{
    "name": "AccessDeniedHandler",
    "type": "StaticResponseHandler",
    "config": {
        "status": 403,
        "reason": "Forbidden",
        "entity": "<html><body><p>User does not have permission</p></body></html>"
    }
}
```

20.10. IG is not logging users in

If you are proxying to more than one application in multiple DNS domains, you must make sure your container is enabled for domain cookies. For details on your specific container, see "Configuring Deployment Containers".

20.11. Read timed out error when sending a request

If a baseURI configuration setting causes a request to come back to IG, IG never produces a response to the request. You then observe the following behavior.

You send a request and IG seems to hang. Then you see a failure message, `HTTP Status 500 - Read timed out`, accompanied by IG throwing an exception, `java.net.SocketTimeoutException: Read timed out`.

To fix this issue, make sure that `baseURI` configuration settings use a different host and port than the host and port for IG.

20.12. IG does not use new route configuration

IG loads all configuration at startup. By default, it then 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 see an error, `No handler to dispatch to`.

20.13. Make IG skip a route

If you have copied routes from another IG server, those routes might depend on environment or container configuration that you have not yet configured locally.

You can work around this problem by changing the route file extension. A router ignores route files that do not have the `.json` extension.

For example, suppose you copy route all sample route configurations from the documentation, and then start IG without first configuring your container. This can result in an error such as the following:

```
/handler/config/filters/0/config/dataSource: javax.naming.NameNotFoundException;
    remaining name 'jdbc/forgerock'
[      JsonValueException] > /handler/config/filters/0/config/dataSource:
    javax.naming.NameNotFoundException; remaining name 'jdbc/forgerock'
[   NameNotFoundException] > null

org.forgerock.json.fluent.JsonValueException:
    /handler/config/filters/0/config/dataSource:
    javax.naming.NameNotFoundException; remaining name 'jdbc/forgerock'
 at org.forgerock.openig.filter.SqlAttributesFilter$Heaplet.create(
    SqlAttributesFilter.java:211)
 at org.forgerock.openig.heap.GenericHeaplet.create(GenericHeaplet.java:81)
 at org.forgerock.openig.heap.HeapImpl.extract(HeapImpl.java:316)
 at org.forgerock.openig.heap.HeapImpl.get(HeapImpl.java:281)
...
```

This arises from the route in `03-sql.json`, which defines an SqlAttributesFilter that depends on a JNDI data source configured in the container:

```
{
    "type": "SqlAttributesFilter",
    "config": {
        "dataSource": "java:comp/env/jdbc/forgerock",
        "preparedStatement":
          "SELECT username, password FROM users WHERE email = ?;",
        "parameters": [
            "george@example.com"
        ],
        "target": "${attributes.sql}"
    }
}
```

To prevent IG from loading the route configuration until you have had time to configure the container, change the file extension to render the route inactive:

```
$ mv ~/.openig/config/routes/03-sql.json ~/.openig/config/routes/03-sql.inactive
```

If necessary, restart the container to force IG to reload the configuration.

When you have configured the data source in the container, change the file extension back to `.json` to render the route active again:

```
$ mv ~/.openig/config/routes/03-sql.inactive ~/.openig/config/routes/03-sql.json
```

Appendix A. SAML 2.0 and Multiple Applications

"*Acting As a SAML 2.0 Service Provider*" describes how to set up IG as a SAML 2.0 service provider for a single application, using AM as the identity provider. This chapter describes how to set up IG as a SAML 2.0 service provider for two applications, still using AM as the identity provider.

Before you try the samples described here, familiarize yourself with IG SAML 2.0 support by reading and working through the examples in "*Acting As a SAML 2.0 Service Provider*". Before you start, you should have IG protecting the sample application as a SAML 2.0 service provider, with AM working as identity provider configured as described in that tutorial.

A.1. Installation Overview

In this chapter you use the Fedlet configuration from "*Acting As a SAML 2.0 Service Provider*" to create a configuration for each new protected application. You then import the new configurations as SAML 2.0 entities in AM. If you subsequently edit a configuration, import it again.

In the following examples, the first application has entity ID `sp1` and runs on the host `sp1.example.com`, the second application has entity ID `sp2` and runs on the host `sp2.example.com`. To prevent unwanted behavior, the applications must have different values.

Tasks for Configuring SAML 2.0 SSO and Federation

Task	See Section(s)
Prepare the network.	"Preparing the Network"

Task	See Section(s)
Prepare the configuration for two IG service providers.	"Configuring the Circle of Trust" "Configuring the Service Provider for Application One" "Configuring the Service Provider for Application Two"
Import the service provider configurations into AM.	"Importing Service Provider Configurations Into AM"
Add IG routes.	"Preparing the Base Configuration File" "Preparing Routes for Application One" "Preparing Routes for Application Two"

A.2. Preparing the Network

Configure the network so that browser traffic to the application hosts is proxied through IG.

Add `sp1.example.com` and `sp2.example.com` to your `/etc/hosts` file:

```
127.0.0.1    localhost openam.example.com openig.example.com app.example.com sp1.example.com sp2.example
.com
```

A.3. Configuring the Circle of Trust

Edit the `$HOME/.openig/SAML/fedlet.cot` file created in "*Acting As a SAML 2.0 Service Provider*" to include the entity IDs `sp1` and `sp2`:

```
cot-name=Circle of Trust
sun-fm-cot-status=Active
sun-fm-trusted-providers=openam,sp1,sp2
sun-fm-saml2-readerservice-url=
sun-fm-saml2-writerservice-url=
```

A.4. Configuring the Service Provider for Application One

To configure the service provider for application one, you can use the example files "Configuration File for Application One" and "Extended Configuration File for Application One", saving them as `sp1.xml` and `sp1-extended.xml`. Alternatively, follow the steps below to use the files you created in "*Acting As a SAML 2.0 Service Provider*".

To Configure the Service Provider for Application One By Using Files Created In "Acting As a SAML 2.0 Service Provider"

1. Copy the SAML configuration files `sp.xml` and `sp-extended.xml` you created in "*Acting As a SAML 2.0 Service Provider*", and save them as `sp1.xml` and `sp1-extended.xml`.

2. Make the following changes in `sp1.xml`:

   - For `entityID`, change `sp` to `sp1`. The `entityID` must match the application.

   - On each line that starts with `Location` or `ResponseLocation`, change `sp.example.com` to `sp1.example.com`, and add `/metaAlias/sp1` at the end of the line.

   For an example of how this file should be, see "Configuration File for Application One".

3. Make the following changes in `sp1-extended.xml`:

   - For `entityID`, change `sp` to `sp1`.

   - For `SPSSOConfig metaAlias`, change `sp` to `sp1`.

   - For `appLogoutUrl`, change `sp` to `sp1`.

   - For `hosted=`, make sure that the value is `1`.

   For an example of how this file should be, see "Extended Configuration File for Application One".

Configuration File for Application One

```
<!--
  - sp1-xml.txt
  - Set the entityID
  - Set metaAlias/<sp-name> at the end of each of the following lines:
     - Location
     - ResponseLocation
  - Note that AssertionConsumerService Location attributes include the metaAlias.
-->
<EntityDescriptor
  entityID="sp1"
  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://sp1.example.com:8080/saml/fedletSloRedirect/metaAlias/sp1"
      ResponseLocation="http://sp1.example.com:8080/saml/fedletSloRedirect/metaAlias/sp1"/>
    <SingleLogoutService
      Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
      Location="http://sp1.example.com:8080/saml/fedletSloPOST/metaAlias/sp1"
```

```
        ResponseLocation="http://sp1.example.com:8080/saml/fedletSloPOST/metaAlias/sp1"/>
    <SingleLogoutService
      Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
      Location="http://sp1.example.com:8080/saml/fedletSloSoap/metaAlias/sp1"/>
    <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://sp1.example.com:8080/saml/fedletapplication/metaAlias/sp1"/>
    <AssertionConsumerService
      index="1"
      Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Artifact"
      Location="http://sp1.example.com:8080/saml/fedletapplication/metaAlias/sp1"/>
  </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>
```

Extended Configuration File for Application One

