



# Gateway Guide

/ ForgeRock Identity Gateway 7

Latest update: 7.0.2

Paul Bryan  
Mark Craig  
Jamie Nelson  
Guillaume Sauthier  
Joanne Henry

ForgeRock AS.  
201 Mission St., Suite 2900  
San Francisco, CA 94105, USA  
+1 415-599-1100 (US)  
[www.forgerock.com](http://www.forgerock.com)

---

Copyright © 2011-2021 ForgeRock AS.

## Abstract

### Instructions for installing and configuring ForgeRock® Identity Gateway.



This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.

To view a copy of this license, visit <https://creativecommons.org/licenses/by-nc-nd/3.0/> or send a letter to Creative Commons, 444 Castro Street, Suite 900, Mountain View, California, 94041, USA.

ForgeRock® and ForgeRock Identity Platform™ are trademarks of ForgeRock Inc. or its subsidiaries in the U.S. and in other countries. Trademarks are the property of their respective owners.

UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

DejaVu Fonts

Bitstream Vera Fonts Copyright

Copyright (c) 2003 by Bitstream, Inc. All Rights Reserved. Bitstream Vera is a trademark of Bitstream, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Bitstream" or the word "Vera".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Bitstream Vera" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL BITSTREAM OR THE GNOME FOUNDATION BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the names of Gnome, the Gnome Foundation, and Bitstream Inc., shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from the Gnome Foundation or Bitstream Inc., respectively. For further information, contact: [fonts at gnome dot org](mailto:fonts at gnome dot org).

Arev Fonts Copyright

Copyright (c) 2006 by Tavmjong Bah. All Rights Reserved.

Permission is hereby granted, free of charge, to any person obtaining a copy of the fonts accompanying this license ("Fonts") and associated documentation files (the "Font Software"), to reproduce and distribute the modifications to the Bitstream Vera Font Software, including without limitation the rights to use, copy, merge, publish, distribute, and/or sell copies of the Font Software, and to permit persons to whom the Font Software is furnished to do so, subject to the following conditions:

The above copyright and trademark notices and this permission notice shall be included in all copies of one or more of the Font Software typefaces.

The Font Software may be modified, altered, or added to, and in particular the designs of glyphs or characters in the Fonts may be modified and additional glyphs or characters may be added to the Fonts, only if the fonts are renamed to names not containing either the words "Tavmjong Bah" or the word "Arev".

This License becomes null and void to the extent applicable to Fonts or Font Software that has been modified and is distributed under the "Tavmjong Bah Arev" names.

The Font Software may be sold as part of a larger software package but no copy of one or more of the Font Software typefaces may be sold by itself.

THE FONT SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT OF COPYRIGHT, PATENT, TRADEMARK, OR OTHER RIGHT. IN NO EVENT SHALL TAVMJONG BAH BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, INCLUDING ANY GENERAL, SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF THE USE OR INABILITY TO USE THE FONT SOFTWARE OR FROM OTHER DEALINGS IN THE FONT SOFTWARE.

Except as contained in this notice, the name of Tavmjong Bah shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Font Software without prior written authorization from Tavmjong Bah. For further information, contact: [tavmjong @ free . fr](mailto:tavmjong @ free . fr).

FontAwesome Copyright

Copyright (c) 2017 by Dave Gandy, <https://fontawesome.com/>.

This Font Software is licensed under the SIL Open Font License, Version 1.1. See <https://opensource.org/licenses/OFL-1.1>.

---

# Table of Contents

Preface .....	vi
About This Guide .....	vi
Example Installation for This Guide .....	vii
External Tools Used In This Guide .....	vii
1. About IG .....	1
IG As an HTTP Gateway .....	1
Processing Requests and Responses .....	3
Development Mode and Production Mode .....	7
Decorators .....	8
Configuration Parameters Declared as Property Variables .....	14
Changing the Configuration and Restarting IG .....	14
Understanding IG APIs With API Descriptors .....	15
Sessions .....	18
Secrets .....	20
2. Installation in Detail .....	25
About Securing Connections .....	25
Installing IG in Standalone Mode .....	26
Installing IG in Apache Tomcat .....	29
Installing IG in Jetty .....	32
Installing IG in JBoss EAP .....	37
Preparing the Network .....	40
Changing the Default Location of the Configuration Folders .....	40
Preparing For Load Balancing and Failover .....	40
Configuring IG For HTTPS (Client-Side) .....	42
Using JWT Sessions .....	45
Setting Up AM .....	54
3. Getting Login Credentials From Data Sources .....	57
Logging In With Credentials From a File .....	57
Logging In With Credentials From a Database .....	60
4. Getting Login Credentials From AM .....	65
5. Single Sign-On and Cross-Domain Single Sign-On .....	71
Authenticating With SSO .....	71
Authenticating With CDSSO .....	76
Using WebSocket Notifications to Evict the Session Info Cache .....	81
6. Enforcing Policy Decisions From AM .....	82
About Policy Enforcement .....	82
Enforcing AM Policy Decisions In the Same Domain .....	83
Enforcing AM Policy Decisions In Different Domains .....	86
Using WebSocket Notifications to Evict the Policy Cache .....	89
7. Hardening Authorization With Advice From AM .....	90
Stepping Up the Authentication Level for an AM Session .....	90
Increasing Authorization for a Single Transaction .....	94
8. Protecting Against CSRF Attacks .....	98
9. Acting As a SAML 2.0 Service Provider .....	103

About SAML 2.0 SSO and Federation .....	103
Set Up SAML 2.0 SSO and Federation .....	105
Using a Non-Transient NameID Format .....	112
Example Fedlet Files .....	113
10. Acting As an OAuth 2.0 Resource Server .....	124
About IG As an OAuth 2.0 Resource Server .....	124
Validating Access_Tokens Through the Introspection Endpoint .....	126
Validating Stateless Access_Tokens With the StatelessAccessTokenResolver .....	130
Validating Certificate-Bound Access Tokens .....	144
Using the OAuth 2.0 Context to Log in to the Sample Application .....	163
Caching Access_Tokens .....	166
11. Acting As an OpenID Connect Relying Party .....	170
About IG With OpenID Connect .....	170
Using AM As a Single OpenID Connect Provider .....	171
Using Multiple OpenID Connect Providers .....	177
Discovering and Dynamically Registering With OpenID Connect Providers .....	180
12. Transforming OpenID Connect ID Tokens Into SAML Assertions .....	186
13. Supporting UMA Resource Servers .....	194
About IG As an UMA Resource Server .....	194
Limitations Of IG As an UMA Resource Server .....	197
Setting Up the UMA Example .....	198
Editing the Example to Match Custom Settings .....	206
Understanding the UMA API With an API Descriptor .....	207
14. Configuring Routers and Routes .....	208
Configuring Routers .....	208
Configuring Routes .....	209
Creating and Editing Routes Through Common REST .....	212
Preventing the Reload of Routes .....	214
Accessing Reserved Routes .....	215
15. Proxying WebSocket Traffic .....	216
16. Implementing Not-Enforced URIs for Authentication .....	221
Implementing Not-Enforced URIs With a SwitchFilter .....	221
Implementing Not-Enforced URIs With a DispatcherHandler .....	225
17. Configuration Templates .....	227
Proxy and Capture .....	227
Simple Login Form .....	229
Login Form With Cookie From Login Page .....	231
Login Form With Password Replay and Cookie Filters .....	233
Login Which Requires a Hidden Value From the Login Page .....	235
HTTP and HTTPS Application .....	237
AM Integration With Headers .....	239
18. Extending IG .....	242
Extending IG Through Scripts .....	242
Extending IG Through the Java API .....	256
Recording Custom Audit Events .....	262
19. Throttling the Rate of Requests to Protected Applications .....	269
About Throttling .....	269

Configuring Simple Throttling .....	270
Configuring Mapped Throttling .....	272
Configuring Scriptable Throttling .....	277
20. SAML 2.0 and Multiple Applications .....	281

# 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>.

## 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, IG enforces security and access control in conjunction with 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

## Example Installation for This Guide

Unless otherwise stated, the examples in this guide assume the following installation:

- IG installed on <http://openig.example.com:8080>, as described in "*Downloading and Starting IG*" in the *Getting Started Guide*.
- Sample application installed on <http://openig.example.com:8081>, as described in "*Downloading and Starting the Sample Application*" in the *Getting Started Guide*.
- AM installed on <http://openam.example.com:8088/openam>, with the default configuration.

If you use a different configuration, substitute in the procedures accordingly.

## External Tools Used In This Guide

The examples in this guide use some of the following third-party tools:

- **curl**: <https://curl.haxx.se>
- **HTTPIe**: <https://httpie.org>
- **jq**: <https://stedolan.github.io/jq/>
- **keytool**: <https://docs.oracle.com/en/java/javase/11/tools/keytool.html>

## Chapter 1

# About IG

The following sections introduce IG:

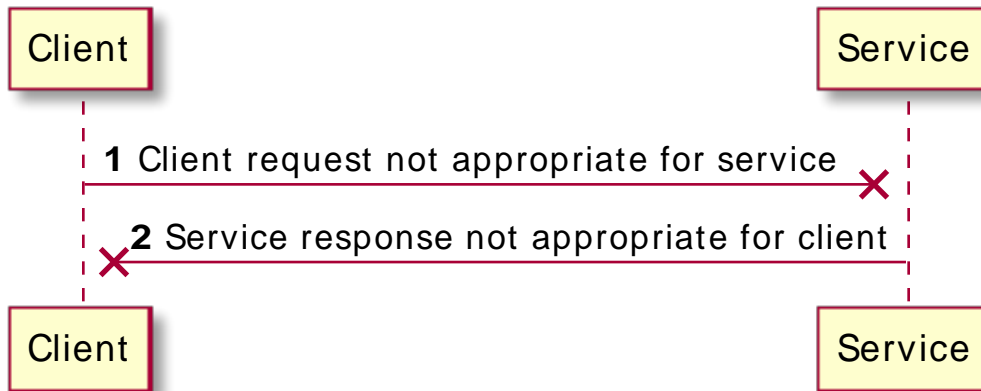
- "IG As an HTTP Gateway"
- "Processing Requests and Responses"
- "Development Mode and Production Mode"
- "Decorators"
- " Configuration Parameters Declared as Property Variables "
- "Changing the Configuration and Restarting IG"
- "Understanding IG APIs With API Descriptors"
- "Sessions"
- "Secrets"

## IG As an HTTP Gateway

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. The following image illustrates an example of a missing gateway.



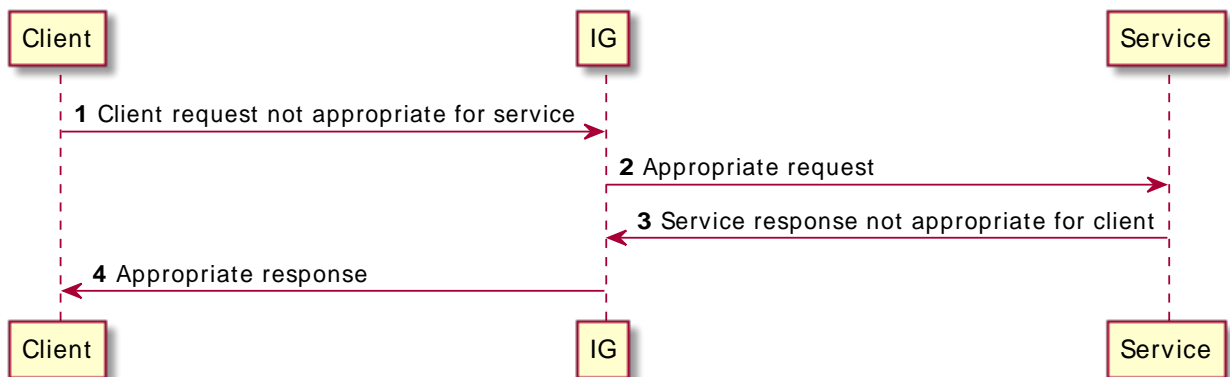
## Missing Gateway



IG works as an HTTP gateway, based on reverse proxy architecture. IG is deployed on a network, so that it can intercept client requests and server responses. IG can check the identity of HTTP traffic, blocking requests without permission, and letting allowed requests pass. IG can also adapt requests and responses. For example, IG can add headers and change the message payload.

The following image illustrates how a request and response flow between a client and application:

## 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.

IG provides the following features:

- 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.

## Processing Requests and Responses

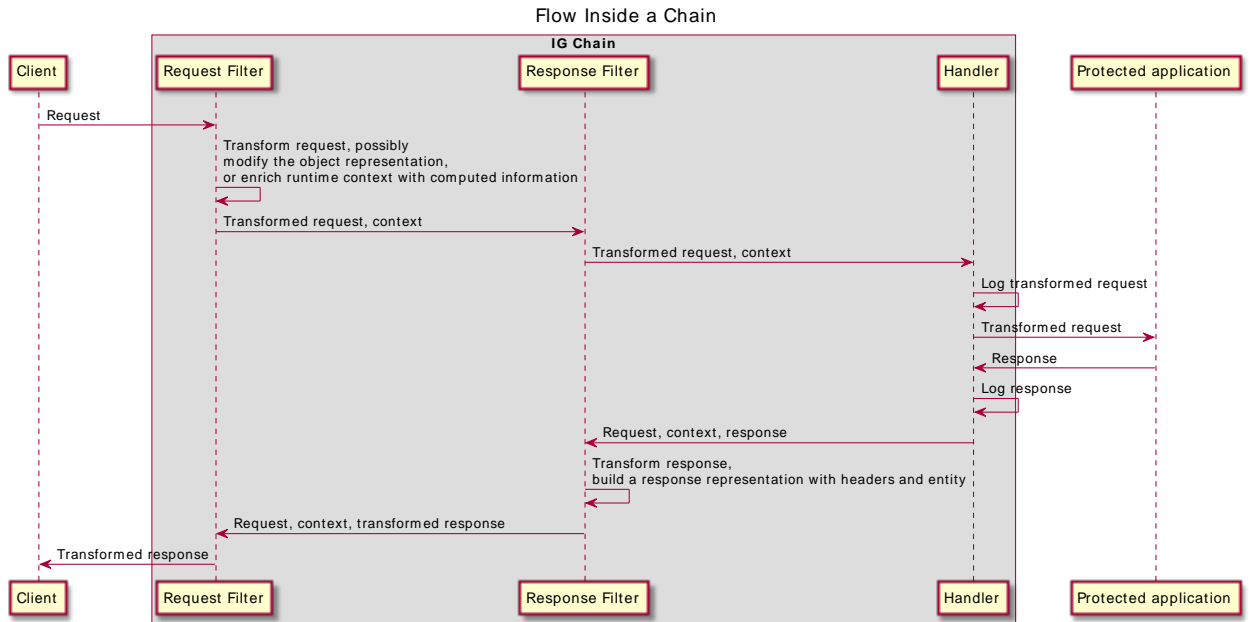
The following sections describe how IG processes requests and responses:

- "IG Object Model"
- "Configuring IG"
- "Configuration Directories and Files"
- "Using Comments in IG Configuration Files"

### IG Object Model

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

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



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

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

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

The route configuration in "Adding Headers and Logging Results" in the *Configuration Reference* demonstrates the flow through a chain to a protected application.

## Configuring IG

The way that IG processes requests and responses is defined by the configuration files `admin.json` and `config.json`, and by Route configuration files. For information about the different files used by IG, see "Configuration Directories and Files".

Configuration files are flat JSON files that define objects with the following parts:

- **name**: A unique string to identify the object. When you declare inline objects, the name is not required.
- **type**: The type name of the object. IG defines many object types for different purposes.
- **config**: Additional configuration settings for the object. The content of the configuration object depends on its type. For information about each object type available in the IG configuration, see the *Configuration Reference*.

If all of the configuration settings for the type are optional, the **config** field is also optional. The object uses default settings when the **config** field isn't configured, or is configured as an empty object (`"config": {}`), or is configured as null (`"config": null`).

Filters, handlers, and other objects whose configuration settings are defined by strings, integers, or booleans, can be defined by expressions that match the expected type.

For information about the objects available, see "AdminHttpApplication (`admin.json`)" in the *Configuration Reference*, "GatewayHttpApplication (`config.json`)" in the *Configuration Reference* and "Route" in the *Configuration Reference*. An IG route typically contains at least the following parts:

- **handler**: An object to specify the point where the request enters the route. If the handler type is a Chain, the request is dispatched to a list of filters, and then to another handler.

Handler objects produce a response for a request, or delegate the request to another handler. Filter objects transform data in the request, response, or context, or perform an action when the request or response passes through the filter.

- **baseURI**: A handler decorator to override the scheme, host, and port of the request URI. After a route processes a request, it reroutes the request to the **baseURI**, which most usually points to the application or service that IG is protecting.
- **heap**: A collection of named objects configured in the top level of `config.json` or in individual routes. Heap objects can be configured once and used multiple times in the configuration.

A heap object in a route can be used in that route. A heap object in `config.json` can be used across the whole configuration, unless it is overridden in a route.

- **condition**: An object to define a condition that a request must meet. A route can handle a request if **condition** is not defined, or if the condition resolves to `true`.

Routes inherit settings from their parent configurations. This means that you can configure objects in the `config.json` heap, for example, and then reference those objects by name in any other IG configuration.

## 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:

### IG Configuration Directories and Files

Purpose	Default location on Linux, macOS, and UNIX systems	Default location on Windows
Administration and gateway configuration, <code>admin.json</code> and <code>config.json</code> . For more information, see "AdminHttpApplication ( <code>admin.json</code> )" in the <i>Configuration Reference</i> and "GatewayHttpApplication ( <code>config.json</code> )" in the <i>Configuration Reference</i>	<code>\$HOME/.openig/config</code>	<code>%appdata%\OpenIG\config</code>
Route configuration files. For more information, see " <i>Configuring Routers and Routes</i> ".	<code>\$HOME/.openig/config/routes</code>	<code>%appdata%\OpenIG\config\routes</code>
SAML 2.0 configuration files. For more information, see " <i>Acting As a SAML 2.0 Service Provider</i> ".	<code>\$HOME/.openig/SAML</code>	<code>%appdata%\OpenIG\SAML</code>
Script files, for Groovy scripted filters and handlers. For more information, see " <i>Extending IG</i> ".	<code>\$HOME/.openig/scripts/groovy</code>	<code>%appdata%\OpenIG\scripts\groovy</code>
Temporary storage files.  To change the directory, configure the property <code>temporaryDirectory</code> in <code>admin.json</code> . For more information, see "AdminHttpApplication ( <code>admin.json</code> )" in the <i>Configuration Reference</i> .	<code>\$HOME/.openig/tmp</code>	<code>%appdata%\OpenIG\tmp</code>
JSON schema for the topic of a custom audit event. For more information, see "Recording Custom Audit Events".  To change the directory, configure the property <code>topicsSchemasDirectory</code> in <code>AuditService</code> . For more information, see "AuditService" in the <i>Configuration Reference</i> .	<code>\$HOME/.openig/audit-schemas</code>	<code>%appdata%\OpenIG\audit-schemas</code>

## 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.

The following conventions are available for commenting:

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

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

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

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

## Development Mode and Production Mode

IG operates in the following modes:

- **Development mode (mutable mode)**

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

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

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

For information about how to switch to development mode, see "Switching from Production Mode to Development Mode" in the *Getting Started Guide*.

For information about restricting access to Studio in development mode, see "*Restricting Access to Studio*" in the *Studio User Guide*.

- **Production mode (immutable mode)**

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 `/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.

For information about how to switch to production mode, see "Switching From Development Mode to Production Mode" in the *Maintenance Guide*.

After installation, IG is by default in production mode.

## Decorators

Decorators are heap objects to extend what another object can do. IG defines `baseURI`, `capture`, and `timer` decorators that you can use without explicitly configuring them. For more information about the types of decorators provided by IG, see "*Decorators*" in the *Configuration Reference*.

The following sections provide an overview of how decorators are implemented in IG:

- "Decorating Objects, the Route Handler, and the Heap"
- "Using Multiple Decorators for the Same Object"
- "Guidelines for Naming Decorators"

### Decorating Objects, the Route Handler, and the Heap

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

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

1. Decorations declared on individual objects. Local decorations that are part of an object's declaration are inherited wherever the object is used.
2. `globalDecorations` declared in parent routes, then in child routes, and then in the current route.
3. Decorations declared on the route handler.

The following sections describes where to place decorators in a route.

## Decorating Individual Objects In a Route

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

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

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://openam.example.com:8088/openam/",
        "version": "7"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "capture": "all",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ]
    }
  },
  "handler": "ReverseProxyHandler"
}
```

## Decorating the Route Handler

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

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

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    }
  ],
```



```

{
  "name": "AmService-1",
  "type": "AmService",
  "config": {
    "agent": {
      "username": "ig_agent",
      "passwordSecretId": "agent.secret.id"
    },
    "secretsProvider": "SystemAndEnvSecretStore-1",
    "url": "http://openam.example.com:8088/openam/",
    "version": "7"
  }
},
"capture": "all",
"handler": {
  "type": "Chain",
  "config": {
    "filters": [
      {
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      }
    ],
    "handler": "ReverseProxyHandler"
  }
}
}

```

## Decorating the Route Heap

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

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

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

```

{
  "globalDecorators":
  {
    "capture": "all",
    "timer": true
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {

```

```

    "name": "AmService-1",
    "type": "AmService",
    "config": {
      "agent": {
        "username": "ig_agent",
        "passwordSecretId": "agent.secret.id"
      },
      "secretsProvider": "SystemAndEnvSecretStore-1",
      "url": "http://openam.example.com:8088/openam/",
      "version": "7"
    }
  },
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ]
    }
  },
  "handler": "ReverseProxyHandler"
}
}
}
}

```

## Decorating a Named Object Differently In Different Parts of the Configuration

When a filter or handler is configured in `config.json` or in the heap, it can be used many times in the configuration. To decorate each use of the filter or handler individually, use a Delegate. For more information, see "Delegate" in the *Configuration Reference*

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

```

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

```

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

## Decorating IG's Interactions With AM

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

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

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

To try the example, replace the `accessTokenResolver` in the IG route of "Validating Access\_Tokens Through the Introspection Endpoint". Test the setup as described for the example, and note that the route's log file contains an HTTP call to the introspection endpoint.

## Using Multiple Decorators for the Same Object

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

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

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

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

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

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

```
2018-09-10T13:23:18,990Z | INFO | http-nio-8080-exec-1 | o.f.o.d.c.C.c.top-level-handler | @myroute |
--- (request) id:f792d2ad-4409-4907-bc46-28e1c3c19ac3-7 --->
GET http://openig.example.com:8080/myroute HTTP/1.1
...
```

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

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

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

```
2018-09-10T13:07:07,524Z | INFO | http-nio-8080-exec-1 | o.f.o.d.c.c.top-level-handler | @myroute |  
--- (request) id:3c26ab12-3cc0-403e-bec6-43bf5621f657-7 --->  
GET http://app.example.com:8081/myroute HTTP/1.1  
...
```

## Guidelines for Naming Decorators

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

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

## 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*" in the *Configuration Reference*.

## Changing the Configuration and Restarting IG

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

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

- Change the configuration of any route, when the `scanInterval` of Router is `disabled` (see "Router" in the *Configuration Reference*).

- Add or change an external object used by the route, such as an environment variable, system property, external URL, or keystore.
- Add or update `config.json` or `admin.json`.
- When IG is running in web container mode, and the container configuration is changed.

## 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 `$HOME/.openig/logs/route-system.log`, where `$HOME/.openig` is the instance directory. Endpoint locations for subroutes are written to other log files. To retrieve the API descriptor for a specific endpoint, append one of the following query string parameters to the endpoint:

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

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

- `_crestapi`, to provide a compact representation that is independent of the transport protocol. This ForgeRock® Common REST (Common REST) API descriptor 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 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 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.



# Sessions

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

The following table compares stateful and stateless sessions:

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

## About Stateful Sessions

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

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

When a request enters a route without a `JwtSession` object in the route or its ascending configuration, a stateful session is created lazily. The session lasts as follows:

- For IG in standalone mode, the duration defined the `session` property in `admin.json`, defaulting to 30 minutes. For more information, see the ``session`` property of "AdminHttpApplication (`admin.json`)" in the *Configuration Reference*.

- For IG in web container mode, until the session reaches the timeout configured by the web container.

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

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

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

Session information is available in `SessionContext` to downstream handlers and filters. For more info see "SessionContext" in the *Configuration Reference*.

### Considerations for clustering IG

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

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

## About Stateless Sessions

Stateless sessions are provided when a `JwtSession` object is configured in `config.json` or in a route. For more information about configuring stateless sessions, see "JwtSession" in the *Configuration Reference*.

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

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

Stateless sessions are managed as follows:

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

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

Because the session content is stored in a cookie on the user agent, stateless sessions can be shared easily between IG instances. The cookie is automatically carried over in requests, and any IG instance can unpack and use the session content.

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

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

Session information is available in `SessionContext` to downstream handlers and filters,. For more information, see "`SessionContext`" in the *Configuration Reference*.

## Secrets

IG uses the ForgeRock Commons Secrets Service to manage secrets, such as passwords and cryptographic keys.

Repositories of secrets are managed through secret stores, provided to the configuration by the `SecretsProvider` object or `secrets` object. For more information about these objects and the types of secret stores provided in IG, see "`SecretsProvider`" in the *Configuration Reference* and "`Secret Stores`" in the *Configuration Reference*.

## Secret Names and Types

The following terms are used to describe secrets:

- **Secret ID:** A label to indicate the purpose of a secret. A secret ID is generally associated with one or more aliases of a key in a keystore or HSM.

- **Stable ID:** A label to identify a secret. The stable ID corresponds to the following values in each type of secret store:
  - Base64EncodedSecretStore: The value of `secret-id` in the `"secret-id": "string"` pair.
  - FileSystemSecretStore: The filename of a file in the specified directory, without the prefix/suffix defined in the store configuration.
  - HsmSecretStore: The value of an `alias` in a `secret-id/aliases` mapping.
  - JwkSetSecretStore: The value of the `kid` of a JWK stored in a JwkSetSecretStore.
  - KeyStoreSecretStore: The value of an `alias` in a `secret-id/aliases` mapping.
  - SystemAndEnvSecretStore: The name of a system property or environment. variable
- **Valid secret:** A secret whose purpose matches the secret ID.
- **Named secret:** A valid secret that a secret store can find by using a secret ID and stable ID.
- **Active secret:** One of the valid secrets that is considered eligible at the time of use.

## Validating the Signature of Signed Tokens

IG validates the signature of signed tokens as follows:

- Named secret resolution:
  - If the JWT contains a `kid`, IG queries the secret stores declared in `secretsProvider` or `secrets` to find a named secret, identified by a secret ID and stable ID.
  - If a named secret is found, IG then uses the named secret to try to validate the signature. If the named secret can't validate the signature, the token is considered as invalid.
  - If a named secret isn't found, IG tries valid secret resolution.
- Valid secret resolution:
  - IG uses the value of `verificationSecretId` as the secret ID, and queries the declared secret stores to find all secrets that match the provided secret ID.
  - All matching secrets are returned as valid secrets, in the order that the secret stores are declared, and for KeyStoreSecretStore and HsmSecretStore, in the order defined by the mappings.
  - IG tries to verify the signature with each valid secret, starting with the first valid secret, and stopping when it succeeds.
  - If no valid secrets are returned, or if none of the valid secrets can verify the signature, the token is considered as invalid.

## Validating the Signature of Signed Tokens by Using a KeyStoreSecretStore

In the following example, a `StatelessAccessTokenResolver` validates a signed `access_token` by using a `KeyStoreSecretStore`:

```
"accessTokenResolver": {
  "type": "StatelessAccessTokenResolver",
  "config": {
    "secretsProvider": {
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "IG_keystore.p12",
        "storeType": "PKCS12",
        "storePassword": "keystore.secret.id",
        "keyEntryPassword": "keystore.secret.id",
        "mappings": [{
          "secretId": "verification.secret.id",
          "aliases": [ "verification.key.1", "verification.key.2" ]
        }]
      }
    },
    "issuer": "http://openam.example.com:8088/openam/oauth2",
    "verificationSecretId": "verification.secret.id"
  }
}
```

The JWT signature is validated as follows:

- If the JWT contains a `kid` with a mapped value, for example `verification.key.1`:
  - The secrets provider queries the `KeyStoreSecretStore` for a named secret with the secret ID `verification.secret.id` and the stable ID `verification.key.1`.
  - Because the `KeyStoreSecretStore` contains that mapping, the `KeyStoreSecretStore` returns a named secret.
  - The `StatelessAccessTokenResolver` tries to validate the JWT signature with the named secret. If it fails, the token is considered as invalid.
- If the JWT contains a `kid` with an unmapped value, for example, `verification.key.3`:
  - The secrets provider queries the `KeyStoreSecretStore` for a named secret with the secret ID `verification.secret.id` and the stable ID `verification.key.3`.
  - Because the `KeyStoreSecretStore` doesn't contain that mapping, named secret resolution fails. IG tries valid secret resolution in the same way as when the JWT doesn't contain a `kid`.
- If the JWT doesn't contain a `kid`:
  - The secrets provider queries the `KeyStoreSecretStore` for all valid secrets, whose alias is mapped to the secret ID `verification.secret.id`. There are two valid secrets, with aliases `verification.key.1` and `verification.key.2`.

- The `StatelessAccessTokenResolver` first tries to verify the signature with `verification.key.1`. If that fails, it tries `verification.key.2`.
- If neither of the valid secrets can verify the signature, the token is considered as invalid.

### Validating the Signature of Signed Tokens With a `JwkSetSecretStore`

In the following example, a `StatelessAccessTokenResolver` validates a signed `access_token` by using a `JwkSetSecretStore`:

```
"accessTokenResolver": {
  "type": "StatelessAccessTokenResolver",
  "config": {
    "secretsProvider": {
      "type": "JwkSetSecretStore",
      "config": {
        "jwkUrl": "http://openam.example.com:8088/openam/oauth2/connect/jwk_uri"
      },
      "issuer": "http://openam.example.com:8088/openam/oauth2",
      "verificationSecretId": "verification.secret.id"
    }
  }
}
```

The JWT signature is validated as follows:

- If the JWT contains a `kid` with a matching secret in the JWK set:
  - The secrets provider queries the `JwkSetSecretStore` for a named secret.
  - The `JwkSetSecretStore` returns the matching secret, identified by a stable ID.
  - The `StatelessAccessTokenResolver` tries to validate the signature with that named secret. If it fails, the token is considered as invalid.

In the route, note that the property `verificationSecretId` must be configured but is not used in named secret resolution.

- If the JWT contains a `kid` without a matching secret in the JWK set:
  - The secrets provider queries the `JwkSetSecretStore` for a named secret.
  - Because the referenced JWK set doesn't contain a matching secret, named secret resolution fails. IG tries valid secret resolution in the same way as when the JWT doesn't contain a `kid`.
- If the JWT doesn't contain a `kid`:
  - The secrets provider queries the `JwkSetSecretStore` for list of valid secrets, whose secret ID is `verification.secret.id`.
  - The `JwkSetSecretStore` returns all secrets in the JWK set whose purpose is signature verification. For example, signature verification keys can have the following JWK parameters:

```
{  
  "use": "sig"  
}
```

```
{  
  "key_opts": [ "verify" ]  
}
```

Secrets are returned in the order that they are listed in the JWK set.

- The `StatelessAccessTokenResolver` tries to validate the signature with each secret sequentially, starting with the first, and stopping when it succeeds.
- If none of the valid secrets can verify the signature, the token is considered as invalid.

## Using Multiple Secret Stores in a Configuration

When multiple secrets stores are provided in a configuration, the secrets stores are queried in the following order:

- Locally in the route, starting with the first secret store in the list, up to the last.
- In ascending parent routes, starting with the first secret store in each list, up to the last.
- In `config.json`, starting with the first secret store in the list, up to the last.
- If a secrets store is not configured in `config.json`, the secret is queried in a default `SystemAndEnvSecretStore`, and a base64-encoded value is expected.
- If a secret is not resolved, an error is produced.

Secrets stores defined in `admin.json` can be accessed only by heap objects in `admin.json`.

## Algorithms for Elliptic Curve Digital Signatures

When the Elliptic Curve Digital Signature Algorithm (ECDSA) is used for signing, and both of the following conditions are met, JWTs are signed with a deterministic ECDSA:

- Bouncy Castle is installed.
- The system property `org.forgerock.secrets.preferDeterministicEcdsa` is `true`, which is its default value.

Otherwise, when ECDSA is used for signing, JWTs are signed with a non-deterministic ECDSA.

A non-deterministic ECDSA signature can be verified by the equivalent deterministic algorithm.

For information about deterministic ECDSA, see *RFC 6979*. For information about Bouncy Castle, see *The Legion of the Bouncy Castle*.

## Chapter 2

# Installation in Detail

For information about how to quickly install and configure IG, see *Getting Started Guide*. The following sections describe other aspects of installation:

- "About Securing Connections"
- "Installing IG in Standalone Mode"
- "Installing IG in Apache Tomcat"
- "Installing IG in Jetty"
- "Installing IG in JBoss EAP"
- "Preparing the Network"
- "Changing the Default Location of the Configuration Folders"
- "Preparing For Load Balancing and Failover"
- "Configuring IG For HTTPS (Client-Side)"
- "Using JWT Sessions"
- "Setting Up AM"

## 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 running in web container mode, and acting as a server, the TLS connection is configured in the container. When IG is running in standalone mode, and acting as a server, the TLS connection is configured in [admin.json](#).

When IG is acting as a client, the TLS connection is configured in the `ReverseProxyHandler`. For details, see "Configuring IG For HTTPS (Client-Side)" and "ReverseProxyHandler" 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, that are suitable for trying out the software, or for deployments where you manage all clients that access IG. For information about how to use well-known CA-signed certificates, see the documentation for the Java Virtual Machine (JVM).

After certificates are properly installed to allow client-server trust, consider the cipher suites configured for use. The cipher suite 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 deployment to use only your preferred cipher suites. IG inherits 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*.

## Installing IG in Standalone Mode

For information about how to install IG in standalone mode (installed from a .zip file, and run outside of a web container), see "Downloading and Starting IG in Standalone Mode" in the *Getting Started Guide*. The following sections describe other installation options for IG in standalone mode:

- "Configuring IG For HTTPS (Server-Side)"
- "Adding .jar Files for IG Extensions"

## Configuring IG For HTTPS (Server-Side)

This section describes how to set up IG to run as a server over HTTPS. IG uses a KeyManager and a private key to prove its identity to the client.

For information about the set up for HTTPS (client-side), see "Configuring IG For HTTPS (Client-Side)".

### Configure IG For HTTPS (Server-Side)

Before you start, install IG in standalone mode, as described in "Downloading and Starting IG in Standalone Mode" in the *Getting Started Guide*.

1. Locate the keystore directory, `ig_keystore_directory`, and in a terminal create an environment variable for it:

```
$ export ig_keystore_directory=/path/to/secrets
```

2. Create a keystore holding a self-signed certificate:

```
$ keytool \  
-genkey \  
-alias https-connector-key \  
-keyalg RSA \  
-keystore $ig_keystore_directory/IG-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. In the secrets directory, add a file called `keystore.pass`, containing the keystore password `password`:

```
$ cd $ig_keystore_directory  
$ echo -n password > keystore.pass
```

4. Add the following line to `$HOME/.openig/bin/env.sh`, replacing the path with your path to the keystore directory:

```
$ export IG_KEYSTORE_DIRECTORY='/path/to/secrets'
```

5. Add the following file to IG:

#### Linux

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

#### Windows

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

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

Notice the following features of the file:

- IG starts on port **8080**, and on **8443** over TLS.
- IG's private keys for TLS are managed by the `SecretsKeyManager`, which references the `KeyStoreSecretStore` that holds the keys.

- The password of the KeyStoreSecretStore is provided by the FileSystemSecretStore.
- The KeyStoreSecretStore maps the keystore alias to the secret ID for retrieving the private signing keys.
- The path to the keystore is provided by an environment variable.

#### 6. Start IG:

```
$ /path/to/identity-gateway/bin/start.sh
...
... started in 1234ms on ports : [8080 8443]
```

#### 7. Access the IG welcome page on <https://openig.example.com:8443>.

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.

## Adding .jar Files for IG Extensions

IG includes a complete Java application programming interface for extending your deployment with customizations. For more information, see "Extending IG Through the Java API"

Create the directory `$HOME/.openig/extra`, and add .jar files for IG extensions to the directory.

When IG starts up, .jar files in `$HOME/.openig/extra` are loaded by the JVM.

## Installing IG in Apache Tomcat

For basic information about how to install IG in Tomcat, see "Downloading and Starting IG in Tomcat" in the *Getting Started Guide*. The following sections describe other installation options:

- "About Using Tomcat"
- "Configuring Cookie Domains in Tomcat"
- "Configuring IG for HTTPS (Server-Side) in Tomcat"
- "Configuring Access to MySQL Over JNDI in Tomcat"
- "Session Stickiness and Session Replication for Tomcat"

## About Using Tomcat

### 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.

## Configuring Cookie Domains in Tomcat

To protect multiple applications running on different hosts, set a cookie domain as follows:

- For stateful sessions, 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" />
```

If `JwtSession` is not configured, stateful sessions are created automatically. For more information, see "Sessions".

- For stateless sessions, configure the `domain` property of `JwtSession`. When set, the JWT cookie can be accessed from different hosts in that domain. When not set, the JWT cookie can be accessed only from the host where the cookie was created. For information, see "JwtSession" in the *Configuration Reference*.

## Configuring IG for HTTPS (Server-Side) in Tomcat

This section describes how to set up IG to run as a server over HTTPS. For information about the set up for HTTPS (client-side), see "Configuring IG For HTTPS (Client-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">
  <SSLHostConfig sslProtocol="TLS" protocols="all" certificateVerification="none">
    <Certificate certificateKeystoreFile="/path/to/tomcat/conf/keystore"
      certificateKeystorePassword="password"
      certificateKeystoreType="PKCS12" />
  </SSLHostConfig>
</Connector>
```

Also create a keystore holding a self-signed certificate:

```
$ keytool \  
-genkey \  
-alias tomcat \  
-keyalg RSA \  
-keystore /path/to/tomcat/conf/keystore \  
-storetype PKCS12 \  
-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.

## Configuring Access to MySQL Over JNDI in Tomcat

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.

## Session Stickiness and Session Replication for Tomcat

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.

## Installing IG in Jetty

For basic information about how to install IG in Jetty, see "Downloading and Starting IG in Jetty" in the *Getting Started Guide*. The following sections describe other installation options:

- "About Using Jetty"
- "Configuring Cookie Domains in Jetty"
- "Configuring IG for HTTPS (Server-Side) in Jetty"
- "Configuring Access MySQL Over JNDI in Jetty"
- "Session Stickiness and Session Replication for Jetty"

### About Using Jetty

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

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.

**Note**

IG depends on `javax.websocket-api` version 1.1, which is a higher version than that provided by Jetty. To prevent errors related to WebSocket, do not include the websocket configuration modules when you configure Jetty.

To change the default port for Jetty in HTTP, edit `http.ini`.

To change the default port for Jetty in HTTPS, edit `server.ini`.

## Configuring Cookie Domains in Jetty

To use IG for multiple protected applications running on different hosts, set a cookie domain as follows:

- For stateful sessions, 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.

If `JwtSession` is not configured, stateful sessions are created automatically. For more information, see "Sessions".

- For stateless sessions, configure the `domain` property of `JwtSession`. When set, the JWT cookie can be accessed from different hosts in that domain. When not set, the JWT cookie can be accessed only from the host where the cookie was created. For information, see "JwtSession" in the *Configuration Reference*.

## Configuring IG for HTTPS (Server-Side) in Jetty

This section describes how to set up Jetty to run IG over HTTPS. For information about the set up for HTTPS (client-side), see "Configuring IG For HTTPS (Client-Side)".

These instructions are for Jetty 9.4.21, 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>.

### Configure Jetty for HTTPS

1. Install Jetty, and set up the location for the Jetty distribution binaries:
  - a. Download a supported version of Jetty server from its download page, and install it to `/path/to/jetty`.



- b. Set the environment variable JETTY\_HOME for `/path/to/jetty`:

```
$ export JETTY_HOME=/path/to/jetty
```

2. Set up the location for configurations and customizations to the Jetty distribution:

- a. Create a directory `/path/to/jetty_base`.
  - b. Set the environment variable JETTY\_BASE for `/path/to/jetty_base`:

```
$ export JETTY_BASE=/path/to/jetty_base
```

3. Set up the keystore:

- a. Remove the built-in keystore:

```
$ rm ${JETTY_HOME}/modules/ssl/keystore
```

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

```
$ keytool \  
-genkey \  
-alias jetty \  
-keyalg RSA \  
-keystore ${JETTY_HOME}/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.

4. Create a directory to store local server customization and configurations in `${JETTY_BASE}`:

- a. Delete the global `start.ini`:

```
$ rm ${JETTY_HOME}/start.ini
```

- b. From `${JETTY_BASE}`, create the `start.d` folder to hold the module `.ini` files:

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

5. From `${JETTY_BASE}`, add the following Jetty configuration modules:

```
$ cd ${JETTY_BASE}  
$ java -jar ${JETTY_HOME}/start.jar \  
--add-to-start=server,webapp,deploy,ssl,jstl,ext,jsp,resources,console-capture,http,https  
INFO : webapp initialized in ${jetty.base}/start.d/webapp.ini
```

```

INFO : ext                initialized in ${jetty.base}/start.d/ext.ini
INFO : server             initialized in ${jetty.base}/start.d/server.ini
INFO : mail               transitively enabled
INFO : servlet            transitively enabled
INFO : jsp                initialized in ${jetty.base}/start.d/jsp.ini
INFO : annotations        transitively enabled
INFO : resources           initialized in ${jetty.base}/start.d/resources.ini
INFO : transactions        transitively enabled
INFO : threadpool         transitively enabled, ini template available with --add-to-start=threadpool
INFO : ssl                initialized in ${jetty.base}/start.d/ssl.ini
INFO : plus               transitively enabled
INFO : deploy             initialized in ${jetty.base}/start.d/deploy.ini
INFO : jstl                initialized in ${jetty.base}/start.d/jstl.ini
INFO : security           transitively enabled
INFO : apache-jsp         transitively enabled
INFO : jndi                transitively enabled
INFO : console-capture    initialized in ${jetty.base}/start.d/console-capture.ini
INFO : apache-jstl        transitively enabled
INFO : http               initialized in ${jetty.base}/start.d/http.ini
INFO : client             transitively enabled
INFO : https              initialized in ${jetty.base}/start.d/https.ini
INFO : bytebufferpool     transitively enabled, ini template available with --add-to-start=bytebufferpool
MKDIR : ${jetty.base}/lib
MKDIR : ${jetty.base}/lib/ext
MKDIR : ${jetty.base}/resources
MKDIR : ${jetty.base}/etc
COPY  : ${jetty.home}/modules/ssl/keystore to ${jetty.base}/etc/keystore
MKDIR : ${jetty.base}/webapps
MKDIR : ${jetty.base}/logs
INFO  : Base directory was modified
  
```

#### Note

IG depends on `javax.websocket-api` version 1.1, which is a higher version than that provided by Jetty. To prevent errors related to WebSocket, do not include the websocket configuration modules when you configure Jetty.

To change the default port for Jetty in HTTP, edit `http.ini`.

To change the default port for Jetty in HTTPS, edit `server.ini`.

- Replace `jetty-util-*.jar` with the version for your installation, and find the obfuscated form of the keystore password:

```

$ cd ${JETTY_HOME}/Lib
$ ls jetty-util-*.jar
  
```

```

$ java -cp jetty-util-*.jar org.eclipse.jetty.util.security.Password password
  
```

```

password
OBF:1v2jluum1xtvlzlej1zler1xtnl1uvk1v1v
MD5:5f4dcc3b5aa765d61d8327deb882cf99
  
```

7. In `${JETTY_BASE}/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:1v2jluumlxtv1zejlzer1xtnluvk1v1v

## KeyManager password
jetty.sslContext.keyManagerPassword=OBF:1v2jluumlxtv1zejlzer1xtnluvk1v1v
```

8. Copy the IG .war file to `${JETTY_BASE}/webapps/IG-7.0.2.war`.
9. Go to `${JETTY_BASE}`, and start Jetty:

```
$ cd ${JETTY_BASE}
$ java -jar ${JETTY_HOME}/start.jar
```

10. Access the IG welcome page on <https://openig.example.com:8443>.

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.

## Configuring Access MySQL Over JNDI in Jetty

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.

## Session Stickiness and Session Replication for Jetty

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.

## Installing IG in JBoss EAP

For basic information about how to install IG in JBoss, see "[Downloading and Starting IG in JBoss](#)" in the *Getting Started Guide*. The following sections describe other installation options:

- "[Configuring Cookie Domains in JBoss EAP](#)"
- "[Configuring IG for HTTPS \(Server-Side\) in JBoss EAP](#)"

### Configuring Cookie Domains in JBoss EAP

To use IG to protect multiple applications running on different hosts, set a cookie domain as follows:

- For stateful sessions, set a cookie domain in JBoss. For information, see the Redhat documentation about *Cookie Domain*.

If `JwtSession` is not configured, stateful sessions are created automatically. For more information, see "[Sessions](#)".

- For stateless sessions, configure the `domain` property of `JwtSession`. When set, the JWT cookie can be accessed from different hosts in that domain. When not set, the JWT cookie can be accessed only from the host where the cookie was created. For information, see "JwtSession" in the *Configuration Reference*.

## Configuring IG for HTTPS (Server-Side) in JBoss EAP

This section describes how to set up JBoss to run IG over HTTPS. These instructions are for JBoss 7.2, and are not compatible with earlier versions. For information about the set up for HTTPS (client-side), see "Configuring IG For HTTPS (Client-Side)".

The default ephemeral DH key size in the JVM is 1024-bit. To support stronger ephemeral DH keys, and protect against weak keys, set the following system property: `jdk.tls.ephemeralDHKeySize=2048`.

### Configure Jetty for HTTPS

Before you start, install IG in JBoss as described in "Downloading and Starting IG in JBoss" in the *Getting Started Guide*. JBoss is installed in `/path/to/jboss`.

1. Set the environment variable `JBOSS_HOME` in two terminals:

```
$ export JBOSS_HOME=/path/to/jboss
```

2. In the first terminal, create a user with administrative permissions to run the setup:

```
$ ${JBOSS_HOME}/bin/add-user.sh myadmin myadmin-password
Added user 'myadmin' to file '${JBOSS_HOME}/standalone/configuration/mgmt-users.properties'
Added user 'myadmin' to file '${JBOSS_HOME}/domain/configuration/mgmt-users.properties'
```

3. Make a temporary directory for the settings and keystore:

```
$ mkdir $JBOSS_HOME/tmp
```

4. Create the following file as `/${JBOSS_HOME}/tmp/batch_settings`:

```
/socket-binding-group=standard-sockets/socket-binding=http/:write-attribute(name=port, value=8080)
/socket-binding-group=standard-sockets/socket-binding=https/:write-attribute(name=port, value=8443)
/socket-binding-group=standard-sockets/socket-binding=ajp/:write-attribute(name=port, value=8009)
/socket-binding-group=standard-sockets/socket-binding=management-http/:write-attribute(name=port,
value=9990)
/socket-binding-group=standard-sockets/socket-binding=management-https/:write-attribute(name=port,
value=9993)
/subsystem=deployment-scanner/scanner=default/:write-attribute(name="scan-interval", value="2000")
/interface=management/:write-attribute(name="inet-address",
value="${jboss.bind.address:openig.example.com}")
/interface=public/:write-attribute(name="inet-address",
value="${jboss.bind.address:openig.example.com}")
```

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

```
$ keytool \  
-genkey \  
-alias jboss \  
-storetype PKCS12 \  
-keyalg RSA \  
-keystore ${JBOSS_HOME}/tmp/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.

6. Start JBoss as a standalone server:

```
$ ${JBOSS_HOME}/bin/standalone.sh
```

7. While JBoss is running, in the second terminal, update the batch settings:

```
$ ${JBOSS_HOME}/bin/jboss-cli.sh --connect \  
--controller=openig.example.com:9990 command="run-batch -v \  
--file=${JBOSS_HOME}/tmp/batch_settings"
```

8. Make sure IG is deployed on port 8080:

```
$ ${JBOSS_HOME}/bin/jboss-cli.sh --connect \  
--controller=openig.example.com:9990 command="deployment list"
```

9. Enable SSL:

- a. Enable the SSL server:

```
$ ${JBOSS_HOME}/bin/jboss-cli.sh --connect \  
--controller=openig.example.com:9990 command="security enable-ssl-http-server \  
--key-store-path=${JBOSS_HOME}/tmp/keystore \  
--key-store-password=password \  
--key-store-type=PKCS12"
```

```
Server reloaded.  
SSL enabled for default-server  
ssl-context is ssl-context-keystore  
key-manager is key-manager-keystore  
key-store is keystore
```

- b. Access the IG welcome page on <https://openig.example.com:8443>.

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.

## 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.

## Changing the Default Location of the Configuration Folders

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

*Linux*

```
$HOME/.openig
```

*Windows*

```
%appdata%\OpenIG
```

Change the default base location in the following ways:

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

*Linux*

```
$ export IG_INSTANCE_DIR=/path/to/instance-dir
```

*Windows*

```
C:> set IG_INSTANCE_DIR=c:\path\to\instance-dir
```

- For IG running in web container mode, set the `ig.instance.dir` Java system property to the full path of the base location. The following example starts Jetty in the foreground and sets the value of `ig.instance.dir`:

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

- For IG running in standalone mode, specify the base location as an argument. The following example reads the configuration from the `config` directory under `/path/to/instance-dir`:

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

## 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 saves state information in the following ways:

- By using a handler, such as a `SamlFederationHandler` or a custom `ScriptableHandler`, that can store information in the context. Most handlers depend on information in the context, some of which is first stored by IG.
- By using filters, such as `AssignmentFilters`, `HeaderFilters`, `OAuth2ClientFilters`, `OAuth2ResourceServerFilters`, `ScriptableFilters`, `SqlAttributesFilters`, and `StaticRequestFilters`, that can store information in the context. Most filters depend on information in the request, response, or context, some of which is first stored by IG.

IG retrieves information locally in the following ways:

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

By default, the context data, including storage of the default session implementation, resides in memory. For information about whether to store session data on the user-agent instead, see "`JwtSession`" in the *Configuration Reference*.

When using `JwtSession` with a cookie domain, share the encryption keys and the signature symmetric secret across all IG configurations so that any server can read or update JWT cookies from any other server in the same cookie domain.

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.

For more information, see "`Session Stickiness and Session Replication for Tomcat`" and "`Session Stickiness and Session Replication for Jetty`".



## Configuring IG For HTTPS (Client-Side)

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

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

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

- `KeyStore`, to hold the server certificates or the CA's signing certificate. See "`KeyStore`" in the *Configuration Reference*.
- `SecretsTrustManager`, to let IG handle the certificates in the `KeyStore` when deciding whether to trust a server certificate. See "`SecretsTrustManager`" in the *Configuration Reference*.
- (Optional) `KeyManager`, to let IG present its certificate from the keystore when the server must authenticate IG as client. See "`KeyManager`" in the *Configuration Reference*.
- `ClientHandler` and `ReverseProxyHandler` reference to `ClientTlsOptions`, for connecting to TLS-protected endpoints. See "`ClientTlsOptions`" in the *Configuration Reference*.

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

### Set Up IG for HTTPS (Client-Side) for Untrusted Servers

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

```
$ cd /path/to/secrets
$ jar --verbose --extract \
--file $sampleapp_install_dir/IG-sample-application-7.0.2.jar tls/sampleapp-cert.pem
inflated: tls/sampleapp-cert.pem
```

The file `/path/to/secrets/tls/sampleapp-cert.pem` is created.

3. From the same directory, import the certificate into the IG keystore, and answer `yes` to trust the certificate:

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

...
Trust this certificate? [no]:yes
Certificate was added to keystore
```

**Note**

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

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

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

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

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

```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcnQ='
```

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

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

*Linux*

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

*Windows*

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

7. Add the following route to IG:

### Linux

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

### Windows

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

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

```
    "capture": "all"
  }
},
"handler": "ReverseProxyHandler-1"
}
```

Notice the following features of the route:

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

## 8. Test the setup:

### a. Start the sample application

```
$ java -jar $sampleapp_install_dir/IG-sample-application-7.0.2.jar
```

### b. Go to `http://openig.example.com:8080/home/client-side-https`.

The request is proxied transparently to the sample application, on the TLS port `8444`. Check the route log for `GET https://app.example.com:8444/home/client-side-https`.

## Using JWT Sessions

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

Authenticated encryption encrypts data and then signs it with HMAC, in a single step. For more information, see [Authenticated encryption](#). For information about `JwtSession`, see "JwtSession" in the [Configuration Reference](#).

- "Encrypting JWT Sessions"
- "Sharing JWT Session Between Multiple Instances of IG"

## Encrypting JWT Sessions

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

### Set Up JWT Encryption Keys

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

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

#### Note

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

2. Add the following route to IG:

#### Linux

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

#### Windows

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

```
{  
  "name": "jwt-session-encrypt",  
  "heap": [{  
    "name": "KeyStoreSecretStore-1",  
    "type": "KeyStoreSecretStore",  
    "config": {  
      "file": "/path/to/secrets/jwtsessionkeystore.pkcs12",  
      "storeType": "PKCS12",  
      "storePassword": "keystore.secret.id",  
      "secretsProvider": ["SystemAndEnvSecretStore-1"],  
      "mappings": [{  
        "secretId": "jwtsession.symmetric.secret.id",  
        "aliases": ["symmetric-key"]  
      }]  
    }  
  }  
],  
  {  
    "name": "SystemAndEnvSecretStore-1",  
    "type": "SystemAndEnvSecretStore"  
  }  
}
```

```
    ],
    "session": {
      "type": "JwtSession",
      "config": {
        "authenticatedEncryptionSecretId": "jwtsession.symmetric.secret.id",
        "encryptionMethod": "A256CBC-HS512",
        "secretsProvider": ["KeyStoreSecretStore-1"],
        "cookie": {
          "name": "IG",
          "domain": ".example.com"
        }
      }
    },
    "handler": {
      "type": "StaticResponseHandler",
      "config": {
        "status": 200,
        "reason": "OK",
        "headers": {
          "Content-Type": [ "text/plain" ]
        },
        "entity": "Hello world!"
      }
    },
    "condition": "${request.uri.path == '/jwt-session-encrypt'}"
  }
}
```

Notice the following features of the route:

- The route matches requests to `/jwt-session-encrypt`.
  - The `KeyStoreSecretStore` uses the `SystemAndEnvSecretStore` in the heap to manage the store password.
  - The `JwtSession` uses the `KeyStoreSecretStore` in the heap to manage the session encryption secret.
3. In the terminal where you will run the IG instance, create an environment variable for the value of the keystore password:

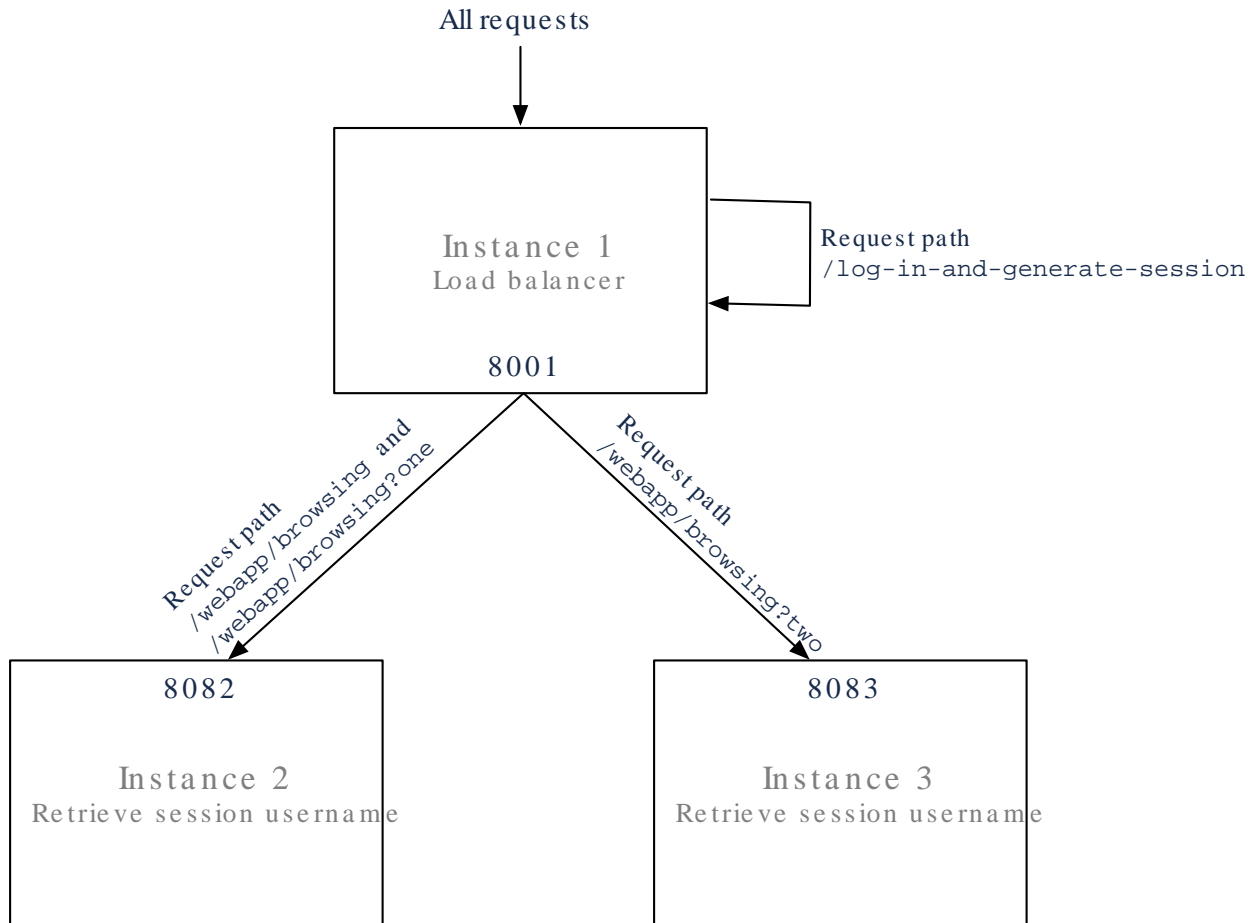
```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcnQ='
```

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

## Sharing JWT Session Between Multiple Instances of IG

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

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



### Set Up Shared Secrets for Multiple Instances of IG

In this example, IG is running in web container mode.