```
<!--
  - sp1-extended.xml
  - Set the entityID.
  - Set the SPSSOConfig metaAlias attribute.
  - Set the value of appLogoutUrl.
  - Set the value of hosted to 1.
  - Comment out the attribute "com.sun.identity.saml2.plugins.DefaultFedletAdapter".
-->
<EntityConfig xmlns="urn:sun:fm:SAML:2.0:entityconfig"
    xmlns:fm="urn:sun:fm:SAML:2.0:entityconfig"
    hosted="1"
    entityID="sp1">

    <SPSSOConfig metaAlias="/sp1">
        <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>com.sun.identity.saml2.plugins.DefaultFedletAdapter</Value>
    </Attribute>
    -->
    <Attribute name="fedletAdapterEnv">
        <Value></Value>
    </Attribute>
    <Attribute name="spAccountMapper">
        <Value>com.sun.identity.saml2.plugins.DefaultLibrarySPAccountMapper</Value>
    </Attribute>
    <Attribute name="useNameIDAsSPUserID">
        <Value>false</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>employeenumber=employeenumber</Value>
        <Value>mail=mail</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://sp1.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="responseArtifactMessageEncoding">
        <Value>URI</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="ECPRequestIDPListGetComplete">
        <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>
    <Attribute name="spSessionSyncEnabled">
        <Value>false</Value>
    </Attribute>
     <Attribute name="relayStateUrlList">
     </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>false</Value>
    </Attribute>
    <Attribute name="wantAssertionEncrypted">
        <Value>false</Value>
    </Attribute>
    <Attribute name="cotlist">
        <Value>Circle of Trust</Value>
    </Attribute>
</XACMLAuthzDecisionQueryConfig>
```

```
</EntityConfig>
```

A.5. Configuring the Service Provider for Application Two

To Configure the Service Provider for Application Two

1. Copy the SAML configuration files `sp1.xml` and `sp1-extended.xml` you created in "Configuring the Service Provider for Application One", and save them as `$HOME/.openig/SAML/sp2.xml` and `$HOME/.openig/SAML/sp2-extended.xml`.

2. In both files, replace all incidences of `sp1` with `sp2`. To prevent unwanted behavior. application two must have different values to application one.

A.6. Importing Service Provider Configurations Into AM

For each new protected application, import a SAML 2.0 entity into AM. If you subsequently edit a service provider configuration, import it again.

To Import the Service Provider Configurations Into AM

1. Log in to AM console as administrator.

2. Select Applications > WS-Fed, and click Import Entity.

   In some versions of AM, this option is on a Federation tab.

3. Import the entity provider:

   • For the metadata file, select File and upload `sp1.xml`.

   • For the extended data file, select File and upload `sp1-extended.xml`.

4. Repeat the previous steps to upload `sp2.xml` and `sp2-extended.xml` for `sp2`.

5. Log out of the AM console.

A.7. Preparing IG Configurations

For each new protected application, prepare an IG configuration. The configurations in this section follow the example in "*Acting As a SAML 2.0 Service Provider*".

> **Tip**
>
> To prevent unspecified behavior, use different keys for session-stored values in the routes for each application. For example, use different keys for `session.sp1Username` and `session.sp2Username`.

A.7.1. Preparing the Base Configuration File

Edit `config.json` to comment the baseURI in the top-level handler. The handler declaration appears as follows:

```
{
    "handler": {
        "type": "Router",
        "name": "_router",
        "_baseURI": "http://app.example.com:8081",
        "capture": "all"
    }
}
```

Restart IG for the changes to take effect.

A.7.2. Preparing Routes for Application One

Set up the following routes for application one:

- `$HOME/.openig/config/routes/05-federate-sp1.json`, to redirect the request for SAML authentication. After authentication, this route logs the user in to the application.

- `$HOME/.openig/config/routes/05-saml-sp1.json`, to map attributes from the SAML assertion into the context, and then redirect the request back to the first route.

05-federate-sp1.json

```
{
    "handler": {
        "type": "DispatchHandler",
        "config": {
            "bindings": [
                {
                    "condition": "${empty session.sp1Username}",
                    "handler": {
                        "type": "StaticResponseHandler",
                        "config": {
                            "status": 302,
                            "reason": "Found",
                            "headers": {
                                "Location": [
                                    "http://sp1.example.com:8080/saml/SPInitiatedSSO?metaAlias=/sp1"
                                ]
                            }
                        }
                    }
```