1. Generate a keystore to contain the encryption key, where the keystore and the key have the password **password**:

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

**Note**

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

2. Set up and start the first instance of IG, which acts as the load balancer:

- a. Download and install the instance to `/path/to/instance1`.
- b. Create a configuration directory for the instance:

```
$ mkdir $HOME/.instance1/
```

c. Add the following route to IG:

*Linux*

```
$HOME/.openig/config/routes/instance1-loadbalancer.json
```

*Windows*

```
%appdata%\OpenIG\config\routes\instance1-loadbalancer.json
```

```
{
  "name": "instance1-loadbalancer",
  "heap": [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file": "/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePassword": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{
        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
      }]
    }
  }],
  {
    "name": "SystemAndEnvSecretStore-1",
    "type": "SystemAndEnvSecretStore"
  }
],
  "session": {
    "type": "JwtSession",
    "config": {
      "authenticatedEncryptionSecretId": "jwtsession.symmetric.secret.id",
      "encryptionMethod": "A256CBC-HS512",
      "secretsProvider": ["KeyStoreSecretStore-1"],
      "cookie": {
        "name": "IG",
        "domain": ".example.com"
      }
    }
  }
}
```



```

    },
    "handler": {
      "type": "DispatchHandler",
      "config": {
        "bindings": [{
          "condition": "${matches(request.uri.path, '/webapp/browsing') and
(contains(request.uri.query, 'one') or empty(request.uri.query))}",
          "baseURI": "http://openig.example.com:8082",
          "handler": "ReverseProxyHandler"
        }, {
          "condition": "${matches(request.uri.path, '/webapp/browsing') and
contains(request.uri.query, 'two')}",
          "baseURI": "http://openig.example.com:8083",
          "handler": "ReverseProxyHandler"
        }, {
          "condition": "${matches(request.uri.path, '/log-in-and-generate-session')}",
          "handler": {
            "type": "Chain",
            "config": {
              "filters": [{
                "type": "AssignmentFilter",
                "config": {
                  "onRequest": [{
                    "target": "${session.authUsername}",
                    "value": "Sam Carter"
                  }]
                }
              ]
            }
          },
          "handler": {
            "type": "StaticResponseHandler",
            "config": {
              "status": 200,
              "headers": {
                "Content-Type": [ "text/html" ]
              },
              "entity": "<html><body>Sam Carter logged IN. (JWT session generated)</body></
html>"
            }
          }
        }
      ]
    }
  },
  "capture": "all"
}

```

Notice the following features of the route:

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

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

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

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

- e. Start the instance on port 8001:

```
$ java -jar start.jar -Djetty.http.port=8001 -Dig.instance.dir=$HOME/.instance1/
```

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

- a. Download and install the instance to `/path/to/instance2`

- b. Create a configuration directory for the instance:

```
$ mkdir $HOME/.instance2/
```

- c. Add the following route as `$HOME/.instance2/config/routes/instance2-retrieve-session-username.json`:

```
{
  "name": "instance2-retrieve-session-username",
  "heap": [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file": "/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePassword": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{
        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
      }]
    }
  ]
},
{
  "name": "SystemAndEnvSecretStore-1",
  "type": "SystemAndEnvSecretStore"
}
],
"session": {
  "type": "JwtSession",
  "config": {
    "authenticatedEncryptionSecretId": "jwtsession.symmetric.secret.id",
    "encryptionMethod": "A256CBC-HS512",
    "secretsProvider": ["KeyStoreSecretStore-1"],
    "cookie": {
      "name": "IG",
      "domain": ".example.com"
    }
  }
}
},
"handler": {
```

```

    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/html" ]
      },
      "entity": "<html><body>${session.authUsername!= null?'Hello, '
        .concat(session.authUsername).concat(' !'):'Session.authUsername is not defined'}! (instance2)</
        body></html>"
    },
    "condition": "${matches(request.uri.path, '/webapp/browsing')}",
    "capture": "all"
  }
}

```

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

- The route matches the condition `/webapp/browsing`. When a request matches `/webapp/browsing`, the `DispatchHandler` dispatches it to instance 2.
  - The `StaticResponseHandler` displays information from the session context.
- d. In the terminal where you will run the IG instance, create an environment variable for the value of the keystore password:

```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcnQ='
```

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

- e. Start the instance on port `8082`:

```
$ java -jar start.jar -Djetty.http.port=8082 -Dig.instance.dir=$HOME/.instance2/
```

#### 4. Set up and start the third instance of IG:

- a. Download and install the instance to `/path/to/instance3`
- b. Create the configuration directory:

```
$ mkdir $HOME/.instance3/
```

- c. Add the following route as `$HOME/.instance3/config/routes/instance3-retrieve-session-username.json`:

```

{
  "name": "instance3-retrieve-session-username",
  "heap": [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file": "/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePassword": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{

```

```

        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
    }
  },
  {
    "name": "SystemAndEnvSecretStore-1",
    "type": "SystemAndEnvSecretStore"
  }
],
"session": {
  "type": "JwtSession",
  "config": {
    "authenticatedEncryptionSecretId": "jwtsession.symmetric.secret.id",
    "encryptionMethod": "A256CBC-HS512",
    "secretsProvider": ["KeyStoreSecretStore-1"],
    "cookie": {
      "name": "IG",
      "domain": ".example.com"
    }
  }
},
"handler": {
  "type": "StaticResponseHandler",
  "config": {
    "status": 200,
    "headers": {
      "Content-Type": [ "text/html" ]
    },
    "entity": "<html><body>${session.authUsername!= null?'Hello,
'.concat(session.authUsername).concat(' !'):'Session.authUsername is not defined'}! (instance3)</
body></html>"
  }
},
"condition": "${matches(request.uri.path, '/webapp/browsing')}",
"capture": "all"
}

```

Notice that the route is the same as `instance2.json`, apart from the text in the entity of the `StaticResponseHandler`.

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

```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcnQ='
```

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

- e. Start the instance on port `8083`:

```
$ java -jar start.jar -Djetty.http.port=8083 -Dig.instance.dir=$HOME/.instance3/
```

5. Test the setup:

- a. Access instance 1, to generate a session:

```
$ curl -v http://openig.example.com:8001/log-in-and-generate-session
GET /log-in-and-generate-session HTTP/1.1
...
HTTP/1.1 200 OK
Content-Length: 84
Set-Cookie: IG=eyJ...HyI; Path=/; Domain=.example.com; HttpOnly
...
Sam Carter logged IN. (JWT session generated)
```

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

```
$ curl -v http://openig.example.com:8001/webapp/browsing?one --header "cookie:IG=<JWT cookie>"
GET /webapp/browsing?one HTTP/1.1
...
cookie: IG=eyJ...QHyI
...
HTTP/1.1 200 OK
...
Hello, Sam Carter !! (instance2)
```

Note that instance 2 can access the session info.

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

```
$ curl -v http://openig.example.com:8001/webapp/browsing?two --header "cookie:IG=<JWT cookie>"
GET /webapp/browsing?two HTTP/1.1
...
cookie: IG=eyJ...QHyI
...
HTTP/1.1 200 OK
...
Hello, Sam Carter !! (instance3)
```


Note that instance 3 can access the session info.

## Setting Up AM

This section contains procedures for setting up items in AM that you can use in many of the tutorials in this guide. For more information about setting up AM, see the *Access Management Docs*.

### Set Up a Sample User in AM

Follow these steps to add an example user to the AM configuration:

1. In the AM console, select the top-level realm, and then select  Identities.
2. Click Add Identity and add a user with the following values:
  - ID/username: **george**
  - First name: **george**

- Last name: `costanza`
- Password: `C0stanza`
- Email Address: `george@example.com`
- Employee number: `123`

### *Set Up an IG Agent in AM*

In AM 7, follow these steps to set up an agent that acts on behalf of IG in the same domain. In AM 6.5 or earlier, follow the steps in "Set Up an IG Agent in AM 6.5 and Earlier". After the agent is authenticated, the token can be used to get the user profile, evaluate policies, and to connect to the AM notification endpoint:

1. In the AM console, select the top-level realm, and then select Applications > Agents > Identity Gateway.
2. Add an agent with the following values:
  - Agent ID: `ig_agent`
  - Password: `password`

### *Set Up an IG Agent in AM 6.5 and Earlier*

In AM 6.5 and earlier versions, follow these steps to set up an agent that acts on behalf of IG. After the agent is authenticated, the token can be used to get the user profile, evaluate policies, and to connect to the AM notification endpoint:

1. In the AM console, select the top-level realm, and then select Applications > Agents > Java (or J2EE).
2. Add an agent with the following values:
  - Agent ID: `ig_agent` for SSO, `ig_agent_cdssso` for CDSSO
  - Agent URL: `http://openig.example.com:8080/agentapp` for SSO, `http://openig.ext.com:8080/agentapp` for CDSSO
  - Server URL: `http://openam.example.com:8088/openam`
  - Password: `password`
3. On the Global tab, deselect Agent Configuration Change Notification.

This option stops IG from being notified about agent configuration changes in AM, because they are not required by IG.

4. (For SSO in different domains) On the SSO tab, select the following values:

- Cross Domain SSO: Deselect this option
- CDSSO Redirect URI: `/home/cdsso/redirect`

(For enforcing AM policy decisions in different domains) On the SSO tab, select the following values:

- Cross Domain SSO: Deselect this option
- CDSSO Redirect URI: `/home/pep-cdsso/redirect`

### *Find the Name of Your AM Session Cookie*

The procedures in this guide assume you are using the default AM session cookie, `iPlanetDirectoryPro`. If not, find your session cookie name, and substitute its value in the procedures.

- In a terminal, access the AM `serverinfo` endpoint to find the session cookie name:

```
$ curl http://openam.example.com:8088/openam/json/serverinfo/*  
  
...  
"cookieName": "iPlanetDirectoryPro"
```

## Chapter 3

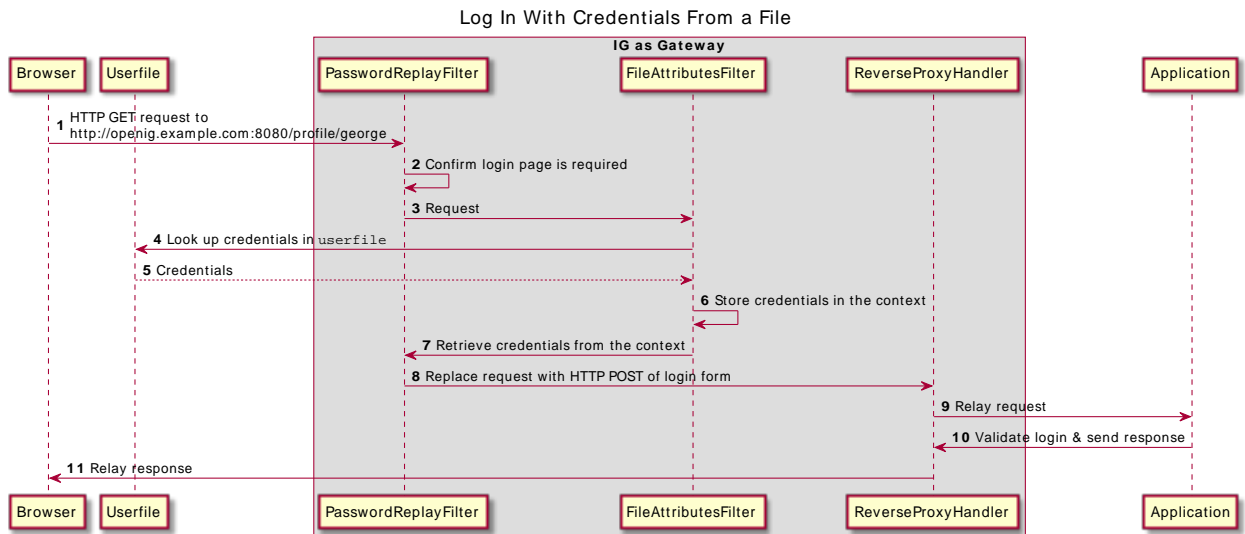
# Getting Login Credentials From Data Sources

The following sections describe how to look up credentials to log in to the sample app:

- "Logging In With Credentials From a File"
- "Logging In With Credentials From a Database"

## Logging In With Credentials From a File

The following figure illustrates the flow of requests when IG uses credentials in a file to log a user in to the sample app:



- IG intercepts the browser's HTTP GET request, which matches the route condition.
- The PasswordReplayFilter confirms that a login page is required, and
- The FileAttributesFilter uses the email address to look up the user credentials in a file, and stores the credentials in the request context attributes map.



- The PasswordReplayFilter retrieves the credentials from the attributes map, builds the login form, and performs the HTTP POST request to the sample app.
- The sample application validates the credentials, and responds with a profile page.
- The ReverseProxyHandler passes the response to the browser.

### Log in to the Sample App With Credentials From a File

Before you start, prepare IG and the sample application as described in [Getting Started Guide](#).

1. On your system, add the following data in a comma-separated value file called `/tmp/userfile` (on Windows `C:\Temp\userfile`):

```
username,password,fullname,email
george,C0stanza,George Costanza,george@example.com
kramer,N3wman12,Kramer,kramer@example.com
bjensen,H1falutin,Babs Jensen,bjensen@example.com
demo,Ch4ng31t,Demo User,demo@example.com
kvaughan,B5ibery12,Kirsten Vaughan,kvaughan@example.com
scarter,S9rain12,Sam Carter,scarter@example.com
```

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

Linux

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

Windows

```
%appdata%\OpenIG\config\routes\static-resources.json
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

3. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/02-file.json
```

Windows

```
%appdata%\OpenIG\config\routes\02-file.json
{
  "name": "02-file",
  "condition": "${matches(request.uri.path, '^/profile')}",
  "capture": "all",
  "handler": {
    "type": "Chain",
    "baseURI": "http://app.example.com:8081",
    "config": {
      "filters": [
```

```
{
  "type": "PasswordReplayFilter",
  "config": {
    "loginPage": "${matches(request.uri.path, '^/profile/george') and (request.method ==
'GET')}}",
    "credentials": {
      "type": "FileAttributesFilter",
      "config": {
        "file": "/tmp/userfile",
        "key": "email",
        "value": "george@example.com",
        "target": "${attributes.credentials}"
      }
    },
    "request": {
      "method": "POST",
      "uri": "http://app.example.com:8081/login",
      "form": {
        "username": [
          "${attributes.credentials.username}"
        ],
        "password": [
          "${attributes.credentials.password}"
        ]
      }
    }
  }
},
"handler": "ReverseProxyHandler"
}
```

Notice the following features of the route:

- The route matches requests to `/profile`.
- The `PasswordReplayFilter` specifies a `loginPage` page property:
  - When a request is an HTTP GET, and the request URI path is `/profile/george`, the expression resolves to `true`. The request is directed to a login page.

The `FileAttributesFilter` looks up the key and value in `/tmp/userfile`, and stores them in the context.

The `request` object retrieves the username and password from the context, and replaces the browser's original HTTP GET request with an HTTP POST login request, containing the credentials to authenticate.

- For other requests, the expression resolves to `false`. The request passes to the `ReverseProxyHandler`, which directs it to the profile page of the sample app.

## Test the Setup

1. Go to `http://openig.example.com:8080/profile/george`.

Because the property `loginPage` resolves to `true`, the `PasswordReplayFilter` processes the request to obtain the login credentials. The sample app returns the profile page for George.

2. Go to `http://openig.example.com:8080/profile/bob`, or to any other URI starting with `http://openig.example.com:8080/profile`.

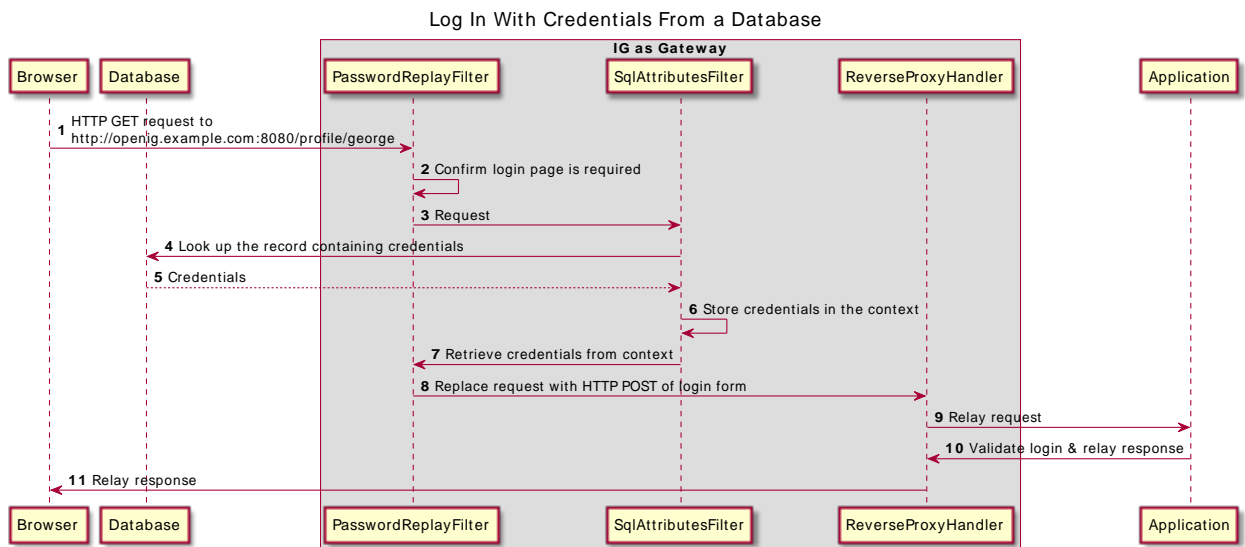
Because the property `loginPage` resolves to `false`, the `PasswordReplayFilter` passes the request directly to the `ReverseProxyHandler`. The sample app returns the login page.

## Logging In With Credentials From a Database

This section describes how to configure IG to get credentials from a database. This example is tested with Jetty and H2 1.4.197.

The following figure illustrates the flow of requests when IG uses credentials from a database to log a user in to the sample app:

*Log in With Credentials From a Database*



- IG intercepts the browser's HTTP GET request.

- The PasswordReplayFilter confirms that a login page is required, and passes the request to the SqlAttributesFilter.
- The SqlAttributesFilter uses the email address to look up credentials in H2, and stores them in the request context attributes map.
- The PasswordReplayFilter retrieves the credentials from the attributes map, builds the login form, and performs the HTTP POST request to the sample app.
- The sample application validates the credentials, and responds with a profile page.

## Set Up the Database

Before you start, prepare IG and the sample application as described in [Getting Started Guide](#).

1. On your system, add the following data in a comma-separated value file called `/tmp/userfile` (on Windows, `C:\Temp\userfile`):

```
username,password,fullname,email
george,C0stanza,George Costanza,george@example.com
kramer,N3wman12,Kramer,kramer@example.com
bjensen,H1falutin,Babs Jensen,bjensen@example.com
demo,Ch4ng31t,Demo User,demo@example.com
kvaughan,B5ibery12,Kirsten Vaughan,kvaughan@example.com
scarter,S9rain12,Sam Carter,scarter@example.com
```

2. Download and unpack the H2 database, and then start H2:

```
$ sh /path/to/h2/bin/h2.sh
```

H2 starts, listening on port 8082, and opens the H2 Console in a browser.

3. In the H2 Console, select the following options, and then select Connect to access the console:

- Saved Settings: `Generic H2 (Server)`

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, add the following text, and then run it to create the user table:

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

5. In the console, add the following text, and then run it to verify that the table contains the same users as the file:

```
SELECT * FROM users;
```

6. Add the `.jar` file `/path/to/h2/bin/h2-*.jar` to the IG configuration:

- For IG in standalone mode, create the directory `$HOME/.openig/extra`, where `$HOME/.openig` is the instance directory: and add `.jar` files to the directory.
- For IG in web container mode, add `.jar` files to the web container classpath. For example, in Jetty use `/path/to/jetty/webapps/ROOT/WEB-INF/lib`.

## Set Up IG

1. Set an environment variable for the database password, and then restart IG:

```
$ export DATABASE_PASSWORD='cGFzc3dvcmQ='
```

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

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

Linux

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

Windows

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

3. In IG, add the following route as `$HOME/.openig/config/routes/03-sql.json` (on Windows, `$HOME/.openig/config/routes/03-sql.json`):

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "JdbcDataSource-1",
      "type": "JdbcDataSource",
      "config": {
        "driverClassName": "org.h2.Driver",
        "jdbcUrl": "jdbc:h2:tcp://localhost/~:/test",
        "username": "sa",
        "passwordSecretId": "database.password",
        "secretsProvider": "SystemAndEnvSecretStore-1"
      }
    }
  ],
  "name": "sql",
  "condition": "${matches(request.uri.path, '^/profile')}",
  "handler": {
    "type": "Chain",

```

```
"baseURI": "http://app.example.com:8081",
"config": {
  "filters": [
    {
      "type": "PasswordReplayFilter",
      "config": {
        "loginPage": "${matches(request.uri.path, '^/profile/george') and (request.method ==
'GET')}",
        "credentials": {
          "type": "SqlAttributesFilter",
          "config": {
            "dataSource": "JdbcDataSource-1",
            "preparedStatement":
"SELECT username, password FROM users WHERE email = ?;",
            "parameters": [
              "george@example.com"
            ],
            "target": "${attributes.sql}"
          }
        },
        "request": {
          "method": "POST",
          "uri": "http://app.example.com:8081/login",
          "form": {
            "username": [
              "${attributes.sql.USERNAME}"
            ],
            "password": [
              "${attributes.sql.PASSWORD}"
            ]
          }
        }
      }
    }
  ],
  "handler": "ReverseProxyHandler"
}
```

Notice the following features of the route:

- The route matches requests to `/profile`.
- The `PasswordReplayFilter` specifies a `loginPage` page property:
  - When a request is an HTTP GET, and the request URI path is `/profile/george`, the expression resolves to true. The request is directed to a login page.

The `SqlAttributesFilter` specifies the data source to access, a prepared statement to look up the user's record, a parameter to pass into the statement, and where to store the search results in the request context attributes map.

The `request` object retrieves the username and password from the context, and replaces the browser's original HTTP GET request with an HTTP POST login request, containing the credentials to authenticate.

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

- For other requests, the expression resolves to `false`. The request passes to the `ReverseProxyHandler`, which directs it to the profile page of the sample app.

### *Test the Setup*

1. Go to `http://openig.example.com:8080/profile`.

Because the property `loginPage` resolves to `false`, the `PasswordReplayFilter` passes the request directly to the `ReverseProxyHandler`. The sample app returns the login page.

2. Go to `http://openig.example.com:8080/profile/george`.

Because the property `loginPage` resolves to `true`, the `PasswordReplayFilter` processes the request to obtain the login credentials. The sample app returns the profile page for George.

## Chapter 4

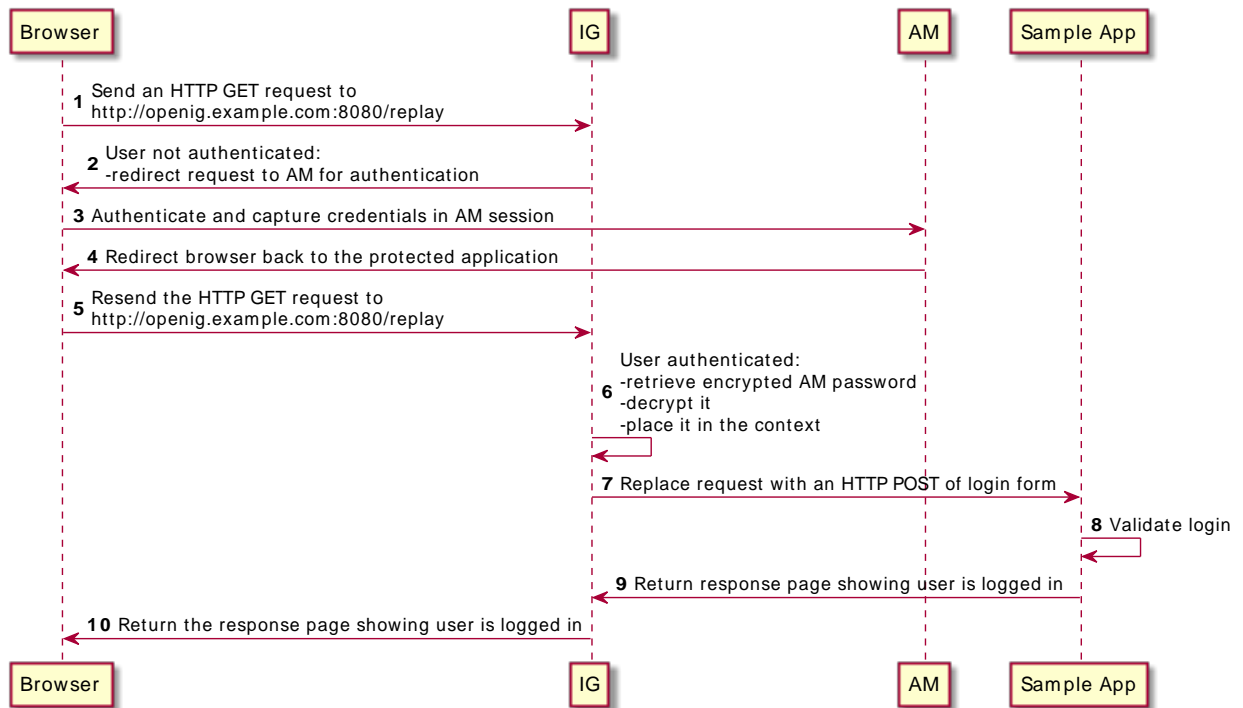
# Getting Login Credentials From AM

Use IG with AM's password capture and replay to bring SSO to legacy web applications, without the need to edit, upgrade, or recode. This feature helps you to integrate legacy web applications with other applications using the same user identity.

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

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.

### Get Login Credentials From AM

Before you start, prepare AM, IG, and the sample application as described in "Example Installation for This Guide".

1. Generate an AES 256-bit key:





```
$ openssl rand -base64 32
loH...UFQ=
```

2. Set up AM:
  - a. (For AM 6.5.x and earlier versions) Select  Identities > demo, and set the demo user password to `Ch4ng31t`.
  - b. (For AM 6.5.3 and later versions) Select  Services > Add a Service, and add a Validation Service with the following Valid goto URL Resources:
    - `http://openig.example.com:8080/*`
    - `http://openig.example.com:8080/*?*`
  - c. Select Applications > Agents > Identity Gateway, add an agent with the following values:

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

Leave all other values as default.

For AM 6.5.x and earlier versions, set up an agent as described in "Set Up an IG Agent in AM 6.5 and Earlier".

- d. Update the Authentication Post Processing Classes for password replay:
  - i. Select  Authentication > Settings > Post Authentication Processing.
  - ii. In Authentication Post Processing Classes, add `com.sun.identity.authentication.spi.JwtReplayPassword`.
- e. Add the AES 256-bit key to AM:
  - i. Select  DEPLOYMENT >  Servers, and then select the AM server name, `http://openam.example.com:8088/openam`.  
In earlier version of AM, select Configuration > Servers and Sites.
  - ii. Select  Advanced, and add the following property:
    - PROPERTY NAME: `com.sun.am.replaypasswd.key`
    - PROPERTY VALUE: The value of the AES 256-bit key from step 1.
- f. Select Configure > Global Services > Platform, and add `example.com` as an AM cookie domain.  
By default, AM sets host-based cookies. After authentication with AM, requests can be redirected to AM instead of to the resource.

### 3. Set up IG:

- a. Set environment variables for the value of the AES 256-bit key in step 1, and the IG agent password, and then restart IG:

```
$ export AES_KEY='AES 256-bit key'  
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

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

*Linux*

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

*Windows*

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to IG:

*Linux*

```
$HOME/.openig/config/routes/04-replay.json
```

*Windows*

```
%appdata%\OpenIG\config\routes\04-replay.json
```

```
{
  "name": "04-replay",
  "condition": "${matches(request.uri.path, '/replay')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://openam.example.com:8088/openam/"
      }
    },
    {
      "name": "CapturedUserPasswordFilter",
      "type": "CapturedUserPasswordFilter",
      "config": {
        "ssoToken": "${contexts.ssoToken.value}",
        "keySecretId": "aes.key",
        "keyType": "AES",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "amService": "AmService-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ]
    }
  }
}
```

```
    },
    {
      "type": "PasswordReplayFilter",
      "config": {
        "loginPage": "${true}",
        "credentials": "CapturedUserPasswordFilter",
        "request": {
          "method": "POST",
          "uri": "http://app.example.com:8081/login",
          "form": {
            "username": [
              "${contexts.ssoToken.info.uid}"
            ],
            "password": [
              "${contexts.capturedPassword.value}"
            ]
          }
        }
      }
    }
  ],
  "handler": "ReverseProxyHandler"
}
}
```

Notice the following features of the route:

- The route matches requests to `/replay`.
- The agent password for AmService is provided by a `SystemAndEnvSecretStore` in the heap.
- If the request does not have a valid AM session cookie, the `SingleSignOnFilter` redirects the request to AM for authentication.

After authentication, the `SingleSignOnFilter` passes the request to the next filter, storing the cookie value in an `SsoTokenContext`.

- The `PasswordReplayFilter` uses the `CapturedUserPasswordFilter` declared in the heap to retrieve the AM password from AM session properties. The `CapturedUserPasswordFilter` uses the AES 256-bit key to decrypt the password, and then makes it available in a `CapturedUserPasswordContext`.

The value of the AES 256-bit key is provided by the `SystemAndEnvSecretStore`.

The `PasswordReplayFilter` retrieves the username and password from the context. It replaces the browser's original HTTP GET request with an HTTP POST login request containing the credentials to authenticate to the sample application.

4. Test the setup:
  - a. If you are logged in to AM, log out.

- b. Go to <http://openig.example.com:8080/replay>. The SingleSignOnFilter redirects the request to AM for authentication.
- c. Log in to AM as user `demo`, password `Ch4ng31t`. The request is redirected to the sample application.

## Chapter 5

# Single Sign-On and Cross-Domain Single Sign-On

The following sections describe how to set up single sign-on for requests in the same domain and in a different domain:

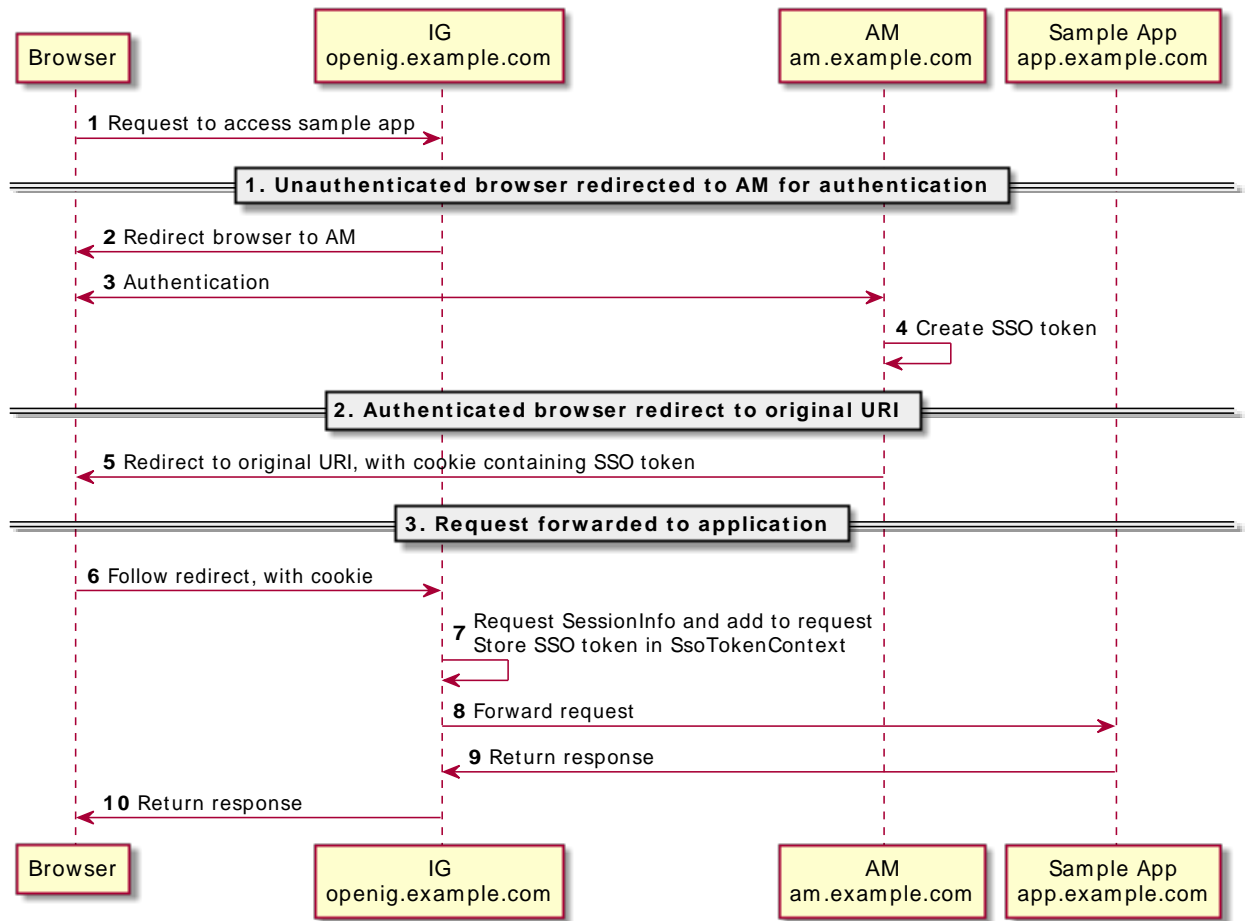
- "Authenticating With SSO"
- "Authenticating With CDSSO"
- "Using WebSocket Notifications to Evict the Session Info Cache"

## Authenticating With SSO

In SSO using the `SingleSignOnFilter`, IG processes a request using authentication provided by AM. IG and the authentication provider must run on the same domain.

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

### 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 AM session info to the request, and stores the SSO token in the context for use by downstream filters and handlers.


- IG forwards the request to the sample app, and the sample app returns the requested resource to the browser.

## Authenticate With SSO

This procedure gives an example of how to set up SSO, where AM on `openam.example.com` authenticates users that are processed by IG on `openig.example.com`.

Before you start, prepare AM, IG, and the sample application as described in "Example Installation for This Guide".

### 1. Set up AM:

- (For AM 6.5.x and earlier versions) Select  Identities > demo, and set the demo user password to `Ch4ng31t`.
- (For AM 6.5.3 and later versions) Select  Services > Add a Service, and add a Validation Service with the following Valid goto URL Resources:
  - `http://openig.example.com:8080/*`
  - `http://openig.example.com:8080/*?*`
- Select Applications > Agents > Identity Gateway, add an agent with the following values:
  - Agent ID: `ig_agent`
  - Password: `password`

Leave all other values as default.

For AM 6.5.x and earlier versions, set up an agent as described in "Set Up an IG Agent in AM 6.5 and Earlier".

- Select Configure > Global Services > Platform, and add `example.com` as an AM cookie domain.

By default, AM sets host-based cookies. After authentication with AM, requests can be redirected to AM instead of to the resource.

### 2. Set up IG:

- Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

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

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

*Linux*



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

Windows

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

- c. Add the following route to IG:

Linux

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

Windows

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

```
{
  "name": "sso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/sso$')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://openam.example.com:8088/openam/",
        "version": "7"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        }
      ]
    }
  }
},
```

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

For information about how to set up the IG route in Studio, see "Policy Enforcement in Structured Editor" in the *Studio User Guide* or "Protecting a Web App With Freeform Designer" in the *Studio User Guide*.

3. Test the setup:
  - a. If you are logged in to AM, log out and clear any cookies.
  - b. Go to `http://openig.example.com:8080/home/sso`.

The `SingleSignOnFilter` redirects the request to AM for authentication.

- c. Log in to AM as user `demo`, password `Ch4ng31t`.

The `SingleSignOnFilter` passes the request to sample app, which returns the profile page.

### *Authenticate With SSO Through an AM Authentication Tree*

This procedure gives an example of how to authenticate by using SSO and the example authentication tree provided in AM, instead of the default authentication service.

1. Set up the example in "Authenticate With SSO".
2. Add the following route to IG:

*Linux*

```
$HOME/.openig/config/routes/sso-authservice.json
```

*Windows*

```
%appdata%\OpenIG\config\routes\sso-authservice.json
```

```
{
  "name": "sso-authservice",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/sso-authservice')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        }
      }
    }
  ]
}
```

```

    },
    "secretsProvider": "SystemAndEnvSecretStore-1",
    "url": "http://openam.example.com:8088/openam/",
    "version": "7"
  }
},
"handler": {
  "type": "Chain",
  "config": {
    "filters": [
      {
        "name": "SingleSignOnFilter-1",
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1",
          "authenticationService": "Example"
        }
      }
    ]
  },
  "handler": "ReverseProxyHandler"
}
}
}

```

Notice the features of the route compared to `sso.json`:

- The route matches requests to `/home/sso-authservice`.
- The `authenticationService` property of `SingleSignOnFilter` refers to `Example`, the name of the example authentication tree in AM. This authentication tree is used for authentication instead of the AM XUI.

### 3. Test the setup:

- If you are logged in to AM, log out and clear any cookies.
- Go to `http://openig.example.com:8080/home/sso-authservice`, and note that the login page is different to that returned in "Authenticate With SSO".

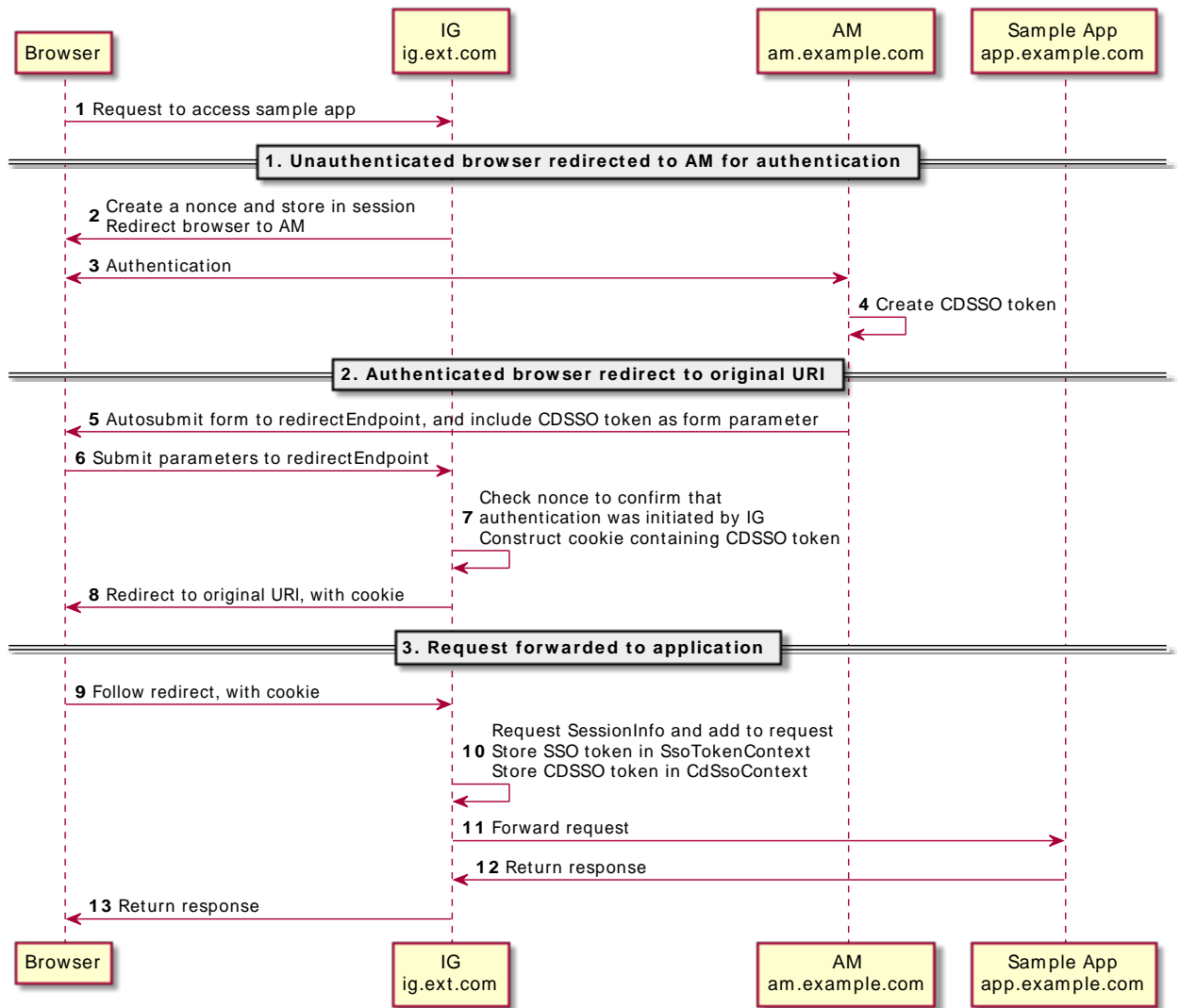
## Authenticating With CDSSO

The SSO mechanism described in "*Single Sign-On and Cross-Domain Single Sign-On*" 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 the AM session info to the request, and stores the SSO token and CDSSO token in the contexts for use by downstream filters and handlers.
- IG forwards the request to the sample app, and the sample app returns the requested resource to the browser.

## Set Up CDSSO

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

Before you start, prepare AM, IG, and the sample application as described in "Example Installation for This Guide".

### 1. Set up AM:

- (For AM 6.5.x and earlier versions) Select **Identities > demo**, and set the demo user password to `Ch4ng31t`.
- Select **Applications > Agents > Identity Gateway**, add an agent with the following values:
  - Agent ID: `ig_agent_cdssso`
  - Password: `password`
  - Redirect URL for CDSSO: `http://openig.ext.com:8080/home/cdssso/redirect`

The agent credentials are the only properties that are used by IG.

For AM 6.5.x and earlier versions, set up an agent as described in "Set Up an IG Agent in AM 6.5 and Earlier".

- (For AM 6.5.3 and later versions) Select **Services > Add a Service**, and add a Validation Service with the following Valid goto URL Resources:
  - `http://openig.ext.com:8080/*`

- `http://openig.ext.com:8080/*?*`

- Select Configure > Global Services > Platform, and add `example.com` as an AM cookie domain.

By default, AM sets host-based cookies. After authentication with AM, requests can be redirected to AM instead of to the resource.

## 2. Set up IG:

- Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

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

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

Linux

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

Windows

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

- Add the following route to IG:

Linux

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

Windows

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

```
{
  "name": "cdsso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/cdsso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
```

```

    "url": "http://openam.example.com:8088/openam",
    "realm": "/",
    "version": "7",
    "agent": {
      "username": "ig_agent_cdsso",
      "passwordSecretId": "agent.secret.id"
    },
    "secretsProvider": "SystemAndEnvSecretStore-1",
    "sessionCache": {
      "enabled": false
    }
  }
},
"handler": {
  "type": "Chain",
  "config": {
    "filters": [
      {
        "name": "CrossDomainSingleSignOnFilter-1",
        "type": "CrossDomainSingleSignOnFilter",
        "config": {
          "redirectEndpoint": "/home/cdsso/redirect",
          "authCookie": {
            "path": "/home",
            "name": "ig-token-cookie"
          },
          "amService": "AmService-1",
          "verificationSecretId": "verify",
          "secretsProvider": {
            "type": "JwkSetSecretStore",
            "config": {
              "jwkUrl": "http://openam.example.com:8088/openam/oauth2/connect/jwk_uri"
            }
          }
        }
      }
    ]
  }
},
"handler": "ReverseProxyHandler"
}
}

```

Notice the following features of the route:

- The route matches requests to `/home/cdsso`.
- The agent password for AmService is provided by a SystemAndEnvSecretStore in the heap.
- The property `verificationSecretId` is configured with a value. If this property is not configured, the filter does not verify the signature of signed `access_tokens`.
- The JwkSetSecretStore specifies the URL to a JWK set on AM, that contains signing keys identified by a `kid`.

The `JwkSetSecretStore` verifies the signature of the token when the value of a `kid` in the JWK set matches a `kid` in the the signed `access_token`.

If the JWT doesn't have a `kid`, or if the JWK set doesn't contain a key with the same value, the `JwkSetSecretStore` looks for valid secrets with the same purpose as the value of `verificationSecretId`.

3. Test the setup:
  - a. If you are logged in to AM, log out and clear any cookies.
  - b. Go to `http://openig.ext.com:8080/home/cdsso`.

The `CrossDomainSingleSignOnFilter` redirects the request to AM for authentication.

- c. Log in to AM as user `demo`, password `Ch4ng31t`.

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

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

## Using WebSocket Notifications to Evict the Session Info Cache

When WebSocket notifications are enabled, IG receives notifications whenever a user logs out of AM, or when an AM session is modified, closed, or times out.

The following procedure gives an example of how to change the configuration in "Authenticating With SSO" and "Authenticating With CDSSO" to evict entries related to the event from the cache. For information about WebSocket notifications, see "WebSocket Notifications" in the *Maintenance Guide*.

### *Evict Entries From the Session Info Cache*

Before you start, set up and test the example in "Authenticating With SSO" or "Authenticating With CDSSO".

- In the `AmService` heap object of your route, enable `sessionCache`:

```
"sessionCache": {  
  "enabled": true  
}
```



## Chapter 6

# Enforcing Policy Decisions From AM

The following sections describe how to set up single sign on for requests in the same domain and in a different domain:

- "About Policy Enforcement"
- "Enforcing AM Policy Decisions In the Same Domain"
- "Enforcing AM Policy Decisions In Different Domains"
- "Using WebSocket Notifications to Evict the Policy Cache"

## About Policy Enforcement

IG as a policy enforcement point (PEP) intercepts requests for a resource, and provides information about the request to AM.

AM as a policy decision point (PDP) evaluates requests based on their context and the configured policies. AM then returns decisions that indicate what actions are allowed or denied, as well as any advices, subject attributes, or static attributes for the specified resources.

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

- If the request is allowed, processing continues.
- If the request is denied with advices, IG checks whether it can respond to the advices. If IG can respond, it sends a redirect and information about how to meet the conditions in the advices.

By default, the request is redirected to AM. If the `SingleSignOnFilter` property `loginEndpoint` is configured, the request is redirected to that endpoint.

- 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.



For information about the `PolicyEnforcementFilter`, see "PolicyEnforcementFilter" in the *Configuration Reference*. For information about AM authentication and session upgrade, see AM's *Authentication and Single Sign-On Guide*.

## Enforcing AM Policy Decisions In the Same Domain

The following procedure gives an example of how to create a policy in AM and configure an agent that can request policy decisions, when IG and AM are in the same domain.

### *Enforce AM Policy Decisions in the Same Domain*

Before you start, prepare AM, IG, and the sample application as described in "Example Installation for This Guide".

1. Set up an AM agent with permission to request policy decisions:
  - a. (For AM 6.5.x and earlier versions) Select  Identities > demo, and set the demo user password to `Ch4ng31t`.
  - b. (For AM 6.5.3 and later versions) Select  Services > Add a Service, and add a Validation Service with the following Valid goto URL Resources:


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

- c. Select Applications > Agents > Identity Gateway, add an agent with the following values:

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

Leave all other values as default.

For AM 6.5.x and earlier versions, set up an agent as described in "Set Up an IG Agent in AM 6.5 and Earlier".

- d. Set up a policy:
      - i. Select  Authorization > Policy Sets > New Policy Set, and add a policy set with the following values:

- Id: `PEP-SSO`
- Resource Types: `URL`

- ii. In the new policy set, add a policy with the following values:

- Name: `IG Policy SSO`
- Resource Type: `URL`
- Resource pattern: `*://*:*/*`

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

This policy protects the home page of the sample application.

- iii. On the Actions tab, add an action to allow HTTP `GET`.
- iv. On the Subjects tab, remove any default subject conditions, add a subject condition for all `Authenticated Users`.
- e. Select Configure > Global Services > Platform, and add `example.com` as an AM cookie domain.

By default, AM sets host-based cookies. After authentication with AM, requests can be redirected to AM instead of to the resource.

## 2. Set up IG:

- a. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcnQ='
```

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

- b. Add the following route to IG, to serve `.css` and other static resources for the sample application:

Linux

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

Windows

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

- c. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/04-pep.json
```

Windows

```
%appdata%\OpenIG\config\routes\04-pep.json
```

```
{
  "name": "pep-sso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/pep-sso')}",
  "heap": [
```

```

{
  "name": "SystemAndEnvSecretStore-1",
  "type": "SystemAndEnvSecretStore"
},
{
  "name": "AmService-1",
  "type": "AmService",
  "config": {
    "agent": {
      "username": "ig_agent",
      "passwordSecretId": "agent.secret.id"
    },
    "secretsProvider": "SystemAndEnvSecretStore-1",
    "url": "http://openam.example.com:8088/openam/",
    "version": "7"
  }
},
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "SingleSignOnFilter-1",
          "type": "SingleSignOnFilter",
          "config": {
            "amService": "AmService-1"
          }
        },
        {
          "name": "PolicyEnforcementFilter-1",
          "type": "PolicyEnforcementFilter",
          "config": {
            "pepRealm": "/",
            "application": "PEP-SSO",
            "ssoTokenSubject": "${contexts.ssoToken.value}",
            "amService": "AmService-1"
          }
        }
      ]
    }
  },
  "handler": "ReverseProxyHandler"
}
}

```

For information about how to set up the IG route in Studio, see "Policy Enforcement in Structured Editor" in the *Studio User Guide* or "Protecting a Web App With Freeform Designer" in the *Studio User Guide*.

For an example route that uses `claimsSubject` instead of `ssoTokenSubject` to identify the subject, see "Example Policy Enforcement Using `claimsSubject`" in the *Configuration Reference*.

3. Test the setup:
  - a. If you are logged in to AM, log out and clear any cookies.

- b. Go to `http://openig.example.com:8080/home/pep-ss0`.

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.

- c. Log in to AM as user `demo`, password `Ch4ng31t`.

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


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

## Enforcing AM Policy Decisions In Different Domains

The following procedure gives an example of how to create a policy in AM and configure an agent that can request policy decisions, when IG and AM are in different domains.

### *Enforce AM Policy Decisions in a Different Domain*

Before you start, prepare AM, IG, and the sample application as described in "Example Installation for This Guide".

1. Set up AM:
  - a. (For AM 6.5.x and earlier versions) Select  Identities > demo, and set the demo user password to `Ch4ng31t`.
  - b. (For AM 6.5.3 and later versions) Select  Services > Add a Service, and add a Validation Service with the following Valid goto URL Resources:
    - `http://openig.ext.com:8080/*`
    - `http://openig.ext.com:8080/*?*`
  - c. Select Applications > Agents > Identity Gateway, add an agent with the following values:
    - Agent ID: `ig_agent_cdss0`
    - Password: `password`
    - Redirect URL for CDSSO: `http://openig.ext.com:8080/home/pep-cdss0/redirect`

The agent credentials are the only properties that are used by IG.

For AM 6.5.x and earlier versions, set up an agent as described in "Set Up an IG Agent in AM 6.5 and Earlier".

d. Set up a policy:

- i. Select **Authorization > Policy Sets > New Policy Set**, and add a policy set with the following values:

- Id: `PEP-CDSSO`
- Resource Types: `URL`

- ii. In the new policy set, add a policy with the following values:

- Name: `IG Policy CDSSO`
- Resource Type: `URL`
- Resource pattern: `*://*:*/*`
- Resource value: `http://app.example.com:8081/home/pep-cdsso*`

This policy protects the home page of the sample application.

- iii. On the Actions tab, add an action to allow HTTP `GET`.

- iv. On the Subjects tab, remove any default subject conditions, add a subject condition for all `Authenticated Users`.

- e. Select **Configure > Global Services > Platform**, and add `example.com` as an AM cookie domain.

By default, AM sets host-based cookies. After authentication with AM, requests can be redirected to AM instead of to the resource.

2. Set up IG:

- a. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

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

- b. Add the following route to IG, to serve `.css` and other static resources for the sample application:

*Linux*

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

*Windows*

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/04-pep-cdsso.json
```

Windows

```
%appdata%\OpenIG\config\routes\04-pep-cdsso.json
```

```
{
  "name": "pep-cdsso",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/pep-cdsso')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent_cdsso",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://openam.example.com:8088/openam/",
        "version": "7"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "CrossDomainSingleSignOnFilter-1",
          "type": "CrossDomainSingleSignOnFilter",
          "config": {
            "redirectEndpoint": "/home/pep-cdsso/redirect",
            "authCookie": {
              "path": "/home",
              "name": "ig-token-cookie"
            }
          },
          "amService": "AmService-1"
        }
      ],
      "name": "PolicyEnforcementFilter-1",
```

```
    "type": "PolicyEnforcementFilter",
    "config": {
      "pepRealm": "/",
      "application": "PEP-CDSSO",
      "ssoTokenSubject": "${contexts.cdsso.token}",
      "amService": "AmService-1"
    }
  ],
  "handler": "ReverseProxyHandler"
}
}
```

For information about how to set up the IG route in Studio, see "Policy Enforcement for CDSSO in Structured Editor" in the *Studio User Guide*.

3. Test the setup:
  - a. If you are logged in to AM, log out and clear any cookies.
  - b. Go to <http://openig.ext.com:8080/home/pep-cdsso>.  
IG redirects you to AM for authentication.
  - c. Log in to AM as user **demo**, password **Ch4ng31t**.

When you have authenticated, AM redirects you back to the request URL, and IG requests a policy decision. AM returns a policy decision that grants access to the sample application.

## Using WebSocket Notifications to Evict the Policy Cache

When WebSocket notifications are enabled, IG receives notifications whenever AM creates, deletes, or changes a policy.

The following procedure gives an example of how to change the configuration in "Enforcing AM Policy Decisions In the Same Domain" and "Enforcing AM Policy Decisions In Different Domains" to evict outdated entries from the policy cache. For information about WebSocket notifications, see "WebSocket Notifications" in the *Maintenance Guide*.

### Evict Entries From the Policy Cache

1. Set up and test the example in "Enforcing AM Policy Decisions In the Same Domain" or "Enforcing AM Policy Decisions In Different Domains".
2. In the PolicyEnforcementFilter, enable **cache**:

```
"cache": {
  "enabled": true
}
```



## Chapter 7

# Hardening Authorization With Advice From AM

To protect sensitive resources, AM policies can be configured with additional conditions to harden the authorization. When AM communicates these policy decisions to IG, the decision includes advices to indicate what extra conditions the user must meet.

Conditions can include requirements to access the resource over a secure channel, access during working hours, or to authenticate again at a higher authentication level. For more information, see AM's *Authorization Guide*.

The following sections build on the policies in "*Enforcing Policy Decisions From AM*" to step up the authentication level:

- "Stepping Up the Authentication Level for an AM Session"
- "Increasing Authorization for a Single Transaction"

## Stepping Up the Authentication Level for an AM Session

When you step up the authentication level for an AM session, the authorization is verified and then captured as part of the AM session, and the user-agent is authorized to that authentication level for the duration of the AM session.

This section uses the policies you created in "*Enforcing AM Policy Decisions In the Same Domain*" and "*Enforcing AM Policy Decisions In Different Domains*", adding an authorization policy with a *Authentication by Service* environment condition. Except for the paths where noted, procedures for single domain and cross-domain are the same.

After the user-agent redirects the user to AM, if the user is not already authenticated they are presented with a login page. If the user is already authenticated, or after they authenticate, they are presented with a second page asking for a verification code to meet the *AuthenticateToService* environment condition.

### *Set Up an AM Authentication Chain*


Before you start, set up one of the following examples:

- For SSO, "*Enforcing AM Policy Decisions In the Same Domain*".
- For CDSSO, "*Enforcing AM Policy Decisions In Different Domains*".

1. In the AM console, add an environment condition to the policy:

- a. Select a policy set:
  - For SSO, select  Authorization > Policy Sets > PEP-SSO.
  - For CDSSO, select  Authorization > Policy Sets > PEP-CDSSO.
- b. In the policy, select Environments, and add the following environment condition:
  - All of
  - Type: Authentication by Service
  - Authenticate to Service: VerificationCodeLevel1

## 2. Set up client-side and server-side scripts:

- a. Select  Scripts > Scripted Module - Client Side, and replace the default script with the following script:

*For AM 6 and later versions*

```

/*
 * 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 == "clientScriptOutputData") element.value = "{}";
        var json = JSON.parse(element.value);
        json["answer"] = event.target.value;
        element.value = JSON.stringify(json);
    };
    parent.appendChild(input);
}
    
```

```

    var fieldset = document.forms[0].getElementsByTagName("fieldset")[0];
    fieldset.prepend(parent);
}

if (document.readyState !== 'loading') {
    callback();
} else {
    document.addEventListener("DOMContentLoaded", callback);
}

```

For AM 5 and earlier versions

```

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.

- b. Select </> Scripts > Scripted Module - Server Side, and replace the default script with the following script:

```

username = 'demo'
logger.error('username: ' + username)

// Test whether the user 'demo' enters the correct validation code
data = JSON.parse(clientScriptOutputData);
answer = data.answer;



if (answer !== '123456') {
    logger.error('Authentication Failed !!!')
    authState = FAILED;
} else {
    logger.error('Authenticated !!!')
    authState = SUCCESS;
}

```

Leave all other values as default.

This server-side script tests that the user `demo` has entered `123456` as the verification code.

3. Add an authentication module:

- a. In the top level realm, select  Authentication > Modules, and add a module with the following settings:
    - Name: `VerificationCodeLevel1`
    - Type: `Scripted Module`
  - b. 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`
4. Add the authentication module to an authentication chain:
    - a. Select  Authentication > Chains, and add a chain called `VerificationCodeLevel1`.
    - b. Add a module with the following settings:
      - Select Module: `VerificationCodeLevel1`
      - Select Criteria: `Required`

### *Test the Setup*

1. Log out of AM.
2. Access the route:
  - For SSO, go to `http://openig.example.com:8080/home/pep-sso`.
  - For CDSSO, go to `http://openig.ext.com:8080/home/pep-cdsso`.

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 `Ch4ng3!t`.

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

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

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

AM upgrades the authentication level for the session to 1, and grants access to the sample application. If you try to access the sample application again in the same session, you don't need to provide the verification code.

## Increasing Authorization for a Single Transaction

Transactional authorization improves security by requiring a user to perform additional actions when trying to access a resource protected by an AM policy. For example, they must reauthenticate to an authentication module or respond to a push notification on their mobile device.

Performing the additional action successfully grants access to the protected resource, but only once. Additional attempts to access the resource require the user to perform the configured actions again.

This section builds on the example in "Stepping Up the Authentication Level for an AM Session", adding a simple authorization policy with a **Transaction** environment condition. Each time the user-agent tries to access the protected resource, the user must reauthenticate to an authentication module by providing a verification code.

This feature is supported with AM 5.5 and later versions.

### *Use Transactional Authorization*

Before you start, configure AM as described in "Set Up an AM Authentication Chain". The IG configuration is not changed.

1. In the AM console, add a new Environment condition:
  - a. Select the policy set:
    - For SSO, select Authorization > Policy Sets > PEP-SSO.
    - For CDSSO, select Authorization > Policy Sets > PEP-CDSSO.
  - b. In the IG policy, select Environments and add another environment condition:
    - **All of**
    - Type: **Transaction**
    - Authentication strategy: **Authenticate To Module**
    - Strategy specifier: **TxVerificationCodeLevel5**
2. Set up client-side and server-side scripts:
  - a. Select </> Scripts > New Script, and add the following client-side script:
    - Name: **Tx Scripted Module - Client Side**

- Script Type: **Client-side Authentication**

```
/*
 * 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 TX code";
  input.name = "answer";
  input.id = "answer";
  input.value = "";
  input.oninput = function(event) {
    var element = document.getElementById("clientScriptOutputData");
    if (!element.value || element.value == "clientScriptOutputData") element.value = "{}";
    var json = JSON.parse(element.value);
    json["answer"] = event.target.value;
    element.value = JSON.stringify(json);
  };
  parent.appendChild(input);

  var fieldset = document.forms[0].getElementsByTagName("fieldset")[0];
  fieldset.prepend(parent);
}

if (document.readyState !== 'loading') {
  callback();
} else {
  document.addEventListener("DOMContentLoaded", callback);
}
```

This client-side script adds a field to the AM form, in which the user is required to enter a TX code. The script formats the entered code as a JSON object, as required by the server-side script.

- Select `</>` Scripts > New Script, and add the following server side script:
  - Name: **Tx Scripted Module - Server Side**

- Script Type: **Server-side Authentication**

```
username = 'demo'
logger.error('username: ' + username)

// Test whether the user 'demo' enters the correct validation code
data = JSON.parse(clientScriptOutputData);
answer = data.answer;

if (answer !== '789') {
  logger.error('Authentication Failed !!!')
  authState = FAILED;
} else {
  logger.error('Authenticated !!!')
  authState = SUCCESS;
}
```

This server-side script tests that the user **demo** has entered **789** as the verification code.

3. Add an authentication module:
  - a. Select **Authentication > Modules**, and add a module with the following settings:
    - Name: **TxVerificationCodeLevel5**
    - Type: **Scripted Module**
  - b. In the authentication module, enable the option for client-side script, and select the following options:
    - Client-side Script: **Tx Scripted Module - Client Side**
    - Server-side Script: **Tx Scripted Module - Server Side**
    - Authentication Level: **5**

### Test the Setup

1. Log out of AM.
2. Go to your route:
  - For SSO, go to <http://openig.example.com:8080/home/pep-ss0>.
  - For CDSSO, go to <http://openig.ext.com:8080/home/pep-cdss0>.

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

3. Log in to AM as user **demo**, password **Ch4ng31t**.

AM creates a session with the default authentication level **0**, and IG requests a policy decision.

4. Enter the verification code **123456** to upgrade the authorization level for the session to **1**.

The authentication module you configured for transactional authorization requires authentication level **5**, so AM issues a **TransactionConditionAdvice**.

5. In the transaction upgrade window, enter the verification code **789**.

AM upgrades the authentication level for this policy evaluation to **5**, and then returns a policy decision that grants a one-time access to the sample application. If you try to access the sample application again, you must enter the code again.



## Chapter 8

# Protecting Against CSRF Attacks

In a Cross Site Request Forgery (CSRF) attack, a user unknowingly executes a malicious request on a website where they are authenticated. A CSRF attack usually includes a link or script in a web page. When a user accesses the link or script, the page executes an HTTP request on the site where the user is authenticated.

CSRF attacks interact with HTTP requests as follows:

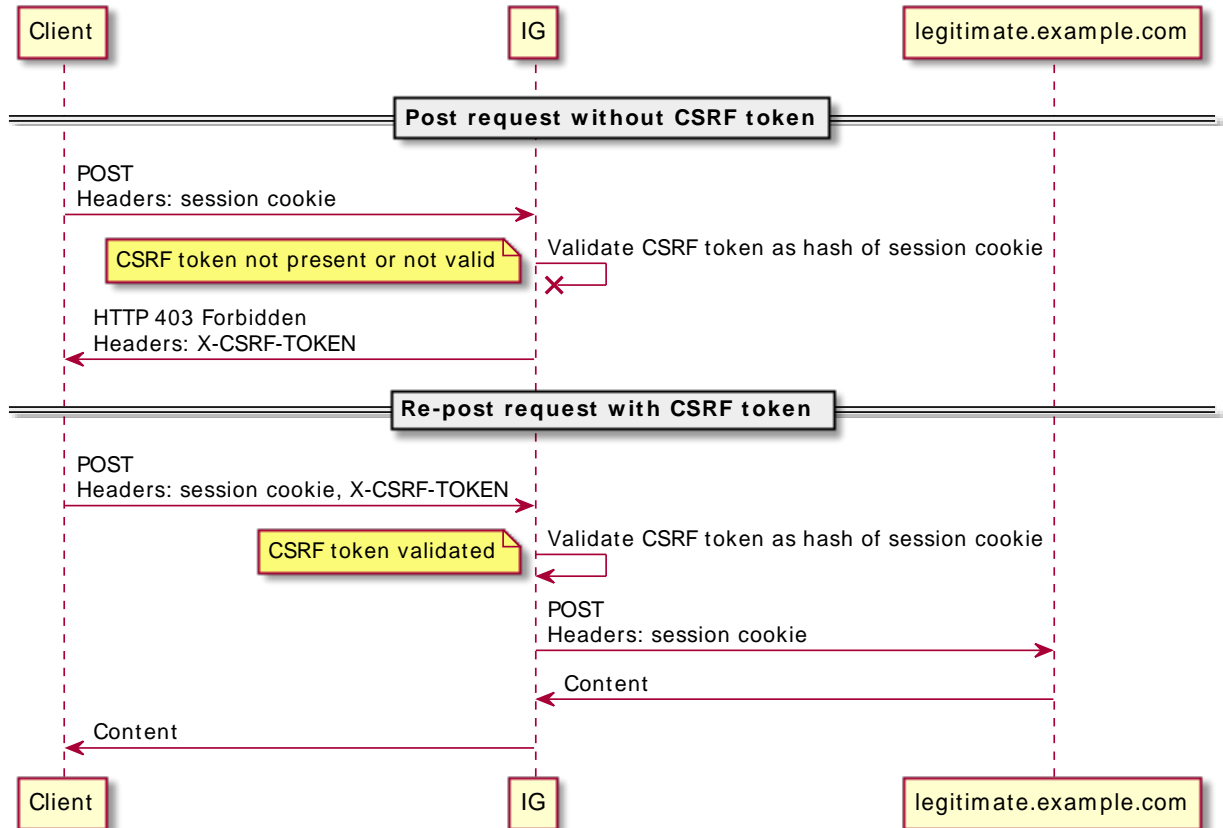
- CSRF attacks can execute POST, PUT, and DELETE requests on the targeted server. For example, a CSRF attack can transfer funds out of a bank account or change a user's password.
- Because of same-origin policy, CSRF attacks **cannot** access any response from the targeted server.