```
                    }
                },
                {
                    "handler": {
                        "type": "Chain",
                        "config": {
                            "filters": [
                                {
                                    "type": "StaticRequestFilter",
                                    "config": {
                                        "method": "POST",
                                        "uri": "http://app.example.com:8081/login",
                                        "form": {
                                            "username": [
                                                "${session.sp1Username[0]}"
                                            ],
                                            "password": [
                                                "${session.sp1Password[0]}"
                                            ]
                                        }
                                    }
                                }
                            ],
                            "handler": "ReverseProxyHandler"
                        }
                    }
                }
            ]
        }
    },
    "condition": "${matches(request.uri.host, 'sp1.example.com') and not matches(request.uri.path, '^/
saml')}"
}
```

05-saml-sp1.json

```
{
    "handler": {
        "type": "SamlFederationHandler",
        "config": {
            "comment": "Use unique session properties for this SP.",
            "assertionMapping": {
                "sp1Username": "mail",
                "sp1Password": "employeenumber"
            },
            "authnContext": "sp1AuthnContext",
            "sessionIndexMapping": "sp1SessionIndex",
            "subjectMapping": "sp1SubjectName",
            "redirectURI": "/sp1"
        }
    },
    "condition": "${matches(request.uri.host, 'sp1.example.com') and matches(request.uri.path, '^/saml')}"
}
```

A.7.3. Preparing Routes for Application Two

Set up the following routes for application two:

- `$HOME/.openig/config/routes/05-federate-sp2.json`, to redirect the request for SAML authentication. After authentication, this route logs the user in to the application.

- `$HOME/.openig/config/routes/05-saml-sp2.json`, to map attributes from the SAML assertion into the context, and then redirect the request back to the first route.

05-federate-sp2.json

```
{
    "handler": {
        "type": "DispatchHandler",
        "config": {
            "bindings": [
                {
                    "condition": "${empty session.sp2Username}",
                    "handler": {
                        "type": "StaticResponseHandler",
                        "config": {
                            "status": 302,
                            "reason": "Found",
                            "headers": {
                                "Location": [
                                    "http://sp2.example.com:8080/saml/SPInitiatedSSO?metaAlias=/sp2"
                                ]
                            }
                        }
                    }
                },
                {
                    "handler": {
                        "type": "Chain",
                        "config": {
                            "filters": [
                                {
                                    "type": "StaticRequestFilter",
                                    "config": {
                                        "method": "POST",
                                        "uri": "http://app.example.com:8081/login",
                                        "form": {
                                            "username": [
                                                "${session.sp2Username[0]}"
                                            ],
                                            "password": [
                                                "${session.sp2Password[0]}"
                                            ]
                                        }
                                    }
                                }
                            ],
                            "handler": "ReverseProxyHandler"
                        }
```

```
                }
            }
        ]
    }
},
"condition": "${matches(request.uri.host, 'sp2.example.com') and not matches(request.uri.path, '^/
saml')}"
}
```

05-saml-sp2.json

```
{
    "handler": {
        "type": "SamlFederationHandler",
        "config": {
            "comment": "Use unique session properties for this SP.",
            "assertionMapping": {
                "sp2Username": "mail",
                "sp2Password": "employeenumber"
            },
            "authnContext": "sp2AuthnContext",
            "sessionIndexMapping": "sp2SessionIndex",
            "subjectMapping": "sp2SubjectName",
            "redirectURI": "/sp2"
        }
    },
    "condition": "${matches(request.uri.host, 'sp2.example.com') and matches(request.uri.path, '^/saml')}"
}
```

A.8. Test the Configuration

If you use the example configurations described in this chapter, try the SAML 2.0 web single sign-on profile with application one by selecting either of the following links and logging in to AM with username george and password costanza:

• The link for SP-initiated SSO.

• The link for IDP-initiated SSO.

Similarly, try the SAML 2.0 web single sign-on profile with application two by selecting either of the following links and logging in to AM with username george and password costanza:

• The link for SP-initiated SSO.

• The link for IDP-initiated SSO.

If you have not configured the examples exactly as shown in this guide, then adapt the SSO links accordingly.