When IG processes POST, PUT, and DELETE requests, it checks a custom HTTP header in the request. If a CSRF token is not present in the header or not valid, IG rejects the request and returns a valid CSRF token in the response.

Rogue websites that attempt CSRF attacks operate in a different website domain to the targeted website. Because of same-origin policy, rogue websites can't access a response from the targeted website, and cannot, therefore, access the response or CSRF token.

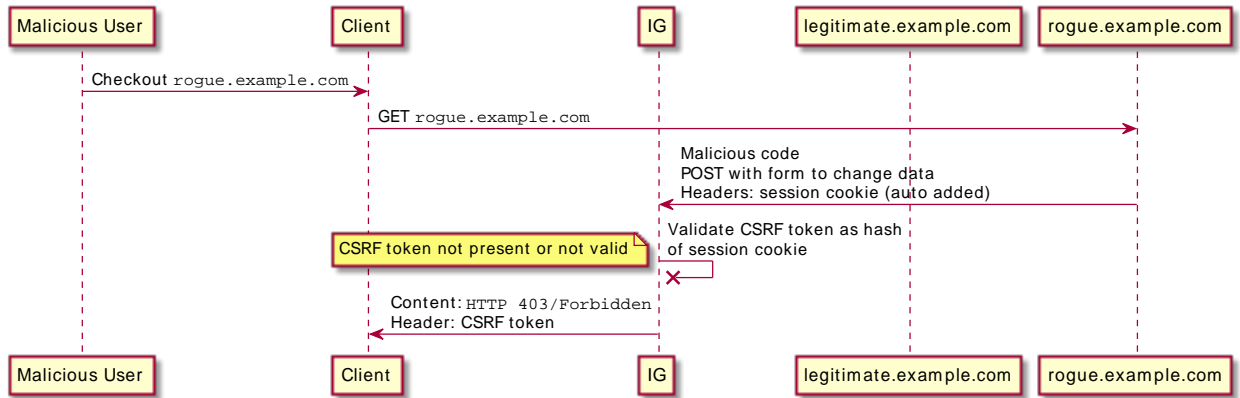
The following example shows the data flow when an authenticated user sends a POST request to an application protected against CSRF:

## Flow of Requests From Authenticated User to Application Protected Against CSRF



The following example shows the data flow when an authenticated user sends a POST request from a rogue site to an application protected against CSRF:

## Flow of Requests From Rogue Site to Application Protected Against CSRF



## Protect Against CSRF Attacks

1. Set up SSO, so that AM authenticates users to the sample app through IG:
  - a. Set up AM and IG as described in "Authenticate With SSO".
  - b. Remove the condition in `sso.json`, so that the route matches all requests:
 

```
"condition": "${matches(request.uri.path, '^/home/sso')}"
```
2. Test the setup without CSRF protection:
  - a. Go to `http://openig.example.com:8080/bank/index`, and log in to the Sample App Bank through AM, as user `demo`, password `Ch4ng3!t`.
  - b. Send a bank transfer of \$10 to Bob, and note that the transfer is successful.
  - c. Go to `http://localhost:8081/bank/attack-autosubmit` to simulate a CSRF attack.

When you access this page, a hidden HTML form is automatically submitted to transfer \$1000 to the rogue user, using the IG session cookie to authenticate to the bank.

In the bank transaction history, note that \$1000 has been debited.

3. Test the setup with CSRF protection:
  - a. In IG, replace `sso.json` with the following route:

```
{
  "name": "Csrf",
  "baseURI": "http://app.example.com:8081",
  "heap": [
```

```
{
  "name": "SystemAndEnvSecretStore-1",
  "type": "SystemAndEnvSecretStore"
},
{
  "name": "AmService-1",
  "type": "AmService",
  "config": {
    "agent": {
      "username": "ig_agent",
      "passwordSecretId": "agent.secret.id"
    },
    "secretsProvider": "SystemAndEnvSecretStore-1",
    "url": "http://openam.example.com:8088/openam/",
    "version": "7"
  }
},
{
  "name": "FailureHandler-1",
  "type": "StaticResponseHandler",
  "config": {
    "status": 403,
    "headers": {
      "Content-Type": [ "text/plain" ]
    },
    "entity": "Request forbidden"
  }
}
],
"handler": {
  "type": "Chain",
  "config": {
    "filters": [
      {
        "name": "SingleSignOnFilter-1",
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      },
      {
        "name": "CsrfFilter-1",
        "type": "CsrfFilter",
        "config": {
          "cookieName": "iPlanetDirectoryPro",
          "failureHandler": "FailureHandler-1"
        }
      }
    ]
  }
},
"handler": "ReverseProxyHandler"
}
}
```

Notice the following features of the route compared to `sso.json`:

- The Csrffilter checks the AM session cookie for the `X-CSRF-Token` header. If a CSRF token is not present in the header or not valid, the filter rejects the request and provides a valid CSRF token in the header.
- b. Go to `http://openig.example.com:8080/bank/index`, and send a bank transfer of \$10 to Alice. Because there is no CSRF token, IG responds with an HTTP 403, and provides the token.
- c. Send the transfer again, and note that because the CSRF token is provided the transfer is successful.
- d. Go to `http://localhost:8081/bank/attack-autosubmit` to automatically send a rogue transfer.

Because there is no CSRF token, IG rejects the request and provides the CSRF token. However, because the rogue site is in a different domain to `openig.example.com` it can't access the CSRF token.

## Chapter 9

# Acting As a SAML 2.0 Service Provider

The following sections describe IG's role as a SAML 2.0 service provider, and give an example of how to set up IG, with AM as an identity provider:

- "About SAML 2.0 SSO and Federation"
- "Set Up SAML 2.0 SSO and Federation"
- "Using a Non-Transient NameID Format"
- "Example Fedlet Files"

For information about how to set up multiple service providers, see "*SAML 2.0 and Multiple Applications*".

## About SAML 2.0 SSO and Federation

The IG federation component implements SAML 2.0, to validate users and log them in to protected applications.

The SAML 2.0 standard describes the messages that providers exchange, and how 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 in SAML and federation:

- *Identity Provider* (IDP): The service that manages the user identity.
- *Service Provider* (SP): The service that a user wants to access. IG acts as a SAML 2.0 SP for SSO, providing users with an interface to applications that don't support SAML 2.0.
- *Circle of trust* (CoT): An IDP and SP that participate in the federation.

When an IDP and an SP participate in a federation, they agree on what security information to exchange, and mutually configure access to each other's services.

After an IDP authenticates a user, it provides the SP with SAML assertions that attest to which user is authenticated, when the authentication succeeded, how long the assertion is valid, and so on. The SP uses the SAML assertions to make authorization decisions, for example, to let an authenticated user complete a purchase that gets charged to the user's account at the IDP.

The IDP and SP usually communicate about a user identified by a name identifier. In SP-initiated SSO and IDP-initiated SSO, the NameID format can be any format supported by the IDP. For more information, see "Using a Non-Transient NameID Format".

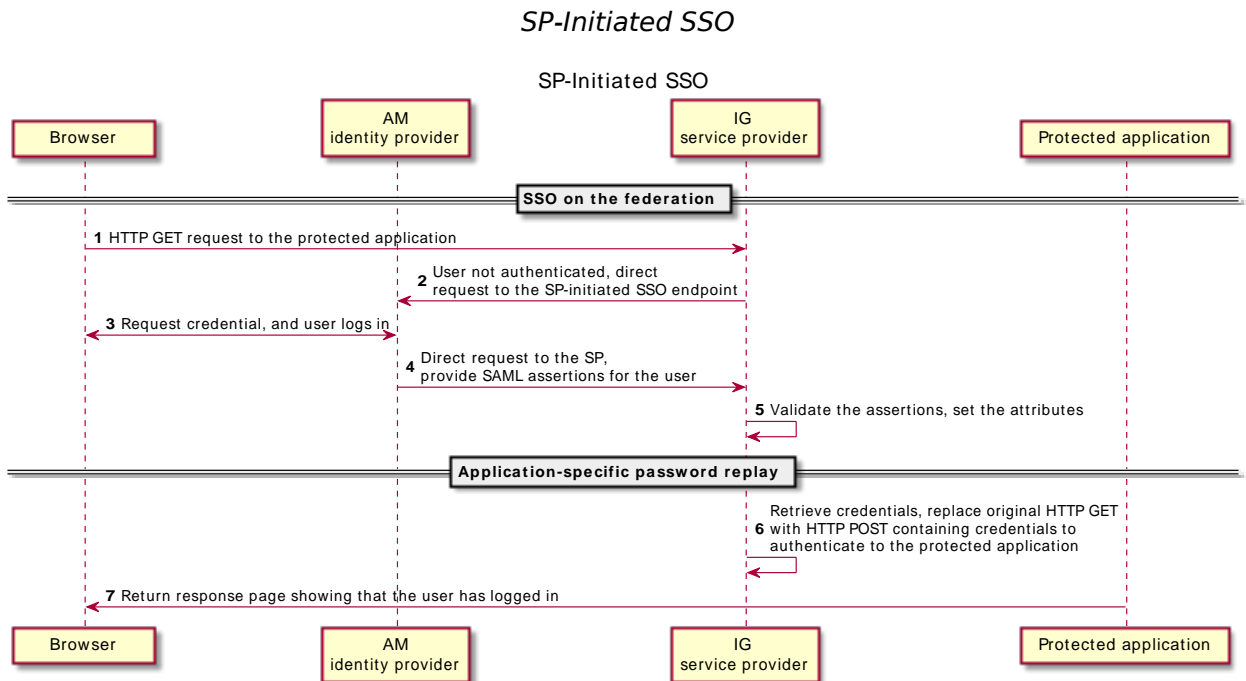
SAML assertions can be signed and encrypted. ForgeRock recommends using \*SHA-256 variants (rsa-sha256 or ecdsa-sha256).

SAML assertions can contain configurable attribute values, such as user meta-information or anything else provided by the IDP. The attributes of a SAML assertion can contain one or more values, made available as a list of strings. Even if an attribute contains a single value, it is made available as a list of strings.

## About SP-Initiated SSO

SP-initiated SSO occurs when a user attempts to access a protected application directly through the SP. Because the user's federated identity is managed by the IDP, the SP sends a SAML authentication request to the IDP. After the IDP authenticates the user, it provides the SP with a SAML assertion for the user.

The following sequence diagram shows the flow of information in SP-initiated SSO, when IG acts as a SAML 2.0 SP:



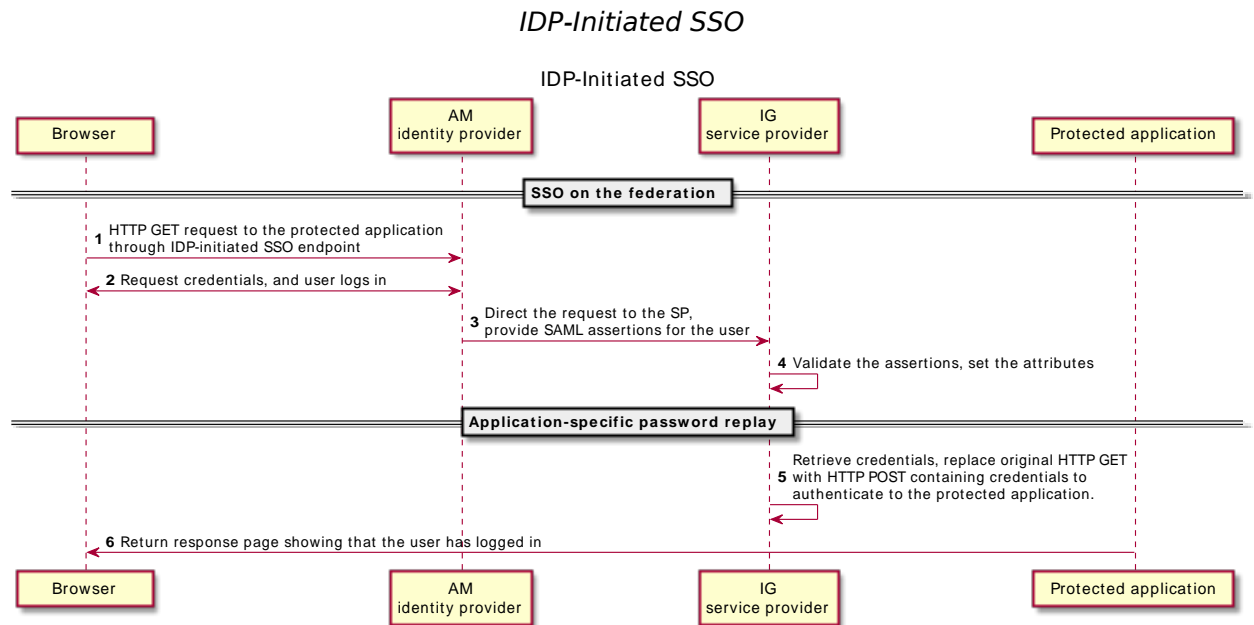
## About IDP-Initiated SSO

IDP-initiated SSO occurs when a user attempts to access a protected application, using the IDP for authentication. The IDP sends an unsolicited authentication statement to the SP.

Before IDP-initiated SSO can occur:

- The user must access a link on the IDP that refers to the remote SP.
- The user must authenticate to the IDP.
- The IDP must be configured with links that refer to the SP.

The following sequence diagram shows the flow of information in IDP-initiated SSO when IG acts as a SAML 2.0 SP:



## Set Up SAML 2.0 SSO and Federation

For examples of the federation configuration files, see "Example Fedlet Files". To set up multiple SPs, work through this section, and then "SAML 2.0 and Multiple Applications".

### 1. Set up the network:

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



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

Traffic to the application is proxied through IG, using the host name `sp.example.com`.

## 2. Configure a Java Fedlet:

### Note

The SAML library component validates the SP's AssertionConsumerService Location against the incoming IDP SAML Assertion, based on the request information, including the port. In `sp.xml`, always specify the port in the `Location` value of `AssertionConsumerService`, even when using defaults of `443` or `80`, as follows:

```
<AssertionConsumerService isDefault="true" index="0"
  Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="https://sp.example.com:443/
  fedletapplication"/>
```

For more information about Java Fedlets, see *Creating and Configuring the Fedlet in AM's SAML v2.0 Guide*.

- a. Copy and unzip the fedlet zip file, `Fedlet-7.0.2.zip`, delivered with the AM installation, into a local directory.

```
$ unzip $HOME/openam/Fedlet-7.0.2.zip
Archive: Fedlet-7.0.0-SNAPSHOT.zip
creating: conf/
inflating: README
inflating: conf/FederationConfig.properties
inflating: conf/fedlet.cot-template
inflating: conf/idp-extended.xml-template
inflating: conf/sp-extended.xml-template
inflating: conf/sp.xml-template
inflating: fedlet.war
```

- b. For AM 6.5.2 and earlier versions, add the following lines to `FederationConfig.properties`:

```
# Specifies implementation for
# org.forgerock.openam.federation.plugin.rooturl.RootUrlProvider interface.
# This property defines the default base url provider.
com.sun.identity.plugin.root.url.class.default=org.forgerock.openam.federation.plugin.rooturl.impl.FedletR
```

- c. In each file, search and replace the following properties:


- `IDP_ENTITY_ID`: replace with `openam`
- `FEDLET_ENTITY_ID`: replace with `sp`
- `FEDLET_PROTOCOL://FEDLET_HOST:FEDLET_PORT/FEDLET_DEPLOY_URI`: replace with `http://sp.example.com:8080/saml`
- `fedletcot` and `FEDLET_COT`: replace with `Circle of Trust`

- `sp.example.com:8080/saml/fedletapplication`: replace with `sp.example.com:8080/saml/fedletapplication/metaAlias/sp`

d. Save the files as `.xml`, without the `-template` extension.

By default, AM as an IDP uses the NameID format `urn:oasis:names:tc:SAML:2.0:nameid-format:transient` to communicate about a user. For information about using a different NameID format, see "Using a Non-Transient NameID Format".


### 3. Set up AM:

a. Select  Identities, and add a user with the following values:


- ID/username: `george`
- First name: `george`
- Last name: `C0stanza`

Note that, for this example, the last name must be the same as the password.


- Password: `C0stanza`

b. Select  Applications > Federation > Circles of Trust, and add a circle of trust called `Circle of Trust`, with the default settings.

c. Set up a remote service provider:

- Select  Applications > Federation > Entity Providers, and add a remote entity provider.
- Drag in or import `sp.xml` created in Step 2.
- Select Circles of Trust: `Circle of Trust`

d. Set up a hosted identity provider:

- Select  Applications > Federation > Entity Providers, and add a hosted entity provider with the following values:
  - Entity ID: `openam`
  - Entity Provider Base URL: `http://openam.example.com:8088/openam`
  - Identity Provider Meta Alias: `idp`
  - Circles of Trust: `Circle of Trust`
- Select Assertion Processing > Attribute Mapper, map the following SAML attribute keys and values, and then save your changes:

- SAML Attribute: `cn`, Local Attribute: `cn`
- SAML Attribute: `sn`, Local Attribute: `sn`

iii. In a terminal, export the XML-based metadata for the IPD:

```
$ curl -v \
--output idp.xml \
"http://openam.example.com:8088/openam/saml2/jsp/exportmetadata.jsp?entityid=openam"
```

The `idp.xml` file is created locally.

#### 4. Set up IG:

a. Retrieve the Fedlet configuration files:

- Copy the edited fedlet files, and the exported `idp.xml` file into the IG configuration, at `$HOME/.openig/SAML`.

```
$ ls -l $HOME/.openig/SAML
FederationConfig.properties
fedlet.cot
idp-extended.xml
idp.xml
sp-extended.xml
sp.xml
```

b. In `config.json`, comment out or remove the `baseURI`:

```
{
  "handler": {
    "_baseURI": "http://app.example.com:8081",
    ...
  }
}
```

Requests to the `SamFederationHandler` must not be rebased, because the request URI must match the endpoint in the SAML metadata.

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

Linux

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

Windows

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

d. Add the following route to IG:

Linux

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

Windows

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

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

Notice the following features of the route:

- The route matches requests to `/saml`.
- After authentication, the `SamlFederationHandler` extracts `cn` and `sn` from the SAML assertion, and maps them to the `SessionContext`, at `session.username` and `session.password`.
- The handler stores the subject name as a string in the session field `session.sp-subject-name`, which is named by the `subjectMapping` property. By default, the subject name is stored in the session field `session.subjectName`.
- The handler redirects the request to the `/federate` route.
- The route uses the `JwtSession` implementation, meaning it stores encrypted session information in a browser cookie. The name is a reference to the `JwtSession` object defined in `config.json`. For details, see "JwtSession" in the *Configuration Reference*.

e. Add the following route to IG:

Linux

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

Windows

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

```
{
```

```

"name": "federate",
"condition": "${matches(request.uri.path, '^/home/federate')}",
"session": "JwtSession",
"baseURI": "http://app.example.com:8081",
"handler": {
  "type": "DispatchHandler",
  "config": {
    "bindings": [
      {
        "condition": "${empty session.username}",
        "handler": {
          "type": "StaticResponseHandler",
          "config": {
            "status": 302,
            "reason": "Found",
            "headers": {
              "Location": [
                "http://sp.example.com:8080/saml/SPInitiatedSSO?metaAlias=/sp"
              ]
            }
          }
        }
      }
    ],
    {
      "handler": {
        "type": "Chain",
        "config": {
          "filters": [
            {
              "type": "HeaderFilter",
              "config": {
                "messageType": "REQUEST",
                "add": {
                  "x-username": ["${session.username[0]}"],
                  "x-password": ["${session.password[0]}"]
                }
              }
            }
          ],
          "handler": "ReverseProxyHandler"
        }
      }
    }
  ]
}
}

```

Notice the following features of the route:

- The route matches requests to `/home/federate`.
- If the user is not authenticated with AM, the username is not populated in the context. The `DispatchHandler` then dispatches the request to the `StaticResponseHandler`, which redirects it to the SP-initiated SSO endpoint.

If the credentials are in the context, or after successful authentication, the `DispatchHandler` dispatches the request to the Chain.

- The `HeaderFilter` adds headers for the first value for the `username` and `password` attributes of the SAML assertion.
- The route uses the `JwtSession` implementation, meaning it stores encrypted session information in a browser cookie. The name is a reference to the `JwtSession` object defined in `config.json`. For details, see "JwtSession" in the *Configuration Reference*.

#### Tip

For more control over the URL where the user agent is redirected, use the `RelayState` query string parameter in the URL of the redirect `Location` header. `RelayState` specifies where to redirect the user when the SAML 2.0 web browser SSO process is complete. It overrides the `redirectURI` set in the `SamlFederationHandler`.

The `RelayState` value must be URL-encoded. When using an expression, use a function to encode the value. For example, use `${urlencodeQueryParamNameOrValue(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=
    ${urlencodeQueryParamNameOrValue(contexts.router.originalUri)}"
  ]
}
```

- f. Restart IG.
5. Test the setup:
    - a. Log out of AM, and test the setup with the following links:
      - IDP-initiated SSO
      - SP-initiated SSO
    - b. Log in to AM with username `george` and password `C0stanza`.

IG returns the response page showing that the George has logged in.

## Using a Non-Transient NameID Format

By default, AM as an IDP uses the NameID format `urn:oasis:names:tc:SAML:2.0:nameid-format:transient`. For more information, see Hosted Identity Provider Configuration Properties in AM's *SAML v2.0 Guide*.

When the IDP uses another NameID format, configure IG to use that NameID format by editing the Fedlet configuration file `sp-extended.xml`:

- To use the NameID value provided by the IDP, add the following attribute:

```
<Attribute name="useNameIDAsSPUserID">
  <Value>true</Value>
</Attribute>
```

- To use an attribute from the assertion, add the following attribute:

```
<Attribute name="autofedEnabled">
  <Value>true</Value>
</Attribute>
<Attribute name="autofedAttribute">
  <Value>sn</Value>
</Attribute>
```

This example uses the value in **SN** to identify the subject.

Although IG supports the **persistent** NameID format, IG does not store the mapping. To configure this behavior, edit the file `sp-extended.xml`:

- To disable attempts to persist the user mapping, add the following attribute:

```
<Attribute name="spDoNotWriteFederationInfo">
  <Value>true</Value>
</Attribute>
```

- To enable attempts to persist the user mapping, add the following attribute:

```
<Attribute name="spDoNotWriteFederationInfo">
  <Value>false</Value>
</Attribute>
```

If a login request doesn't contain a NameID format query parameter, the value is defined by the presence and content of the NameID format list for the SP and IDP. For example, an SP-initiated login can be constructed with the binding and **NameIDFormat** as a parameter, as follows:

```
http://fedlet.example.org:7070/fedlet/SPInitiatedSSO?binding=urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST&NameIDFormat=urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified
```

When the NameID format is provided in a list, it is resolved as follows:

- If both the IDP and SP have a list, the first matching NameID format in the lists.
- If either the IDP or SP list is empty, the first NameID format in the other list.
- If neither the IDP nor SP has a list, then AM defaults to **transient**, and IG defaults to **persistent**.

# Example Fedlet Files

## Summary of Fedlet Files

File	Description
<code>FederationConfig.properties</code>	Defines fedlet properties
<code>fedlet.cot</code>	Circle of trust for IG and the IDP
<code>idp.xml</code>	Standard metadata for the IDP
<code>idp-extended.xml</code>	Metadata extensions for the IDP
<code>sp.xml</code>	Standard metadata for the IG SP
<code>sp-extended.xml</code>	Metadata extensions for the IG SP

### *FederationConfig.properties*

The following example of `$HOME/.openid/SAML/FederationConfig.properties` defines the fedlet properties:

```
#
# DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.
#
# Copyright (c) 2006 Sun Microsystems Inc. All Rights Reserved
#
# The contents of this file are subject to the terms
# of the Common Development and Distribution License
# (the License). You may not use this file except in
# compliance with the License.
#
# You can obtain a copy of the License at
# https://opensso.dev.java.net/public/CDDLv1.0.html or
# opensso/legal/CDDLv1.0.txt
# See the License for the specific language governing
# permission and limitations under the License.
#
# When distributing Covered Code, include this CDDL
# Header Notice in each file and include the License file
# at opensso/legal/CDDLv1.0.txt.
# If applicable, add the following below the CDDL Header,
# with the fields enclosed by brackets [] replaced by
# your own identifying information:
# "Portions Copyrighted [year] [name of copyright owner]"
#
# $Id: FederationConfig.properties,v 1.21 2010/01/08 22:41:28 exu Exp $
#
# Portions Copyright 2016-2020 ForgeRock AS.

# If a component wants to use a different datastore provider than the
# default one defined above, it can define a property like follows:
# com.sun.identity.plugin.datastore.class.<componentName>=<provider class>

# com.sun.identity.plugin.configuration.class specifies implementation for
# com.sun.identity.plugin.configuration.ConfigurationInstance interface.
com.sun.identity.plugin.configuration.class=com.sun.identity.plugin.configuration.impl.FedletConfigurationImpl
```



```

# Specifies implementation for
# com.sun.identity.plugin.datastore.DataStoreProvider interface.
# This property defines the default datastore provider.
com.sun.identity.plugin.datastore.class.default=com.sun.identity.plugin.datastore.impl.FedletDataStoreProvider

# Specifies implementation for
# org.forgerock.openam.federation.plugin.rooturl.RootUrlProvider interface.
# This property defines the default base url provider.
com.sun.identity.plugin.root.url.class.default=org.forgerock.openam.federation.plugin.rooturl.impl.FedletRootUrlPro

# com.sun.identity.plugin.log.class specifies implementation for
# com.sun.identity.plugin.log.Logger interface.
com.sun.identity.plugin.log.class=com.sun.identity.plugin.log.impl.FedletLogger

# com.sun.identity.plugin.session.class specifies implementation for
# com.sun.identity.plugin.session.SessionProvider interface.
com.sun.identity.plugin.session.class=com.sun.identity.plugin.session.impl.FedletSessionProvider

# com.sun.identity.plugin.monitoring.agent.class specifies implementation for
# com.sun.identity.plugin.monitoring.FedMonAgent interface.
com.sun.identity.plugin.monitoring.agent.class=com.sun.identity.plugin.monitoring.impl.FedletAgentProvider

# com.sun.identity.plugin.monitoring.saml2.class specifies implementation for
# com.sun.identity.plugin.monitoring.FedMonSAML2Svc interface.
com.sun.identity.plugin.monitoring.saml2.class=com.sun.identity.plugin.monitoring.impl.FedletMonSAML2SvcProvider

# com.sun.identity.saml.xmlsig.keyprovider.class specified the implementation
# class for com.sun.identity.saml.xmlsig.KeyProvider interface
com.sun.identity.saml.xmlsig.keyprovider.class=com.sun.identity.saml.xmlsig.JKSKeyProvider

# com.sun.identity.saml.xmlsig.signatureprovider.class specified the
# implementation class for com.sun.identity.saml.xmlsig.SignatureProvider
# interface
com.sun.identity.saml.xmlsig.signatureprovider.class=com.sun.identity.saml.xmlsig.AMSignatureProvider

com.iplanet.am.server.protocol=http
com.iplanet.am.server.host=openam.example.com
com.iplanet.am.server.port=8080
com.iplanet.am.services.deploymentDescriptor=/openam
com.iplanet.am.logstatus=ACTIVE

# Name of the webcontainer.
# Even though the servlet/JSP are web container independent,
# Access/Federation Manager uses servlet 2.3 API request.setCharacterEncoding()
# to decode incoming non English characters. These APIs will not work if
# Access/Federation Manager is deployed on Sun Java System Web Server 6.1.
# We use gx_charset mechanism to correctly decode incoming data in
# Sun Java System Web Server 6.1 and S1AS7.0. Possible values
# are BEA6.1, BEA 8.1, IBM5.1 or IAS7.0.
# If the web container is Sun Java System Webserver, the tag is not replaced.
com.sun.identity.webcontainer=WEB_CONTAINER

# Identify saml xml signature keystore file, keystore password file
# key password file
com.sun.identity.saml.xmlsig.keystore=%BASE_DIR%/security/kestores/keystore.jks
com.sun.identity.saml.xmlsig.storepass=%BASE_DIR%/storepass
com.sun.identity.saml.xmlsig.keypass=%BASE_DIR%/keypass
com.sun.identity.saml.xmlsig.certalias=test
    
```

```

# Type of KeyStore used for saml xml signature. Default is JKS.
#
# com.sun.identity.saml.xmlsig.storetype=JKS

# Specifies the implementation class for
# com.sun.identity.saml.xmlsig.PasswordDecoder interface.
com.sun.identity.saml.xmlsig.passwordDecoder=com.sun.identity.fedlet.FedletEncodeDecode

# The following key is used to specify the maximum content-length
# for an HttpRequest that will be accepted by the OpenSSO
# The default value is 16384 which is 16k
com.iplanet.services.comm.server.pllrequest.maxContentLength=16384

# The following keys are used to configure the Debug service.
# Possible values for the key 'level' are: off | error | warning | message.
# The key 'directory' specifies the output directory where the debug files
# will be created.
# Trailing spaces are significant.
# Windows: Use forward slashes "/" separate directories, not backslash "\".
# Windows: Spaces in the file name are allowed for Windows.
#
com.iplanet.services.debug.level=message
com.iplanet.services.debug.directory=%BASE_DIR%%SERVER_URI%/debug

# The following keys are used to configure the Stats service.
# Possible values for the key 'level' are: off | file | console
# Stats state 'file' will write to a file under the specified directory,
# and 'console' will write into webserver log files
# The key 'directory' specifies the output directory where the debug files
# will be created.
# Trailing spaces are significant.
# Windows: Use forward slashes "/" separate directories, not backslash "\".
# Windows: Spaces in the file name are allowed for Windows.
# Stats interval should be atleast 5 secs to avoid CPU saturation,
# the product would assume any thing less than 5 secs is 5 secs.
com.iplanet.am.stats.interval=60
com.iplanet.services.stats.state=file
com.iplanet.services.stats.directory=%BASE_DIR%/var/stats

# The key that will be used to encrypt and decrypt passwords.
am.encryption.pwd=@AM_ENC_PWD@

# SecureRandom Properties: The key
# "com.iplanet.security.SecureRandomFactoryImpl"
# specifies the factory class name for SecureRandomFactory
# Available impl classes are:
#   com.iplanet.am.util.JSSSecureRandomFactoryImpl (uses JSS)
#   com.iplanet.am.util.SecureRandomFactoryImpl (pure Java)
com.iplanet.security.SecureRandomFactoryImpl=com.iplanet.am.util.SecureRandomFactoryImpl

# SocketFactory properties: The key "com.iplanet.security.SSLSocketFactoryImpl"
# specifies the factory class name for LDAPSocketFactory
# Available classes are:
#   com.iplanet.services.ldap.JSSSocketFactory (uses JSS)
#   com.sun.identity.shared.ldap.factory.JSSESocketFactory (pure Java)
com.iplanet.security.SSLSocketFactoryImpl=com.sun.identity.shared.ldap.factory.JSSESocketFactory

# Encryption: The key "com.iplanet.security.encryptor" specifies
    
```

```
# the encrypting class implementation.
# Available classes are:
#   com.iplanet.services.util.JCEEncryption
#   com.iplanet.services.util.JSSEncryption
com.iplanet.security.encryptor=com.iplanet.services.util.JCEEncryption

# Determines if JSS will be added with highest priority to JCE
# Set this to "true" if other JCE providers should be used for
# digital signatures and encryptions.
com.sun.identity.jss.donotInstallAtHighestPriority=true

# Configuration File (serverconfig.xml) Location
com.iplanet.services.configpath=@BASE_DIR@
```

### *fedlet.cot*

The following example of `$HOME/.openig/SAML/fedlet.cot` defines a circle of trust between AM as the IDP, and IG as the SP:

```
cot-name=Circle of Trust
sun-fm-cot-status=Active
sun-fm-trusted-providers=openam, sp
sun-fm-saml2-readerservice-url=
sun-fm-saml2-writerservice-url=
```

### *idp.xml*

The following example of `$HOME/.openig/SAML/idp.xml` defines a SAML configuration file for the AM IDP, `idp`:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<EntityDescriptor entityID="openam" xmlns="urn:oasis:names:tc:SAML:2.0:metadata" xmlns:query="urn:oasis:names:tc:SAML:2.0:metadata:query"
  xmlns:xenc="http://www.w3.org/2009/xmlenc11#" xmlns:xmldsig="http://www.w3.org/2000/09/xmldsig#" xmlns:alg="urn:oasis:names:tc:SAML:metadata:alg-support"
  xmlns:x509="urn:oasis:names:tc:SAML:2.0:metadata:x509" />
  <IDPSSODescriptor protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
    <KeyDescriptor use="signing">
      <ds:KeyInfo>
        <ds:X509Data>
          <ds:X509Certificate>
            ...
          </ds:X509Certificate>
        </ds:X509Data>
      </ds:KeyInfo>
    </KeyDescriptor>
    <KeyDescriptor use="encryption">
      <ds:KeyInfo>
        <ds:X509Data>
          <ds:X509Certificate>
            ...
          </ds:X509Certificate>
        </ds:X509Data>
      </ds:KeyInfo>
      <EncryptionMethod Algorithm="http://www.w3.org/2009/xmlenc11#rsa-oaep">
        <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc11#sha256"/>
        <xenc11:MGF Algorithm="http://www.w3.org/2009/xmlenc11#mgf1sha256"/>
      </EncryptionMethod>
    </KeyDescriptor>
  </IDPSSODescriptor>
</EntityDescriptor>
```

```

        </EncryptionMethod>
        <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc">
            <xenc:KeySize>128</xenc:KeySize>
        </EncryptionMethod>
    </KeyDescriptor>

    <ArtifactResolutionService index="0" Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://
    openam.example.com:8088/openam/ArtifactResolver/metaAlias/idp"/>
        <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-
        Redirect" Location="http://openam.example.com:8088/openam/IDPSloRedirect/metaAlias/
        idp" ResponseLocation="http://openam.example.com:8088/openam/IDPSloRedirect/metaAlias/idp"/>
            <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="http://
            openam.example.com:8088/openam/IDPSloPOST/metaAlias/idp" ResponseLocation="http://openam.example.com:8088/
            openam/IDPSloPOST/metaAlias/idp"/>
                <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://
                openam.example.com:8088/openam/IDPSloSoap/metaAlias/idp"/>
                    <ManageNameIDService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-
                    Redirect" Location="http://openam.example.com:8088/openam/IDPMniRedirect/metaAlias/
                    idp" ResponseLocation="http://openam.example.com:8088/openam/IDPMniRedirect/metaAlias/idp"/>
                        <ManageNameIDService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="http://
                        openam.example.com:8088/openam/IDPMniPOST/metaAlias/idp" ResponseLocation="http://openam.example.com:8088/
                        openam/IDPMniPOST/metaAlias/idp"/>
                            <ManageNameIDService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://
                            openam.example.com:8088/openam/IDPMniSoap/metaAlias/idp"/>
                                <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:persistent</NameIDFormat>
                                <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:transient</NameIDFormat>
                                <NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-format:emailAddress</NameIDFormat>
                                <NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-format:unspecified</NameIDFormat>
                                <NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-format:WindowsDomainQualifiedName</NameIDFormat>
                                <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:kerberos</NameIDFormat>
                                <NameIDFormat>urn:oasis:names:tc:SAML:1.1:nameid-format:X509SubjectName</NameIDFormat>
                            <SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-
                            Redirect" Location="http://openam.example.com:8088/openam/SSORedirect/metaAlias/idp"/>
                                <SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="http://
                                openam.example.com:8088/openam/SSOPOST/metaAlias/idp"/>
                                    <SingleSignOnService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://
                                    openam.example.com:8088/openam/SSOSoap/metaAlias/idp"/>
                                        <NameIDMappingService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://
                                        openam.example.com:8088/openam/NIMSoap/metaAlias/idp"/>
                                            <AssertionIDRequestService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://
                                            openam.example.com:8088/openam/AIDReqSoap/IDPRole/metaAlias/idp"/>
                                                <AssertionIDRequestService Binding="urn:oasis:names:tc:SAML:2.0:bindings:URI" Location="http://
                                                openam.example.com:8088/openam/AIDReqUri/IDPRole/metaAlias/idp"/>
                                                    </IDPSSODescriptor>
                                                </EntityDescriptor>
    
```

## idp-extended.xml

The following example of `$HOME/.openig/SAML/idp-extended.xml` defines an AM-specific SAML descriptor file for the IDP:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<!--
    DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.

    Copyright (c) 2002-2010 Sun Microsystems Inc. All Rights Reserved

```

The contents of this file are subject to the terms of the Common Development and Distribution License (the License). You may not use this file except in compliance with the License.

You can obtain a copy of the License at <https://opensso.dev.java.net/public/CDDLv1.0.html> or [opensso/legal/CDDLv1.0.txt](https://opensso/legal/CDDLv1.0.txt) See the License for the specific language governing permission and limitations under the License.

When distributing Covered Code, include this CDDL Header Notice in each file and include the License file at [opensso/legal/CDDLv1.0.txt](https://opensso/legal/CDDLv1.0.txt). If applicable, add the following below the CDDL Header, with the fields enclosed by brackets [] replaced by your own identifying information:  
 "Portions Copyrighted [year] [name of copyright owner]"

Portions Copyrighted 2010-2017 ForgeRock AS.

```
-->
<EntityConfig entityID="openam" hosted="0" xmlns="urn:sun:fm:SAML:2.0:entityconfig">
  <IDPSSOConfig>
    <Attribute name="description">
      <Value/>
    </Attribute>
    <Attribute name="cotlist">
      <Value>Circle of Trust</Value>
    </Attribute>
  </IDPSSOConfig>
  <AttributeAuthorityConfig>
    <Attribute name="cotlist">
      <Value>Circle of Trust</Value>
    </Attribute>
  </AttributeAuthorityConfig>
  <XACMLPDPConfig>
    <Attribute name="wantXACMLAuthzDecisionQuerySigned">
      <Value></Value>
    </Attribute>
    <Attribute name="cotlist">
      <Value>Circle of Trust</Value>
    </Attribute>
  </XACMLPDPConfig>
</EntityConfig>
```

## sp.xml

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

```
<!--
DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.

Copyright (c) 2002-2010 Sun Microsystems Inc. All Rights Reserved

The contents of this file are subject to the terms
of the Common Development and Distribution License
(the License). You may not use this file except in
```

```

compliance with the License.

You can obtain a copy of the License at
https://opensso.dev.java.net/public/CDDLv1.0.html or
opensso/legal/CDDLv1.0.txt
See the License for the specific language governing
permission and limitations under the License.

When distributing Covered Code, include this CDDL
Header Notice in each file and include the License file
at opensso/legal/CDDLv1.0.txt.
If applicable, add the following below the CDDL Header,
with the fields enclosed by brackets [] replaced by
your own identifying information:
"Portions Copyrighted [year] [name of copyright owner]"

Portions Copyrighted 2010-2017 ForgeRock AS.
-->
<EntityDescriptor entityID="sp" xmlns="urn:oasis:names:tc:SAML:2.0:metadata">
  <SPSSODescriptor AuthnRequestsSigned="false" WantAssertionsSigned="false" protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
    <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Redirect" Location="http://sp.example.com:8080/saml/fedletSloRedirect" ResponseLocation="http://sp.example.com:8080/saml/fedletSloRedirect"/>
    <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="http://sp.example.com:8080/saml/fedletSloPOST" ResponseLocation="http://sp.example.com:8080/saml/fedletSloPOST"/>
    <SingleLogoutService Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP" Location="http://sp.example.com:8080/saml/fedletSloSoap"/>
    <NameIDFormat>urn:oasis:names:tc:SAML:2.0:nameid-format:transient</NameIDFormat>
    <AssertionConsumerService isDefault="true" index="0" Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST" Location="http://sp.example.com:8080/saml/fedletapplication"/>
    <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:2.0:metadata:ext:query" xsi:type="query:AttributeQueryDescriptorType" protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
    </RoleDescriptor>
    <XACMLAuthzDecisionQueryDescriptor WantAssertionsSigned="false" protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
    </XACMLAuthzDecisionQueryDescriptor>
  </EntityDescriptor>

```

## sp-extended.xml

The following example of `$HOME/.openid/SAML/sp-extended.xml` defines an AM-specific SAML descriptor file for the SP:

```

<!--
DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.

Copyright (c) 2002-2010 Sun Microsystems Inc. All Rights Reserved

The contents of this file are subject to the terms
of the Common Development and Distribution License
(the License). You may not use this file except in
compliance with the License.

```

You can obtain a copy of the License at  
<https://opensso.dev.java.net/public/CDDLv1.0.html> or  
[opensso/legal/CDDLv1.0.txt](https://opensso/legal/CDDLv1.0.txt)  
 See the License for the specific language governing  
 permission and limitations under the License.

When distributing Covered Code, include this CDDL  
 Header Notice in each file and include the License file  
 at [opensso/legal/CDDLv1.0.txt](https://opensso/legal/CDDLv1.0.txt).

If applicable, add the following below the CDDL Header,  
 with the fields enclosed by brackets [] replaced by  
 your own identifying information:

"Portions Copyrighted [year] [name of copyright owner]"

Portions Copyrighted 2010-2017 ForgeRock AS.

-->

```
<EntityConfig xmlns="urn:sun:fm:SAML:2.0:entityconfig" xmlns:fm="urn:sun:fm:SAML:2.0:entityconfig" hosted="1" entity
  <SPSSOConfig metaAlias="/sp">
    <Attribute name="description">
      <Value></Value>
    </Attribute>
    <Attribute name="signingCertAlias">
      <Value></Value>
    </Attribute>
    <Attribute name="encryptionCertAlias">
      <Value></Value>
    </Attribute>
    <Attribute name="basicAuthOn">
      <Value>>false</Value>
    </Attribute>
    <Attribute name="basicAuthUser">
      <Value></Value>
    </Attribute>
    <Attribute name="basicAuthPassword">
      <Value></Value>
    </Attribute>
    <Attribute name="autofedEnabled">
      <Value>>false</Value>
    </Attribute>
    <Attribute name="autofedAttribute">
      <Value></Value>
    </Attribute>
    <Attribute name="transientUser">
      <Value>anonymous</Value>
    </Attribute>
    <Attribute name="spAdapter">
      <Value></Value>
    </Attribute>
    <Attribute name="spAdapterEnv">
      <Value></Value>
    </Attribute>
    <Attribute name="fedletAdapter">
      <Value></Value>
    </Attribute>
    <Attribute name="fedletAdapterEnv">
      <Value></Value>
    </Attribute>
    <Attribute name="spAccountMapper">
```

```

        <Value>com.sun.identity.saml2.plugins.DefaultLibrarySPAAccountMapper</Value>
    </Attribute>
    <Attribute name="spAttributeMapper">
        <Value>com.sun.identity.saml2.plugins.DefaultSPAttributeMapper</Value>
    </Attribute>
    <Attribute name="spAuthncontextMapper">
        <Value>com.sun.identity.saml2.plugins.DefaultSPAuthnContextMapper</Value>
    </Attribute>
    <Attribute name="spAuthncontextClassrefMapping">
        <Value>urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport|0|default</Value>
    </Attribute>
    <Attribute name="spAuthncontextComparisonType">
        <Value>exact</Value>
    </Attribute>
    <Attribute name="attributeMap">
        <Value>*=</Value>
    </Attribute>
    <Attribute name="saml2AuthModuleName">
        <Value></Value>
    </Attribute>
    <Attribute name="localAuthURL">
        <Value></Value>
    </Attribute>
    <Attribute name="intermediateUrl">
        <Value></Value>
    </Attribute>
    <Attribute name="defaultRelayState">
        <Value></Value>
    </Attribute>
    <Attribute name="appLogoutUrl">
        <Value>http://sp.example.com:8080/saml/logout</Value>
    </Attribute>
    <Attribute name="assertionTimeSkew">
        <Value>300</Value>
    </Attribute>
    <Attribute name="wantAttributeEncrypted">
        <Value></Value>
    </Attribute>
    <Attribute name="wantAssertionEncrypted">
        <Value></Value>
    </Attribute>
    <Attribute name="wantNameIDEncrypted">
        <Value></Value>
    </Attribute>
    <Attribute name="wantPOSTResponseSigned">
        <Value></Value>
    </Attribute>
    <Attribute name="wantArtifactResponseSigned">
        <Value></Value>
    </Attribute>
    <Attribute name="wantLogoutRequestSigned">
        <Value></Value>
    </Attribute>
    <Attribute name="wantLogoutResponseSigned">
        <Value></Value>
    </Attribute>
    <Attribute name="wantMNIRequestSigned">
        <Value></Value>
    </Attribute>

```



```

<Attribute name="wantMNIResponseSigned">
  <Value></Value>
</Attribute>
<Attribute name="cotlist">
  <Value>Circle of Trust</Value></Attribute>
<Attribute name="saeAppSecretList">
</Attribute>
<Attribute name="saeSPUrl">
  <Value></Value>
</Attribute>
<Attribute name="saeSPLogoutUrl">
</Attribute>
<Attribute name="ECPRequestIDPLListFinderImpl">
  <Value>com.sun.identity.saml2.plugins.ECPIPDFinder</Value>
</Attribute>
<Attribute name="ECPRequestIDPLList">
  <Value></Value>
</Attribute>
<Attribute name="enableIDPProxy">
  <Value>>false</Value>
</Attribute>
<Attribute name="idpProxyList">
  <Value></Value>
</Attribute>
<Attribute name="idpProxyCount">
  <Value>0</Value>
</Attribute>
<Attribute name="useIntroductionForIDPProxy">
  <Value>>false</Value>
</Attribute>
</SPSSOConfig>
<AttributeQueryConfig metaAlias="/attrQuery">
  <Attribute name="signingCertAlias">
    <Value></Value>
  </Attribute>
  <Attribute name="encryptionCertAlias">
    <Value></Value>
  </Attribute>
  <Attribute name="wantNameIDEncrypted">
    <Value></Value>
  </Attribute>
  <Attribute name="cotlist">
    <Value>Circle of Trust</Value>
  </Attribute>
</AttributeQueryConfig>
<XACMLAuthzDecisionQueryConfig metaAlias="/pep">
  <Attribute name="signingCertAlias">
    <Value></Value>
  </Attribute>
  <Attribute name="encryptionCertAlias">
    <Value></Value>
  </Attribute>
  <Attribute name="basicAuthOn">
    <Value>>false</Value>
  </Attribute>
  <Attribute name="basicAuthUser">
    <Value></Value>
  </Attribute>
  <Attribute name="basicAuthPassword">

```

```
<Value></Value>
</Attribute>
<Attribute name="wantXACMLAuthzDecisionResponseSigned">
  <Value></Value>
</Attribute>
<Attribute name="wantAssertionEncrypted">
  <Value></Value>
</Attribute>
<Attribute name="cotlist">
  <Value>Circle of Trust</Value>
</Attribute>
</XACMLAuthzDecisionQueryConfig>
</EntityConfig>
```

## Chapter 10

# Acting As an OAuth 2.0 Resource Server

The following sections describe how IG acts as an OAuth 2.0 Resource Server, to resolve and validate access\_tokens, and inject them into the context:

- "About IG As an OAuth 2.0 Resource Server"
- "Validating Access\_Tokens Through the Introspection Endpoint"
- "Validating Stateless Access\_Tokens With the StatelessAccessTokenResolver"
- "Validating Certificate-Bound Access Tokens"
- "Using the OAuth 2.0 Context to Log in to the Sample Application"
- "Caching Access\_Tokens"

For information about allowing third-party applications to access users' resources without having users' credentials, see [OAuth 2.0 Authorization Framework](#).

For information about the context, see "OAuth2Context" in the *Configuration Reference*. For examples that use fields in OAuth2Context to throttle access to the sample application, see "Configuring Mapped Throttling" and "Configuring Scriptable Throttling".

## About IG As an OAuth 2.0 Resource Server

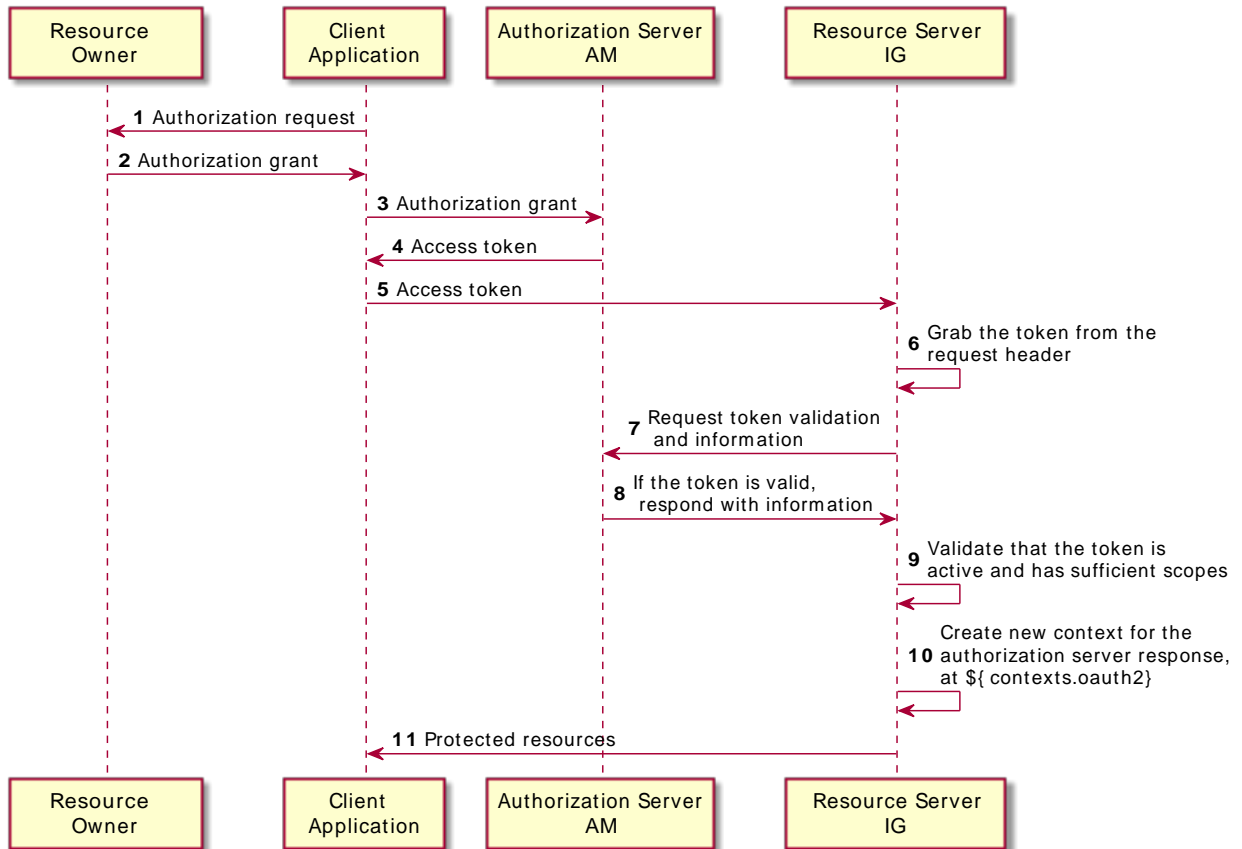
OAuth 2.0 includes the following entities:

- *Resource owner*: A user who owns protected resources on a resource server. For example, a resource owner can store photos in a web service.
- *Resource server*: A service that gives authorized client applications access to the resource owner's protected resources. In OAuth 2.0, an authorization server grants authorization to a client application, based on the resource owner's consent. For example, a resource server can be a web service that holds a user's photos.
- *Client*: An application that requests access to the resource owner's protected resources, on behalf of the resource owner. For example, a client can be a photo printing service requesting access to a resource owner's photos stored on a web service, after the resource owner gives the client consent to download the photos.

- *Authorization server*: A service responsible for authenticating resource owners, and obtaining their consent to allow client applications to access their resources. For example, AM can act as the OAuth 2.0 authorization server to authenticate resource owners and obtain their consent. Other services, such as Google and Facebook can provide OAuth 2.0 authorization services.

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*, representing the resource owner's consent. For information about the different OAuth 2.0 grant mechanisms supported by AM, see OAuth 2.0 Grant Flows in AM's *OAuth 2.0 Guide*.
- The application authenticates to the authorization server and requests an *access\_token*. The authorization server returns an *access\_token* to the application.

An OAuth 2.0 access\_token is an opaque string issued by the authorization server. When the client interacts with the resource server, the client presents the access\_token in the `Authorization` header. For example:

```
Authorization: Bearer 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 access\_token represents the authorization to access protected resources. Because an access\_token is a bearer token, anyone who has the access\_token can use it to get the resources. Access\_tokens must therefore be protected, so that requests involving them go over HTTPS.

In OAuth 2.0, the token scopes are strings that identify the scope of access authorized to the client, but can also be used for other purposes.

- The application supplies the access\_token to the resource server, which then resolves and validates the access\_token by using an access\_token resolver, as described in "*Access Token Resolvers*" in the *Configuration Reference*.

If the access\_token is valid, the resource server permits the client access to the requested resource.


## Validating Access\_Tokens Through the Introspection Endpoint



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

For more information about configuring AM as an OAuth 2.0 authorization service, see AM's *OAuth 2.0 Guide*.

### *Validate Access\_Tokens Through the Introspection Endpoint*

Before you start, prepare AM, IG, and the sample application as described in "Example Installation for This Guide".

1. Set up AM:
  - a. Select Applications > Agents > Identity Gateway, add an agent with the following values:
    - Agent ID: `ig_agent`
    - Password: `password`
    - Token Introspection: `Realm Only`
  - b. Select  Identities, and add a user with the following values:
    - ID/username: `george`

- First name: `george`
  - Last name: `costanza`
  - Password: `C0stanza`
  - Email Address: `george@example.com`
  - Employee number: `123`
- c. Create an OAuth 2.0 Authorization Server:
- i. Select  Services > Add a Service > OAuth2 Provider.
  - ii. Add a service with the default values.
- d. Create an OAuth 2.0 Client to request OAuth 2.0 access\_tokens:
- i. Select  Applications > OAuth 2.0 > Clients, and add a client with the following values:
    - Client ID: `client-application`
    - Client secret: `password`
    - Scope(s): `mail, employeenumber`
  - ii. (From AM 6.5) On the Advanced tab, select the following value:
    - Grant Types: `Resource Owner Password Credentials`

## 2. Set up IG

- a. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

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

- b. Add the following route to IG:

*Linux*

```
$HOME/.openig/config/routes/rs-introspect.json
```

*Windows*

```
%appdata%\OpenIG\config\routes\rs-introspect.json
```

```
{
  "name": "rs-introspect",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/rs-introspect$')}",
  "heap": [
```

```

    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://openam.example.com:8088/openam/",
        "version": "7"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "TokenIntrospectionAccessTokenResolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {
                    "filters": [
                      {
                        "type": "HttpBasicAuthenticationClientFilter",
                        "config": {
                          "username": "ig_agent",
                          "passwordSecretId": "agent.secret.id",
                          "secretsProvider": "SystemAndEnvSecretStore-1"
                        }
                      }
                    ]
                  }
                }
              }
            },
            "handler": "ForgeRockClientHandler"
          }
        }
      ]
    }
  }
}

```

```

        "type": "StaticResponseHandler",
        "config": {
            "status": 200,
            "headers": {
                "Content-Type": [ "text/html" ]
            },
            "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></body></html>"
        }
    }
}
}
}
}

```

For information about how to set up the IG route in Studio, see "Resource Server Using the Introspection Endpoint in Structured Editor" in the *Studio User Guide*.

Notice the following features of the route:

- The route matches requests to `/rs-introspect`.
- The `OAuth2ResourceServerFilter` expects an OAuth 2.0 `access_token` in the header of the incoming authorization request, with the scopes `mail` and `employeenumber`.

The `accessTokenResolver` uses the AM server declared in the heap. The introspection endpoint to validate the `access_token` is extrapolated from the URL of the AM server.

For convenience in this test, `requireHttps` is false. In production environments, set it to true.

- After the filter validates the `access_token`, it creates a new context from the authorization server response. The context is named `oauth2`, and can be reached at `contexts.oauth2` or `contexts['oauth2']`.

The context contains information about the `access_token`, which can be reached at `contexts.oauth2.accessToken.info`. Filters and handlers further down the chain can access the token info through the context.

If there is no `access_token` in the request, or token validation does not complete successfully, the filter returns an HTTP error status to the user-agent, and IG does not continue processing the request. This is done as specified in the RFC, OAuth 2.0 Bearer Token Usage.

- The `HttpBasicAuthenticationClientFilter` adds the credentials to the outgoing token introspection request.
- The `StaticResponseHandler` returns the content of the `access_token` from the context `${contexts.oauth2.accessToken.info}`.

### 3. Test the setup:



- a. In a terminal window, use a **curl** command similar to the following to retrieve an `access_token`:

```
$ mytoken=$(curl -s \  
--user "client-application:password" \  
--data "grant_type=password&username=george&password=C0stanza&scope=mail%20employeeenumber" \  
http://openam.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
```

- b. Validate the `access_token` returned in the previous step:

```
$ curl -v http://openig.example.com:8080/rs-introspect --header "Authorization: Bearer ${mytoken}"  
  
{  
  active = true,  
  scope = employeeenumber mail,  
  client_id = client - application,  
  user_id = george,  
  token_type = Bearer,  
  exp = 158...907,  
  sub = george,  
  iss = http://openam.example.com:8088/openam/oauth2, ...  
  ...  
}
```

## Validating Stateless Access\_Tokens With the StatelessAccessTokenResolver

The `StatelessAccessTokenResolver` confirms that `stateless access_token`s provided by AM are well-formed, have a valid issuer, have the expected `access_token` name, and have a valid signature.

After the `StatelessAccessTokenResolver` resolves an `access_token`, the `OAuth2ResourceServerFilter` checks that the token is within the expiry time, and that it provides the required scopes. For more information, see "`StatelessAccessTokenResolver`" in the *Configuration Reference*. This feature is supported with OpenAM 13.5, and AM 5 and later versions.




The following sections provide examples of how to validate signed and encrypted `access_token`s:

- "Validating Signed Access\_Tokens With the StatelessAccessTokenResolver and JwkSetSecretStore"
- "Validating Signed Access\_Tokens With the StatelessAccessTokenResolver and KeyStoreSecretStore"
- "Validating Encrypted Access\_Tokens With the StatelessAccessTokenResolver and KeyStoreSecretStore"

## Validating Signed Access\_Tokens With the StatelessAccessTokenResolver and JwkSetSecretStore

This section provides examples of how to validate signed access\_tokens with the StatelessAccessTokenResolver, using a JwkSetSecretStore. For more information about JwkSetSecretStore, see "JwkSetSecretStore" in the *Configuration Reference*.

### Set Up AM

1. Configure an OAuth 2.0 Authorization Provider:
  - a. Select  Services, and add an OAuth 2.0 Provider.
  - b. Accept all of the default values, and select Create. The service is added to the  Services list.
  - c. On the Core tab, select the following option:
    - Use Client-Based Access & Refresh Tokens: **on**
  - d. On the Advanced tab, select the following options:
    - Client Registration Scope Whitelist: **myscope**
    - OAuth2 Token Signing Algorithm: **RS256**
    - Encrypt Client-Based Tokens: Deselected
2. Create an OAuth2 Client to request OAuth 2.0 access\_tokens:
  - a. Select  Applications > OAuth 2.0 > Clients, and add a client with the following values:
    - Client ID: **client-application**
    - Client secret: **password**
    - Scope(s): **myscope**
  - b. (From AM 6.5) On the Advanced tab, select the following values:
    - Grant Types: **Resource Owner Password Credentials**
    - Response Types: **code token**
  - c. On the Signing and Encryption tab, include the following setting:
    - ID Token Signing Algorithm: **RS256**

### Set Up IG

1. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/rs-stateless-signed.json
```

Windows

```
%appdata%\OpenIG\config\routes\rs-stateless-signed.json
```

```
{
  "name": "rs-stateless-signed",
  "condition": "${matches(request.uri.path, '/rs-stateless-signed')}",
  "heap": [
    {
      "name": "SecretsProvider-1",
      "type": "SecretsProvider",
      "config": {
        "stores": [
          {
            "type": "JwkSetSecretStore",
            "config": {
              "jwkUrl": "http://openam.example.com:8088/openam/oauth2/connect/jwk_uri"
            }
          }
        ]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": ["myscope"],
            "requireHttps": false,
            "accessTokenResolver": {
              "type": "StatelessAccessTokenResolver",
              "config": {
                "secretsProvider": "SecretsProvider-1",
                "issuer": "http://openam.example.com:8088/openam/oauth2",
                "verificationSecretId": "any.value.in.regex.format"
              }
            }
          }
        }
      ]
    }
  },
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/html" ]
      },
      "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></body></html>"
    }
  }
}
```

```
}  
  }  
} }  
}
```

2. Notice the following features of the route:

- The route matches requests to `/rs-stateless-signed`.
- A `SecretsProvider` in the heap declares a `JwkSetSecretStore` to manage secrets for signed `access_tokens`.
- The `JwkSetSecretStore` specifies the URL to a JWK set on AM, that contains the signing keys.
- The `OAuth2ResourceServerFilter` expects an OAuth 2.0 `access_token` in the header of the incoming authorization request, with the scope `myscope`.
- The `StatelessAccessTokenResolver` uses the `SecretsProvider` to verify the signature of the provided `access_token`.
- After the `OAuth2ResourceServerFilter` validates the `access_token`, it creates the `OAuth2Context` context. For more information, see "`OAuth2Context`" in the *Configuration Reference*.
- If there is no `access_token` in a request, or token validation does not complete successfully, the filter returns an HTTP error status to the user-agent, and IG does not continue processing the request. This is done as specified in the RFC OAuth 2.0 Bearer Token Usage.
- The `StaticResponseHandler` returns the content of the `access_token` from the context.

### Test the Setup For a Signed Access\_Token

1. Get an `access_token` for the demo user, using the scope `myscope`:

```
$ mytoken=$(curl -s \  
--user "client-application:password" \  
--data "grant_type=password&username=demo&password=Ch4ng3!t&scope=myscope" \  
http://openam.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
```

2. Display the token:

```
$ echo ${mytoken}
```

Note that the token is structured as a signed token.

3. Access the route by providing the token returned in the previous step:

```
$ curl -v http://openig.example.com:8080/rs-stateless-signed --header "Authorization: Bearer  
${mytoken}"  
...  
Decoded access_token: {  
  sub=demo,  
  cts=0AUTH2_STATELESS_GRANT,  
  ...  
}
```

## Validating Signed Access\_Tokens With the StatelessAccessTokenResolver and KeyStoreSecretStore

This section provides examples of how to validate signed access\_tokens with the StatelessAccessTokenResolver, using a KeyStoreSecretStore. For more information about KeyStoreSecretStore, see "KeyStoreSecretStore" in the *Configuration Reference*.

### Set Up Keys for Signing

1. Locate the following directories for keys, keystores, and certificates, and in a terminal create variables for them:

- Directory where the keystore is created: `keystore_directory`
- AM keystore directory: `am_keystore_directory`
- IG keystore directory: `ig_keystore_directory`

2. Set up the keystore for signing keys:

a. Generate a private key called `signature-key`, and a corresponding public certificate called `x509certificate.pem`:

```
$ openssl req -x509 \  
-newkey rsa:2048 \  
-nodes \  
-subj "/CN=openig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \  
-keyout $keystore_directory/signature-key.key \  
-out $keystore_directory/x509certificate.pem \  
-days 365  
...  
writing new private key to '$keystore_directory/signature-key.key'
```

b. Convert the private key and certificate files into a PKCS12 file, called `signature-key`, and store them in a keystore named `keystore.p12`:

```
$ openssl pkcs12 \  
-export \  
-in $keystore_directory/x509certificate.pem \  
-inkey $keystore_directory/signature-key.key \  
-out $keystore_directory/keystore.p12 \  
-passout pass:password \  
-name signature-key
```

- c. List the keys in `keystore.p12`:

```
$ keytool -list \
-v \
-keystore "$keystore_directory/keystore.p12" \
-storepass "password" \
-storetype PKCS12
...
Your keystore contains 1 entry
Alias name: signature-key
```

3. Set up keys for AM:

- a. Copy the signing key `keystore.p12` to AM:

```
$ cp $keystore_directory/keystore.p12 $am_keystore_directory/AM_keystore.p12
```

- b. List the keys in the AM keystore:

```
$ keytool -list \
-v \
-keystore "$am_keystore_directory/AM_keystore.p12" \
-storepass "password" \
-storetype PKCS12
...
Your keystore contains 1 entry
Alias name: signature-key
```

- c. Add a file called `keystore.pass`, with the content `password`:

```
$ cd $am_keystore_directory
$ echo -n password > keystore.pass
```

The filename corresponds to the secret ID of the store password and entry password for the `KeyStoreSecretStore`.

- d. Restart AM.

4. Set up keys for IG:

- a. Import the public certificate to the IG keystore, with the alias `verification-key`:

```
$ keytool -import \
-trustcacerts \
-rfc \
-alias verification-key \
-file "$keystore_directory/x509certificate.pem" \
-keystore "$ig_keystore_directory/IG_keystore.p12" \
-storetype PKCS12 \
-storepass "password"
...
Trust this certificate? [no]: yes
Certificate was added to keystore
```

- b. List the keys in the IG keystore:

```
$ keytool -list \  
-v \  
-keystore "$ig_keystore_directory/IG_keystore.p12" \  
-storepass "password" \  
-storetype PKCS12  
  
...  
Your keystore contains 1 entry  
Alias name: verification-key
```

- c. In the IG configuration, set an environment variable for the keystore password:


```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcnQ='
```

- d. Restart IG.

### *Validate Signed Access\_Tokens With the StatelessAccessTokenResolver and KeyStoreSecretStore*

1. Set up AM:

- a. Create a KeyStoreSecretStore to manage the new AM keystore:

- i. In AM, select  Secret Stores, and then add a secret store with the following values:


- Secret Store ID: `keystoresecretstore`
- Store Type: `Keystore`
- File: `am_keystore_directory/AM_keystore.p12`
- Keystore type: `PKCS12`
- Store password secret ID: `keystore.pass`
- Entry password secret ID: `keystore.pass`




- ii. Select the Mappings tab, and add a mapping with the following values:

- Secret ID: `am.services.oauth2.stateless.signing.RSA`
- Aliases: `signature-key`

The mapping sets `signature-key` as the active alias to use for signature generation.

- b. Create a FileSystemSecretStore to manage secrets for the KeyStoreSecretStore:

- select  Secret Stores, and then create a secret store with the following configuration:

- Secret Store ID: `filesystemsecretstore`
  - Store Type: `File System Secret Volumes`
  - Directory: `am_keystore_directory/secrets`
  - File format: `Plain text`
- c. Configure an OAuth 2.0 Authorization Provider:
- i. Select  Services, and add an OAuth 2.0 Provider.
  - ii. Accept all of the default values, and select Create. The service is added to the  Services list.
  - iii. On the Core tab, select the following option:
    - Use Client-Based Access & Refresh Tokens: `on`
  - iv. On the Advanced tab, select the following options:
    - Client Registration Scope Whitelist: `myscope`
    - OAuth2 Token Signing Algorithm: `RS256`
    - Encrypt Client-Based Tokens: Deselected
- d. Create an OAuth2 Client to request OAuth 2.0 access\_tokens:
- i. Select  Applications > OAuth 2.0 > Clients, and add a client with the following values:
    - Client ID: `client-application`
    - Client secret: `password`
    - Scope(s): `myscope`
  - ii. (From AM 6.5) On the Advanced tab, select the following values:
    - Grant Types: `Resource Owner Password Credentials`
    - Response Types: `code token`
  - iii. On the Signing and Encryption tab, include the following setting:
    - ID Token Signing Algorithm: `RS256`
2. Set up IG:
- Add the following route to IG, and replace `ig_keystore_directory`:



Linux

```
$HOME/.openig/config/routes/rs-stateless-signed-ksss.json
```

Windows

```
%appdata%\OpenIG\config\routes\rs-stateless-signed-ksss.json
```

```
{
  "name": "rs-stateless-signed-ksss",
  "condition" : "${matches(request.uri.path, '/rs-stateless-signed-ksss')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "KeyStoreSecretStore-1",
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "<ig_keystore_directory>/IG_keystore.p12",
        "storeType": "PKCS12",
        "storePassword": "keystore.secret.id",
        "keyEntryPassword": "keystore.secret.id",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "mappings": [
          {
            "secretId": "stateless.access.token.verification.key",
            "aliases": [ "verification-key" ]
          }
        ]
      }
    }
  ],
  "handler" : {
    "type" : "Chain",
    "capture" : "all",
    "config" : {
      "filters" : [ {
        "name" : "OAuth2ResourceServerFilter-1",
        "type" : "OAuth2ResourceServerFilter",
        "config" : {
          "scopes" : [ "myscope" ],
          "requireHttps" : false,
          "accessTokenResolver": {
            "type": "StatelessAccessTokenResolver",
            "config": {
              "secretsProvider": "KeyStoreSecretStore-1",
              "issuer": "http://openam.example.com:8088/openam/oauth2",
              "verificationSecretId": "stateless.access.token.verification.key"
            }
          }
        }
      }
    ]
  },
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,

```

```
        "headers": {
            "Content-Type": [ "text/html" ]
        },
        "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></body></html>"
    }
}
```

Notice the following features of the route:

- The route matches requests to `/rs-stateless-signed-ksss`.
- The keystore password is provided by the `SystemAndEnvSecretStore` in the heap.
- The `OAuth2ResourceServerFilter` expects an OAuth 2.0 `access_token` in the header of the incoming authorization request, with the scope `myscope`.
- The `accessTokenResolver` uses a `StatelessAccessTokenResolver` to resolve and verify the authenticity of the `access_token`. The secret is provided by the `KeyStoreSecretStore` in the heap.
- After the `OAuth2ResourceServerFilter` validates the `access_token`, it creates the `OAuth2Context` context. For more information, see "OAuth2Context" in the *Configuration Reference*.
- If there is no `access_token` in a request, or if the token validation does not complete successfully, the filter returns an HTTP error status to the user-agent, and IG stops processing the request, as specified in the RFC, OAuth 2.0 Bearer Token Usage.
- The `StaticResponseHandler` returns the content of the `access_token` from the context.

### 3. Test the setup for a signed access\_token:

- a. Get an `access_token` for the demo user, using the scope `myscope`:

```
$ mytoken=$(curl -s \
--user "client-application:password" \
--data "grant_type=password&username=demo&password=Ch4ng31t&scope=myscope" \
http://openam.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
```

- b. Display the token:

```
$ echo ${mytoken}
```

- c. Access the route by providing the token returned in the previous step:

```
$ curl -v http://openig.example.com:8080/rs-stateless-signed-ksss --header "Authorization: Bearer
${mytoken}"
...
Decoded access_token: {
sub=demo,
cts=0AUTH2_STATELESS_GRANT,
...
}
```

## Validating Encrypted Access\_Tokens With the StatelessAccessTokenResolver and KeyStoreSecretStore

### Set Up Keys for Encryption

1. Locate the following directories for keys, keystores, and certificates, and in a terminal create variables for them:

- Directory where the keystore is created: `keystore_directory`
- AM keystore directory: `am_keystore_directory`
- IG keystore directory: `ig_keystore_directory`

2. Set up keys for AM:

a. Generate the encryption key:

```
$ keytool -genseckey \
-alias encryption-key \
-dname "CN=openig.example.com, OU=example, O=com, L=fr, ST=fr, C=fr" \
-keystore "$am_keystore_directory/AM_keystore.p12" \
-storetype PKCS12 \
-storepass "password" \
-keyalg AES \
-keysize 256
```

b. List the keys in the AM keystore:

```
$ keytool -list \
-v \
-keystore "$am_keystore_directory/AM_keystore.p12" \
-storepass "password" \
-storetype PKCS12
...
Your keystore contains 1 entry
Alias name: encryption-key
```

c. Add a file called `keystore.pass`, with the content `password`:

```
$ cd $am_keystore_directory
$ echo -n password > keystore.pass
```

The filename corresponds to the secret ID of the store password and entry password for the KeyStoreSecretStore.

- d. Restart AM.
3. Set up keys for IG:
    - a. Import `encryption-key` into the IG keystore, with the alias `decryption-key`:

```
$ keytool -importkeystore \
-srcalias encryption-key \
-srckeystore "$am_keystore_directory/AM_keystore.p12" \
-srcstoretype PKCS12 \
-srcstorepass "password" \
-destkeystore "$ig_keystore_directory/IG_keystore.p12" \
-deststoretype PKCS12 \
-destalias decryption-key \
-deststorepass "password" \
-destkeypass "password"
```

- b. List the keys in the IG keystore:

```
$ keytool -list \
-v \
-keystore "$ig_keystore_directory/IG_keystore.p12" \
-storepass "password" \
-storetype PKCS12


...
Your keystore contains 1 entry
Alias name: decryption-key
```

- c. In the IG configuration, set an environment variable for the keystore password:

```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcnQ='
```

- d. Restart IG.

### *Validate Encrypted Access\_Tokens With the StatelessAccessTokenResolver and KeyStoreSecretStore*

1. Set up AM:
  - a. Set up AM as described in "Validate Signed Access\_Tokens With the StatelessAccessTokenResolver and KeyStoreSecretStore".
  - b. Add a mapping for the encryption keystore:
    - i. select  Secret Stores > `keystoresecretstore`.
    - ii. Select the Mappings tab, and add a mapping with the following values:

- Secret ID: `am.services.oauth2.stateless.token.encryption`
  - Alias: `encryption-key`
- c. Enable token encryption on the OAuth 2.0 Authorization Provider:
- Select  Services > OAuth2 Provider.
  - On the Advanced tab, select Encrypt Client-Based Tokens.
2. Set up IG:
- a. Add the following route to IG, and replace `ig_keystore_directory`:

Linux

```
$HOME/.openig/config/routes/rs-stateless-encrypted.json
```

Windows

```
%appdata%\OpenIG\config\routes\rs-stateless-encrypted.json
```

```
{
  "name": "rs-stateless-encrypted",
  "condition": "${matches(request.uri.path, '/rs-stateless-encrypted')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "KeyStoreSecretStore-1",
      "type": "KeyStoreSecretStore",
      "config": {
        "file": "<ig_keystore_directory>/IG_keystore.p12",
        "storeType": "PKCS12",
        "storePassword": "keystore.secret.id",
        "keyEntryPassword": "keystore.secret.id",
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "mappings": [
          {
            "secretId": "stateless.access.token.decryption.key",
            "aliases": [ "decryption-key" ]
          }
        ]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [ {
        "name": "OAuth2ResourceServerFilter-1",
        "type": "OAuth2ResourceServerFilter",
        "config": {
          "scopes": [ "myscope" ],
```

```

    "requireHttps": false,
    "accessTokenResolver": {
      "type": "StatelessAccessTokenResolver",
      "config": {
        "secretsProvider": "KeyStoreSecretStore-1",
        "issuer": "http://openam.example.com:8088/openam/oauth2",
        "decryptionSecretId": "stateless.access.token.decryption.key"
      }
    }
  },
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/html" ]
      },
      "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></body></html>"
    }
  }
}
}
}

```

b. Notice the following features of the route compared to `rs-stateless-signed.json`, used in: "Validating Signed Access\_Tokens With the StatelessAccessTokenResolver and KeyStoreSecretStore":

- The route matches requests to `/rs-stateless-encrypted`.
- The `OAuth2ResourceServerFilter` and `KeyStoreSecretStore` refer to the configuration for a decryption key instead of a verification key.

### Test the Setup For an Encrypted Access\_Token

1. Get an access\_token for the demo user, using the scope `myscope`:

```

$ mytoken=$(curl -s \
--user "client-application:password" \
--data "grant_type=password&username=demo&password=Ch4ng3!t&scope=myscope" \
http://openam.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")

```

2. Display the token:

```

$ echo ${mytoken}

```

Note that the token is structured as an encrypted token.

3. Access the route by providing the token returned in the previous step:

```
$ curl -v http://openig.example.com:8080/rs-stateless-encrypted --header "Authorization: Bearer  
${mytoken}"  
...  
Decoded access_token: {  
  sub=demo,  
  cts=0AUTH2_STATELESS_GRANT,  
  ...
```

## Validating Certificate-Bound Access Tokens

Clients can authenticate to AM through mutual TLS (mTLS) and X.509 certificates. Certificates must be self-signed or use public key infrastructure (PKI), as described in version 12 of the draft OAuth 2.0 Mutual TLS Client Authentication and Certificate Bound Access Tokens.

When a client requests an `access_token` from AM through mTLS, AM can use a *confirmation key* to bind the `access_token` to the presented client certificate. The confirmation key is the certificate *thumbprint*, computed as `base64url-encode(sha256(der(certificat)))`. The `access_token` is then *certificate-bound*. For more information, see [Authenticating Clients Using Mutual TLS in AM's OAuth 2.0 Guide](#).

When the client connects to IG by using that certificate, IG can verify that the confirmation key corresponds to the presented certificate. This proof-of-possession interaction ensures that only the client in possession of the key corresponding to the certificate can use the `access_token` to access protected resources.

The following sections provide examples of how to validate certificate-bound `access_tokens`:

- ["mTLS Using Standard TLS Client Certificate Authentication"](#)
- ["mTLS Using Trusted Headers"](#)

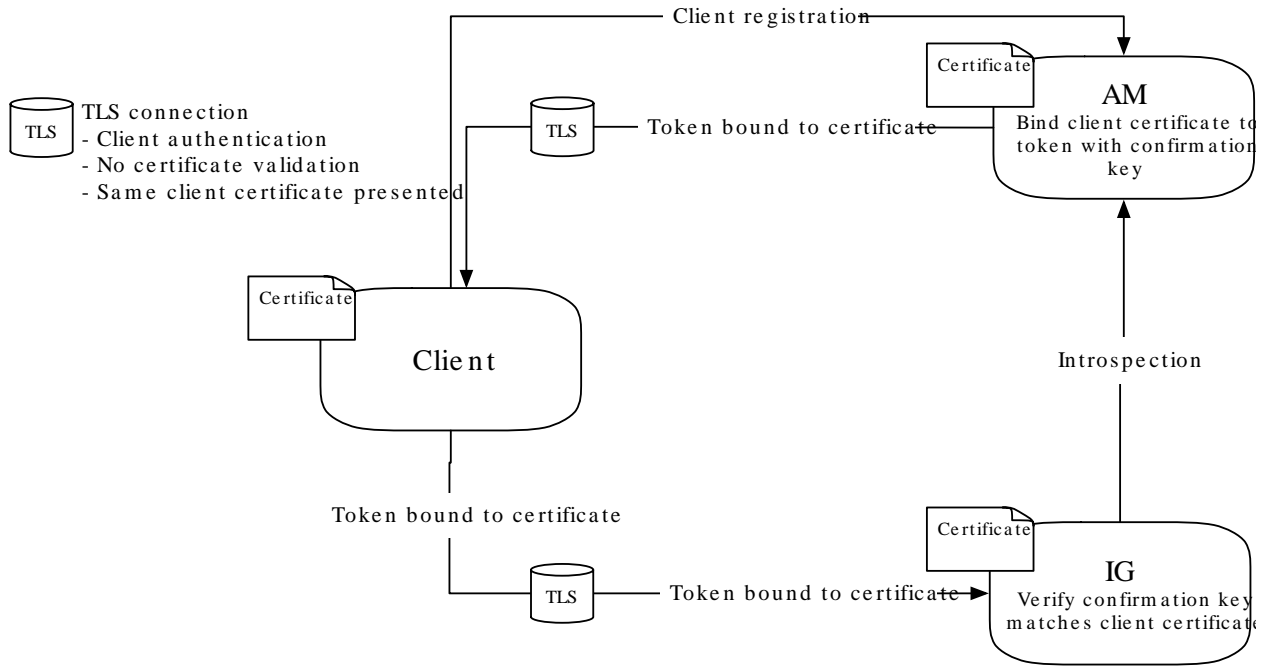
### mTLS Using Standard TLS Client Certificate Authentication

IG can validate the thumbprint of certificate-bound `access_tokens` by reading the client certificate from the TLS connection. When the web container that is running IG performs a successful TLS connection handshake, the connected client is trusted.

For this example, the client must be connected directly to IG through a TLS connection, for which IG is the TLS termination point. If TLS is terminated at a reverse proxy or load balancer before IG, use the example in ["mTLS Using Trusted Headers"](#).

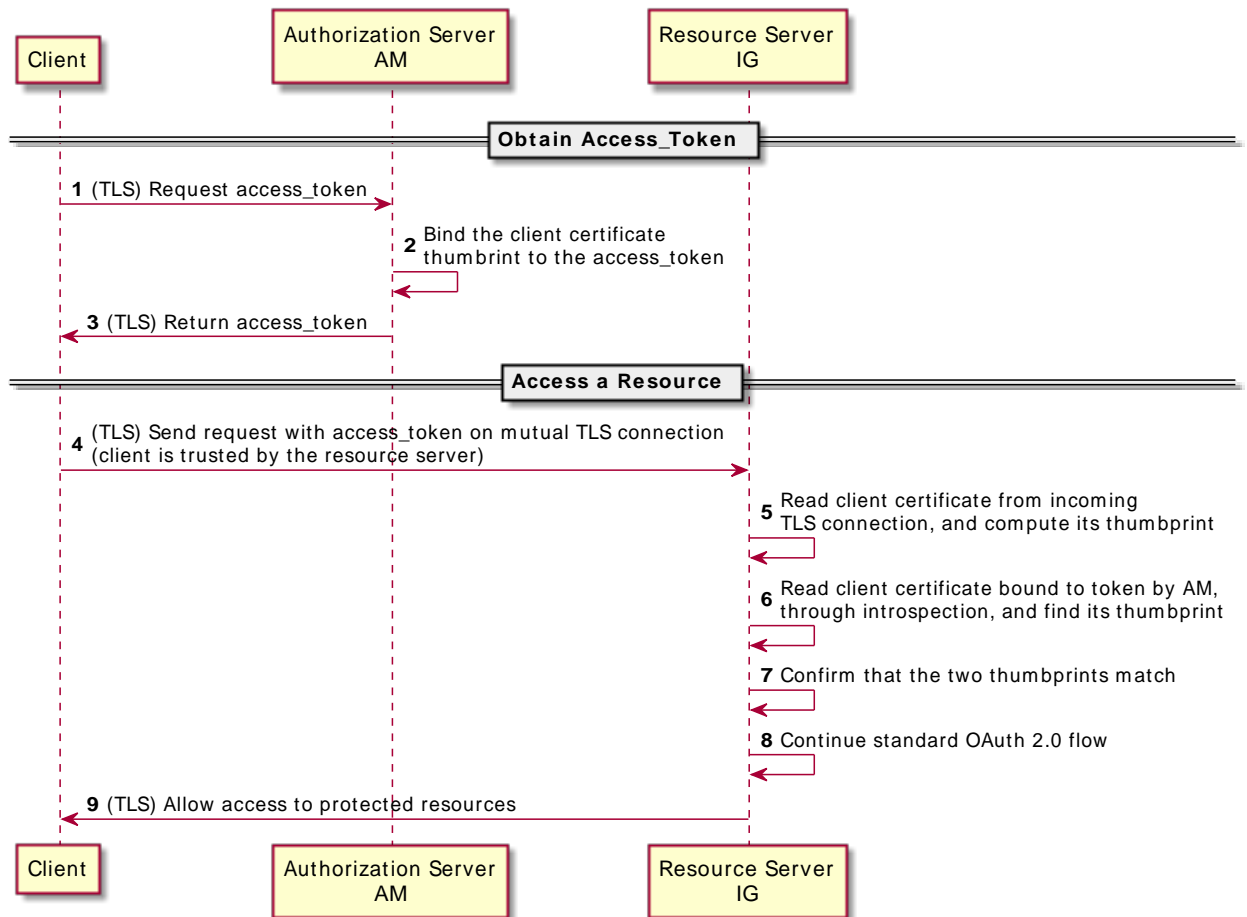
The following images illustrate the example:

*Connections for mTLS Using Standard TLS Client Certificate Authentication*





### Data Flow for mTLS Using Standard TLS Client Certificate Authentication



Perform the procedures in this section to set up and test mTLS using standard TLS client certificate authentication:

- "Set Up Keystores and Truststores"
- "Set Up AM for HTTPS (Server-Side) in Tomcat"
- "Set Up IG for HTTPS (Server-Side) in Tomcat"
- "Set Up IG for HTTPS (Server-Side) in Standalone Mode"
- "Set Up AM As an Authorization Server With mTLS"

- "Set Up IG As a Resource Server With mTLS"
- "Test the Setup"

### Set Up Keystores and Truststores

1. Locate the following keystore directories, and in a terminal create variables for them:

- `oauth2_client_keystore_directory`
- `am_keystore_directory`
- `ig_keystore_directory`

2. Create self-signed RSA key pairs for AM, IG, and the client:

```
$ keytool -genkeypair \
  -alias openam-server \
  -keyalg RSA \
  -keysize 2048 \
  -keystore $am_keystore_directory/keystore.p12 \
  -storepass changeit \
  -storetype PKCS12 \
  -keypass changeit \
  -validity 360 \
  -dname CN=openam.example.com,O=Example,C=FR
```

```
$ keytool -genkeypair \
  -alias openig-server \
  -keyalg RSA \
  -keysize 2048 \
  -keystore $ig_keystore_directory/keystore.p12 \
  -storepass changeit \
  -storetype PKCS12 \
  -keypass changeit \
  -validity 360 \
  -dname CN=openig.example.com,O=Example,C=FR
```

```
$ keytool -genkeypair \
  -alias oauth2-client \
  -keyalg RSA \
  -keysize 2048 \
  -keystore $oauth2_client_keystore_directory/keystore.p12 \
  -storepass changeit \
  -storetype PKCS12 \
  -keypass changeit \
  -validity 360 \
  -dname CN=test
```

3. Export the certificates to .pem so that the **curl** client can verify the identity of the AM and IG servers:

```
$ keytool -export \  
-rfc \  
-alias openam-server \  
-keystore $am_keystore_directory/keystore.p12 \  
-storepass changeit \  
-storetype PKCS12 \  
-file $am_keystore_directory/openam-server.cert.pem  
  
Certificate stored in file ../openam-server.cert.pem
```

```
$ keytool -export \  
-rfc \  
-alias openig-server \  
-keystore $ig_keystore_directory/keystore.p12 \  
-storepass changeit \  
-storetype PKCS12 \  
-file $ig_keystore_directory/openig-server.cert.pem  
  
Certificate stored in file openig-server.cert.pem
```

4. Extract the certificate and client private key to .pem so that the **curl** command can identify itself as the client for the HTTPS connection:

```
$ keytool -export \  
-rfc \  
-alias oauth2-client \  
-keystore $oauth2_client_keystore_directory/keystore.p12 \  
-storepass changeit \  
-storetype PKCS12 \  
-file $oauth2_client_keystore_directory/client.cert.pem  
  
Certificate stored in file ../client.cert.pem
```

```
$ openssl pkcs12 \  
-in $oauth2_client_keystore_directory/keystore.p12 \  
-nocerts \  
-nodes \  
-passin pass:changeit \  
-out $oauth2_client_keystore_directory/client.key.pem  
  
...verified OK
```

You can now delete the client keystore.

5. Create the CACerts truststore so that AM can validate the client identity:

```
$ keytool -import \  
-noprompt \  
-trustcacerts \  
-file $oauth2_client_keystore_directory/client.cert.pem \  
-keystore $oauth2_client_keystore_directory/cacerts.p12 \  
-storepass changeit \  
-storetype PKCS12 \  
-alias client-cert  
  
Certificate was added to keystore
```

## Set Up AM for HTTPS (Server-Side) in Tomcat

This procedure sets up AM for HTTPS in Tomcat. For more information, see *Configuring AM's Container for HTTPS* in AM's *Installation Guide*.

1. Add the following connector configuration to AM's Tomcat `server.xml`, replacing the values for the keystore directories with your paths:

```
<Connector port="8445" protocol="HTTP/1.1" SSLEnabled="true" scheme="https" secure="true">
  <SSLHostConfig protocols="+TLSv1.2,-TLSv1.1,-TLSv1,-SSLv2Hello,-SSLv3"
    certificateVerification="optionalNoCA"
    truststoreFile="oauth2_client_keystore_directory/cacerts.p12"
    truststorePassword="changeit"
    truststoreType="PKCS12">
    <Certificate certificateKeystoreFile="am_keystore_directory/keystore.p12"
      certificateKeystorePassword="changeit"
      certificateKeystoreType="PKCS12" />
  </SSLHostConfig>
</Connector>
```

The `optionalNoCA` property allows the presentation of client certificates to be optional. Tomcat does not check them against the list of trusted CAs.

2. In AM, export an environment variable for the base64-encoded value of the password (`changeit`) for the `cacerts.p12` truststore:

```
$ export PASSWORDSECRETID='Y2hhbmdlaXQ='
```

3. Restart AM, and make sure that you can access it on the secure port `https://openam.example.com:8445/openam`.

## Set Up IG for HTTPS (Server-Side) in Tomcat

This procedure sets up IG for HTTPS in Tomcat. For other container types, see "Configuring IG for HTTPS (Server-Side) in Jetty" and "Configuring IG for HTTPS (Server-Side) in JBoss EAP".

If IG is installed in standalone mode, follow "Set Up IG for HTTPS (Server-Side) in Standalone Mode" instead.

1. Add the following connector configuration to IG's Tomcat `server.xml`, replacing the values for the keystore directories with your paths:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true" scheme="https" secure="true">
  <SSLHostConfig protocols="+TLSv1.2,-TLSv1.1,-TLSv1,-SSLv2Hello,-SSLv3"
    certificateVerification="optionalNoCA"
    truststoreFile="oauth2_client_keystore_directory/cacerts.p12"
    truststorePassword="changeit"
    truststoreType="PKCS12">
    <Certificate certificateKeystoreFile="ig_keystore_directory/keystore.p12"
      certificateKeystorePassword="changeit"
      certificateKeystoreType="PKCS12" />
  </SSLHostConfig>
</Connector>
```

The `optionalNoCA` property allows the presentation of client certificates to be optional. Tomcat does not check them against the list of trusted CAs.

- Restart IG, and make sure that you can access the welcome page on the secure port `https://openig.example.com:8443`.

### Set Up IG for HTTPS (Server-Side) in Standalone Mode

This procedure sets up IG for HTTPS in standalone mode. Before you start, install IG in standalone mode, as described in "Downloading and Starting IG in Standalone Mode" in the *Getting Started Guide*.

When IG is installed in web container mode, follow "Set Up IG for HTTPS (Server-Side) in Tomcat" instead.

- In `ig_keystore_directory`, add a file called `keystore.pass` containing the keystore password:

```
$ cd $ig_keystore_directory
$ echo -n changeit > keystore.pass
```

- Add the following route to IG, replacing instances of `ig_keystore_directory` and `oauth2_client_keystore_directory` with your path:

Linux

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

Windows

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

```
{
  "mode": "DEVELOPMENT",
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443,
      "tls": {
        "type": "ServerTlsOptions",
        "config": {
          "alpn": {
            "enabled": true
          },
          "clientAuth": "REQUEST",
          "keyManager": {
            "type": "SecretsKeyManager",
            "config": {
              "signingSecretId": "key.manager.secret.id",
              "secretsProvider": {
                "type": "KeyStoreSecretStore",
                "config": {
                  "file": "<ig_keystore_directory>/keystore.p12",
                  "storePassword": "keystore.pass",

```

```

        "secretsProvider": "SecretsPasswords",
        "mappings": [
          {
            "secretId": "key.manager.secret.id",
            "aliases": [
              "openig-server"
            ]
          }
        ]
      }
    }
  },
  "trustManager": {
    "type": "SecretsTrustManager",
    "config": {
      "verificationSecretId": "trust.manager.secret.id",
      "secretsProvider": {
        "type": "KeyStoreSecretStore",
        "config": {
          "file": "<oauth2_client_keystore_directory>/cacerts.p12",
          "storePassword": "keystore.pass",
          "secretsProvider": "SecretsPasswords",
          "mappings": [
            {
              "secretId": "trust.manager.secret.id",
              "aliases": [
                "client-cert"
              ]
            }
          ]
        }
      }
    }
  }
},
"heap": [
  {
    "name": "SecretsPasswords",
    "type": "FileSystemSecretStore",
    "config": {
      "directory": "<ig_keystore_directory>",
      "format": "PLAIN"
    }
  }
]
}

```

Notice the following features of the route:

- IG starts on port **8080**, and on **8443** over TLS.
- IG's private keys for TLS are managed by the SecretsKeyManager, which references the KeyStoreSecretStore that holds the keys.

- The password of the KeyStoreSecretStore is provided by the FileSystemSecretStore.
- The KeyStoreSecretStore maps the keystore alias to the secret ID for retrieving the private signing keys.

### 3. Start IG:

```
$ /path/to/identity-gateway/bin/start.sh
...
... started in 1234ms on ports : [8080 8443]
```


## Set Up AM As an Authorization Server With mTLS

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.


### 1. Select Applications > Agents > Identity Gateway, add an agent with the following values:

- Agent ID: `ig_agent`
- Password: `password`
- Token Introspection: `Realm Only`

### 2. Configure an OAuth 2.0 Authorization Server:

- a. Select  Services > Add a Service > OAuth2 Provider, and add a service with the default values.
- b. On the Advanced tab, select the following value:
  - Support TLS Certificate-Bound Access Tokens: enabled

### 3. Configure an OAuth 2.0 client to request access\_tokens:

- a. Select  Applications > OAuth 2.0 > Clients, and add a client with the following values:
  - Client ID: `client-application`
  - Client secret: `password`
  - Scope(s): `test`
- b. On the Advanced tab, select the following values:
  - Grant Types: `Client Credentials`

The `password` is the only grant type used by the client in the example.
  - Token Endpoint Authentication Method: `tls_client_auth`


c. On the signing and Encryption tab, select the following values:

- mTLS Subject DN: `CN=test`

When this option is set, AM requires the subject DN in the client certificate to have the same value. This ensures that the certificate is from the client, and not just any valid certificate trusted by the trust manager.

- Use Certificate-Bound Access Tokens: Enabled

4. Set up AM secret stores to trust the client certificate:

a. Select  Secret Stores, and add a store with the following values:

- Secret Store ID: `trusted-ca-certs`
- Store Type: `Keystore`
- File: `$oauth2_client_keystore_directory/cacerts.p12`
- Keystore type: `PKCS12`
- Store password secret ID: `passwordSecretId`

b. Select Mappings and add the following mapping:

- Secret ID: `am.services.oauth2.tls.client.cert.authentication`
- Aliases: `client-cert`

When the token endpoint authentication method is `tls_client_auth`, this secret is used to validate the client certificate. Add an alias in this list for each client that uses `tls_client_auth`. For certificates signed by a CA, add the CA certificate to the list.

## Set Up IG As a Resource Server With mTLS

1. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

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

2. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/mtls-certificate.json
```

Windows

```
%appdata%\OpenIG\config\routes\mtls-certificate.json
```



```

{
  "name": "mtls-certificate",
  "condition": "${matches(request.uri.path, '/mtls-certificate')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://openam.example.com:8088/openam/",
        "version": "7"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "test"
            ],
            "requireHttps": false,
            "accessTokenResolver": {
              "type": "ConfirmationKeyVerifierAccessTokenResolver",
              "config": {
                "delegate": {
                  "name": "token-resolver-1",
                  "type": "TokenIntrospectionAccessTokenResolver",
                  "config": {
                    "amService": "AmService-1",
                    "providerHandler": {
                      "type": "Chain",
                      "config": {
                        "filters": [
                          {
                            "type": "HttpBasicAuthenticationClientFilter",
                            "config": {
                              "username": "ig_agent",
                              "passwordSecretId": "agent.secret.id",
                              "secretsProvider": "SystemAndEnvSecretStore-1"
                            }
                          }
                        ]
                      }
                    }
                  }
                }
              }
            ],
            "handler": "ForgeRockClientHandler"
          }
        }
      ]
    }
  }
}

```

```

        }
      }
    }
  }
},
1,
"handler": {
  "name": "StaticResponseHandler-1",
  "type": "StaticResponseHandler",
  "config": {
    "status": 200,
    "headers": {
      "Content-Type": [ "text/plain" ]
    },
    "entity": "mTLS\n Valid token: ${contexts.oauth2.accessToken.token}\n Confirmation keys:
${contexts.oauth2}"
  }
}
}
}
}
}
}
}
}
}
}
}

```

Notice the following features of the route:

- The route matches requests to `/mtls-certificate`.
- The `OAuth2ResourceServerFilter` uses the `ConfirmationKeyVerifierAccessTokenResolver` to validate the certificate thumbprint against the thumbprint from the resolved `access_token`, provided by AM.

The `ConfirmationKeyVerifierAccessTokenResolver` then delegates token resolution to the `TokenIntrospectionAccessTokenResolver`.

- The `providerHandler` adds an authorization header to the request, containing the username and password of the OAuth 2.0 client with the scope to examine (introspect) `access_tokens`.
- The `OAuth2ResourceServerFilter` checks that the resolved token has the required scopes, and injects the token info into the context.
- The `StaticResponseHandler` returns the content of the `access_token` from the context.

### Test the Setup

1. Get an `access_token` from AM, over TLS:

```
$ mytoken=$(curl --request POST \
--cacert $am_keystore_directory/openam-server.cert.pem \
--cert $oauth2_client_keystore_directory/client.cert.pem \
--key $oauth2_client_keystore_directory/client.key.pem \
--header 'cache-control: no-cache' \
--header 'content-type: application/x-www-form-urlencoded' \
--data 'client_id=client-application&grant_type=client_credentials&scope=test' \
https://openam.example.com:8445/openam/oauth2/access_token | jq -r .access_token)
```

## 2. Introspect the access\_token on AM:

```
$ curl --request POST \
-u ig_agent:password \
--header 'content-type: application/x-www-form-urlencoded' \
--data token=${mytoken} \
http://openam.example.com:8088/openam/oauth2/realms/root/introspect | jq

{
  "active": true,
  "scope": "test",
  "client_id": "client-application",
  "user_id": "client-application",
  "token_type": "Bearer",
  "exp": 1550590833,
  "sub": "client-application",
  "iss": "http://openam.example.com:8088/openam/oauth2",
  "cnf": {
    "x5t#S256": "T4u...R9Q"
  }
}
```

The `cnf` property indicates the value of the confirmation code, as follows:

- **x5**: X509 certificate
- **t**: thumbprint
- **#**: separator
- **S256**: algorithm used to hash the raw certificate bytes

## 3. Access the IG route to validate the token's confirmation thumbprint with the ConfirmationKeyVerifierAccessTokenResolver:

```
$ curl --request POST \
--cacert $ig_keystore_directory/openig-server.cert.pem \
--cert $oauth2_client_keystore_directory/client.cert.pem \
--key $oauth2_client_keystore_directory/client.key.pem \
--header "authorization: Bearer ${mytoken}" \
https://openig.example.com:8443/mtls-certificate

mTLS
Valid token: 2Bp...s_k
Confirmation keys: {
  ...
}
```

The validated token and confirmation keys are displayed.

## mTLS Using Trusted Headers

IG can validate the thumbprint of certificate-bound access\_tokens by reading the client certificate from a configured, trusted HTTP header.

Use this method when TLS is terminated at a reverse proxy or load balancer before IG. IG cannot authenticate the client through the TLS connection's client certificate because:

- If the connection is over TLS, the connection presents the certificate of the TLS termination point before IG.
- If the connection is not over TLS, the connection presents no client certificate.

If the client is connected directly to IG through a TLS connection, for which IG is the TLS termination point, use the example in "mTLS Using Standard TLS Client Certificate Authentication".

Configure the proxy or load balancer to:

- Forward the encoded certificate to IG in the trusted header.

Encode the certificate in an HTTP-header compatible format that can convey a full certificate, so that IG can rebuild the certificate.

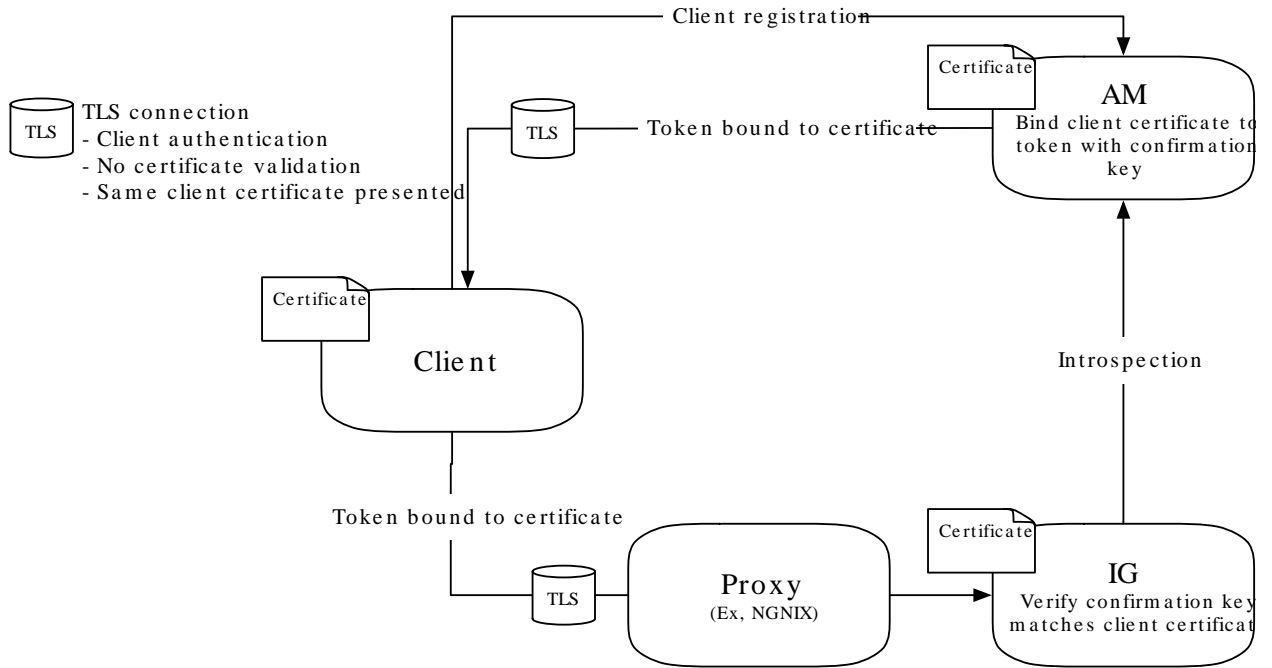
- Strip the trusted header from incoming requests, and change the default header name to something an attacker can't guess.

Because there is a trust relationship between IG and the TLS termination point, IG doesn't authenticate the contents of the trusted header. IG accepts any value in a header from a trusted TLS termination point.

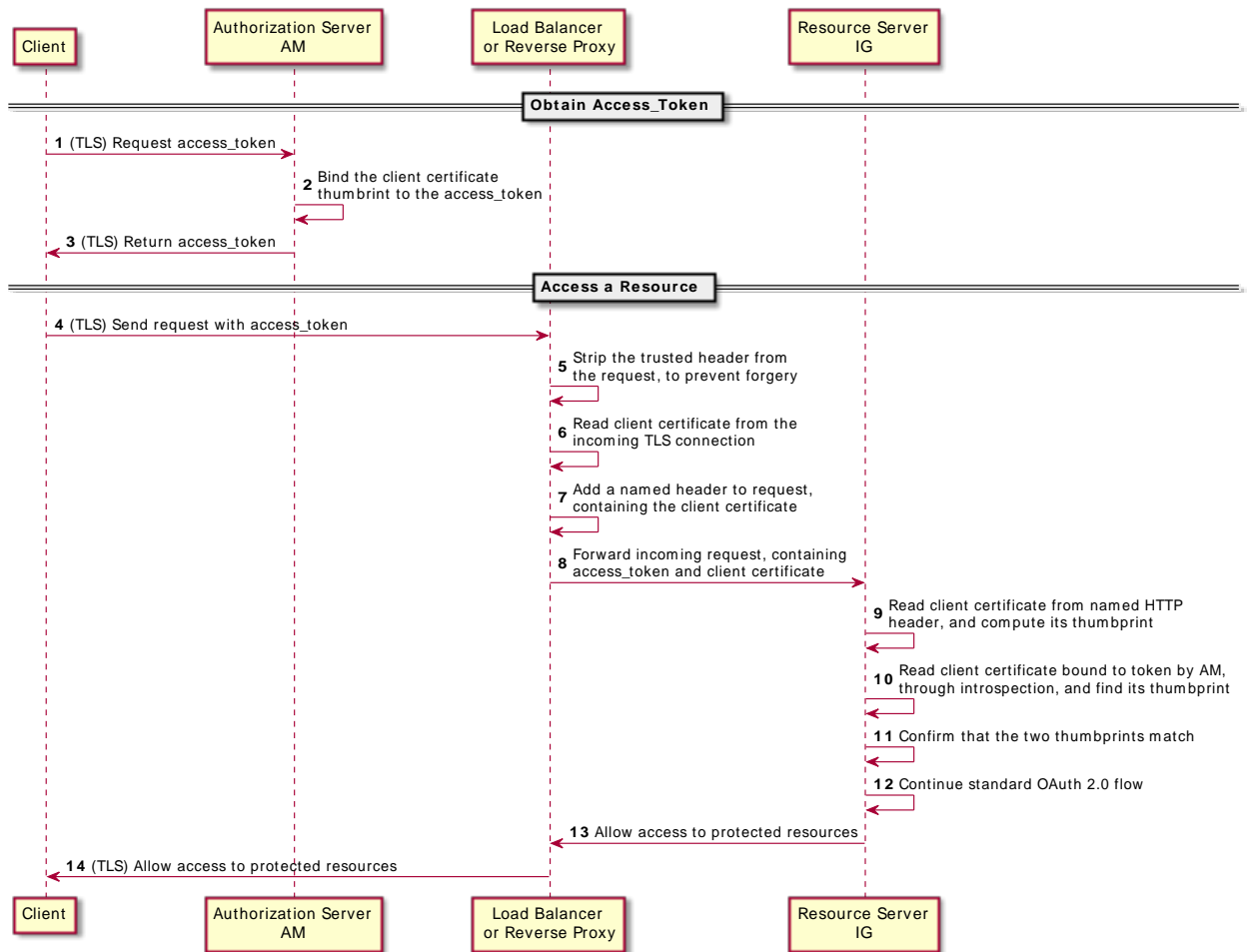
Use this example when the IG instance is running behind a load balancer or other ingress point. If the IG instance is running behind the TLS termination point, consider the example in "mTLS Using Standard TLS Client Certificate Authentication".

The following image illustrates the connections and certificates required by the example:

*Connections for mTLS Using Trusted Headers*



### Data Flow for mTLS Using Trusted Headers



### Set Up mTLS Using Trusted Headers

1. Set up the keystores, truststores, AM, and IG as described in "mTLS Using Standard TLS Client Certificate Authentication".
2. Base64-encode the value of `$oauth2_client_keystore_directory/client.cert.pem`. The value is used in the final POST.
3. Add the following route to IG:

### Linux

```
$HOME/.openig/config/routes/mtls-header.json
```

### Windows

```
%appdata%\OpenIG\config\routes\mtls-header.json
```

```
{
  "name": "mtls-header",
  "condition": "${matches(request.uri.path, '/mtls-header')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://openam.example.com:8088/openam/",
        "version": "7"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "capture": "all",
    "config": {
      "filters": [
        {
          "name": "CertificateThumbprintFilter-1",
          "type": "CertificateThumbprintFilter",
          "config": {
            "certificate": "${pemCertificate(decodeBase64(request.headers['ssl_client_cert'][0]))}",
            "failureHandler": {
              "type": "ScriptableHandler",
              "config": {
                "type": "application/x-groovy",
                "source": [
                  "def response = new Response(Status.TEAPOT);",
                  "response.entity = 'Failure in CertificateThumbprintFilter'",
                  "return response"
                ]
              }
            }
          }
        }
      ]
    }
  },
  {
    "name": "OAuth2ResourceServerFilter-1",
    "type": "OAuth2ResourceServerFilter",
    "config": {
      "scopes": [

```

```

        "test"
    ],
    "requireHttps": false,
    "accessTokenResolver": {
        "type": "ConfirmationKeyVerifierAccessTokenResolver",
        "config": {
            "delegate": {
                "name": "token-resolver-1",
                "type": "TokenIntrospectionAccessTokenResolver",
                "config": {
                    "amService": "AmService-1",
                    "providerHandler": {
                        "type": "Chain",
                        "config": {
                            "filters": [
                                {
                                    "type": "HttpBasicAuthenticationClientFilter",
                                    "config": {
                                        "username": "ig_agent",
                                        "passwordSecretId": "agent.secret.id",
                                        "secretsProvider": "SystemAndEnvSecretStore-1"
                                    }
                                }
                            ],
                            "handler": "ForgeRockClientHandler"
                        }
                    }
                }
            }
        }
    },
    "handler": {
        "name": "StaticResponseHandler-1",
        "type": "StaticResponseHandler",
        "config": {
            "status": 200,
            "headers": {
                "Content-Type": [ "text/plain" ]
            },
            "entity": "mTLS\n Valid token: ${contexts.oauth2.accessToken.token}\n Confirmation keys:
            ${contexts.oauth2}"
        }
    }
}

```

Notice the following features of the route compared to `mtls-certificate.json`:

- The route matches requests to `/mtls-header`.
- The `CertificateThumbprintFilter` extracts a Java certificate from the trusted header, computes the SHA-256 thumbprint of that certificate, and makes the thumbprint available for the `ConfirmationKeyVerifierAccessTokenResolver`.



#### 4. Test the setup:

- a. Get an access\_token from AM, over TLS:

```
$ mytoken=$(curl --request POST \
--cacert $am_keystore_directory/openam-server.cert.pem \
--cert $oauth2_client_keystore_directory/client.cert.pem \
--key $oauth2_client_keystore_directory/client.key.pem \
--header 'cache-control: no-cache' \
--header 'content-type: application/x-www-form-urlencoded' \
--data 'client_id=client-application&grant_type=client_credentials&scope=test' \
https://openam.example.com:8445/openam/oauth2/access_token | jq -r .access_token)
```

- b. Introspect the access\_token on AM:

```
$ curl --request POST \
-u ig_agent:password \
--header 'content-type: application/x-www-form-urlencoded' \
--data token=${mytoken} \
http://openam.example.com:8088/openam/oauth2/realms/root/introspect | jq

{
  "active": true,
  "scope": "test",
  "client_id": "client-application",
  "user_id": "client-application",
  "token_type": "Bearer",
  "exp": 157...994,
  "sub": "client-application",
  "iss": "http://openam.example.com:8088/openam/oauth2",
  "cnf": {
    "x5t#S256": "1QG...Wgc"
  },
  "authGrantId": "lto...8vw",
  "auditTrackingId": "119...480"
}
```

The `cnf` property indicates the value of the confirmation code, as follows:

- `x5`: X509 certificate
- `t`: thumbprint
- `#`: separator
- `S256`: algorithm used to hash the raw certificate bytes

- c. Access the IG route to validate the confirmation key, using the base64-encoded value of `$oauth2_client_keystore_directory/client.cert.pem`:

```
$ curl --request POST \  
--header "authorization:Bearer $mytoken" \  
--header 'ssl_client_cert:base64-encoded-cert' \  
http://openig.example.com:8080/mtls-header  
  
Valid token: zw5...Sj1  
Confirmation keys: {  
  ...  
}
```



The validated token and confirmation keys are displayed.

## Using the OAuth 2.0 Context to Log in to the Sample Application

The introspection returns scopes in the context. This section contains an example route that retrieves the scopes, assigns them as the IG session username and password, and uses them to log the user directly in to the sample application.

For information about the context, see "OAuth2Context" in the *Configuration Reference*.

### Log in to the Sample Application By Using the Token Info

1. Set up AM:
  - a. Set up AM as described in "Validating Access\_Tokens Through the Introspection Endpoint".
  - b. Select  Identities, and change the email address of George to `george`.
  - c. Select  Scripts > OAuth2 Access Token Modification Script, and replace the default script as follows:

```
import org.forgerock.http.protocol.Request  
import org.forgerock.http.protocol.Response  
import com.ipianet.sso.SSOException  
import groovy.json.JsonSlurper  
  
def attributes = identity.getAttributes(["mail"].toSet())  
accessToken.setField("mail", attributes["mail"][0])  
accessToken.setField("password", "C0stanza")
```

The AM script adds user profile information to the `access_token`, and adds a `password` field with the value `C0stanza`.

**Do not use this example in production!** If the token is stateless and unencrypted, the password value is easily accessible when you have the token.

2. Set up IG:

- a. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

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

- b. Add the following route to IG:

*Linux*

```
$HOME/.openig/config/routes/rs-pwreplay.json
```

*Windows*

```
%appdata%\OpenIG\config\routes\rs-pwreplay.json
```

```
{
  "name" : "rs-pwreplay",
  "baseURI" : "http://app.example.com:8081",
  "condition" : "${matches(request.uri.path, '^/rs-pwreplay')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://openam.example.com:8088/openam/",
        "version": "7"
      }
    }
  ],
  "handler" : {
    "type" : "Chain",
    "config" : {
      "filters" : [
        {
          "name" : "OAuth2ResourceServerFilter-1",
          "type" : "OAuth2ResourceServerFilter",
          "config" : {
            "scopes" : [ "mail", "employeenumber" ],
            "requireHttps" : false,
            "realm" : "OpenIG",
            "accessTokenResolver": {
              "name": "TokenIntrospectionAccessTokenResolver-1",
              "type": "TokenIntrospectionAccessTokenResolver",
              "config": {
                "amService": "AmService-1",
                "providerHandler": {
                  "type": "Chain",
                  "config": {

```

```

        "filters": [
          {
            "type": "HttpBasicAuthenticationClientFilter",
            "config": {
              "username": "ig_agent",
              "passwordSecretId": "agent.secret.id",
              "secretsProvider": "SystemAndEnvSecretStore-1"
            }
          },
          {
            "type": "AssignmentFilter",
            "config": {
              "onRequest": [
                {
                  "target": "${session.username}",
                  "value": "${contexts.oauth2.accessToken.info.mail}"
                },
                {
                  "target": "${session.password}",
                  "value": "${contexts.oauth2.accessToken.info.password}"
                }
              ]
            }
          },
          {
            "type": "StaticRequestFilter",
            "config": {
              "method": "POST",
              "uri": "http://app.example.com:8081/login",
              "form": {
                "username": [
                  "${session.username}"
                ],
                "password": [
                  "${session.password}"
                ]
              }
            }
          }
        ],
        "handler": "ReverseProxyHandler"
      }
    ]
  }
}

```

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

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

- The AssignmentFilter accesses the context, and injects the username and password into the SessionContext, `${session}`.
- The StaticRequestFilter retrieves the username and password from `session`, and replaces the original HTTP GET request with an HTTP POST login request that contains the credentials to authenticate.

### 3. Test the setup:

- a. In a terminal window, use a **curl** command similar to the following to retrieve an `access_token`:

```
$ mytoken=$(curl -s \  
--user "client-application:password" \  
--data "grant_type=password&username=george&password=C0stanza&scope=mail%20employeeenumber" \  
http://openam.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
```

- b. Validate the `access_token` returned in the previous step:

```
$ curl -v http://openig.example.com:8080/rs-pwreplay --header "Authorization: Bearer ${mytoken}"
```

HTML for the sample application is displayed.

## Caching Access\_Tokens

This section builds on the example in "Validating Access\_Tokens Through the Introspection Endpoint" to cache `access_tokens`.

When the `access_token` is not cached, IG calls AM to validate the `access_token`. After the `access_token` is cached, IG doesn't validate the `access_token` with AM.

(From AM 6.5.3.) When an `access_token` is revoked on AM, the `CacheAccessTokenResolver` can delete the token from the cache when both of the following conditions are true:

- The `notification` property of `AmService` is enabled.
- The delegate `AccessTokenResolver` provides the token metadata required to update the cache.

When a `refresh_token` is revoked on AM, all associated `access_tokens` are automatically and immediately revoked.

### Cache Access\_Tokens

1. Set up AM as described in "Validating Access\_Tokens Through the Introspection Endpoint".
2. Set up IG:
  - a. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcnQ='
```

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

- b. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/rs-introspect-cache.json
```

Windows

```
%appdata%\OpenIG\config\routes\rs-introspect-cache.json
```

```
{
  "name": "rs-introspect-cache",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/rs-introspect-cache$')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "url": "http://openam.example.com:8088/openam",
        "realm": "/",
        "version": "7",
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "name": "OAuth2ResourceServerFilter-1",
          "type": "OAuth2ResourceServerFilter",
          "config": {
            "scopes": [
              "mail",
              "employeenumber"
            ],
            "requireHttps": false,
            "realm": "OpenIG",
            "accessTokenResolver": {
              "name": "CacheAccessTokenResolver-1",
              "type": "CacheAccessTokenResolver",
              "config": {
                "enabled": true,
                "defaultTimeout": "1 hour",
                "maximumTimeToCache": "1 day",
                "delegate": {
```

```

"name": "TokenIntrospectionAccessTokenResolver-1",
"type": "TokenIntrospectionAccessTokenResolver",
"config": {
  "amService": "AmService-1",
  "providerHandler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "HttpBasicAuthenticationClientFilter",
          "config": {
            "username": "ig_agent",
            "passwordSecretId": "agent.secret.id",
            "secretsProvider": "SystemAndEnvSecretStore-1"
          }
        }
      ],
      "handler": {
        "type": "Delegate",
        "capture": "all",
        "config": {
          "delegate": "ForgeRockClientHandler"
        }
      }
    }
  },
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/html" ]
      },
      "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></body></html>"
    }
  }
}

```

Notice the following features of the route compared to `rs-introspect.json`, in "Validating Access\_Tokens Through the Introspection Endpoint":

- The `OAuth2ResourceServerFilter` uses a `CacheAccessTokenResolver` to cache the `access_token`, and then delegate token resolution to the `TokenIntrospectionAccessTokenResolver`.

- The `TokenIntrospectionAccessTokenResolver` uses a `ForgeRockClientHandler` and a capture decorator to capture IG's interactions with AM.

### 3. Test the setup:

- In a terminal window, use a **curl** command similar to the following to retrieve an `access_token`:

```
$ mytoken=$(curl -s \
--user "client-application:password" \
--data "grant_type=password&username=george&password=C0stanza&scope=mail%20employeeenumber" \
http://openam.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
```

- Access the route, using the `access_token` returned in the previous step:

```
$ curl http://openig.example.com:8080/rs-introspect-cache --header "Authorization: Bearer
${mytoken}"

{
  active = true,
  scope = employeeenumber mail,
  client_id = client - application,
  user_id = george,
  token_type = Bearer,
  exp = 158...907,
  sub = george,
  iss = http://openam.example.com:8088/openam/oauth2, ...
  ...
}
```

- In the route log, note that IG calls AM to introspect the `access_token`:

```
POST http://openam.example.com:8088/openam/oauth2/realms/root/introspect HTTP/1.1
```

- Access the route again, and in the route log note that this time IG doesn't call AM, because the token is cached.

Disable the cache and repeat the previous steps to cause IG to call AM to validate the `access_token` for each request.



## Chapter 11

# Acting As an OpenID Connect Relying Party

The following sections provide an overview of how IG supports OpenID Connect 1.0, and examples of to set up IG as an OpenID Connect relying party in different deployment scenarios:

- "About IG With OpenID Connect"
- "Using AM As a Single OpenID Connect Provider"
- "Using Multiple OpenID Connect Providers"
- "Discovering and Dynamically Registering With OpenID Connect Providers"

## About IG With OpenID Connect

IG supports OpenID Connect 1.0, an authentication layer built on OAuth 2.0. OpenID Connect 1.0 is a specific implementation of OAuth 2.0, where the identity provider holds the protected resource that the third-party application wants to access. For more information, see [OpenID Connect](#).

OpenID Connect refers to the following entities:

- *End user*: An OAuth 2.0 resource owner whose user information the application needs to access.

The end user wants to use an application through an existing identity provider account without signing up and creating credentials for another web service.

- *Relying Party (RP)*: An OAuth 2.0 client that needs access to the end user's protected user information.

For example, an online mail application needs to know which end user is accessing the application in order to present the correct inbox.

As another example, an online shopping site needs to know which end user is accessing the site in order to present the right offerings, account, and shopping cart.

- *OpenID Provider (OP)*: An OAuth 2.0 authorization server and also resource server that holds the user information and grants access.

The OP requires the end user to give the RP permission to access to some of its user information. Because OpenID Connect 1.0 defines unique identification for an account (subject identifier + issuer identifier), the RP can use that identification to bind its own user profile to a remote identity.

For the online mail application, this key could be used to access the mailboxes and related account information. For the online shopping site, this key could be used to access the offerings, account, shopping cart and others. The key makes it possible to serve users as if they had local accounts.

- *UserInfo*: The protected resource that the third-party application wants to access. The information about the authenticated end user is expressed in a standard format. The user-info endpoint is hosted on the authorization server and is protected with OAuth 2.0.

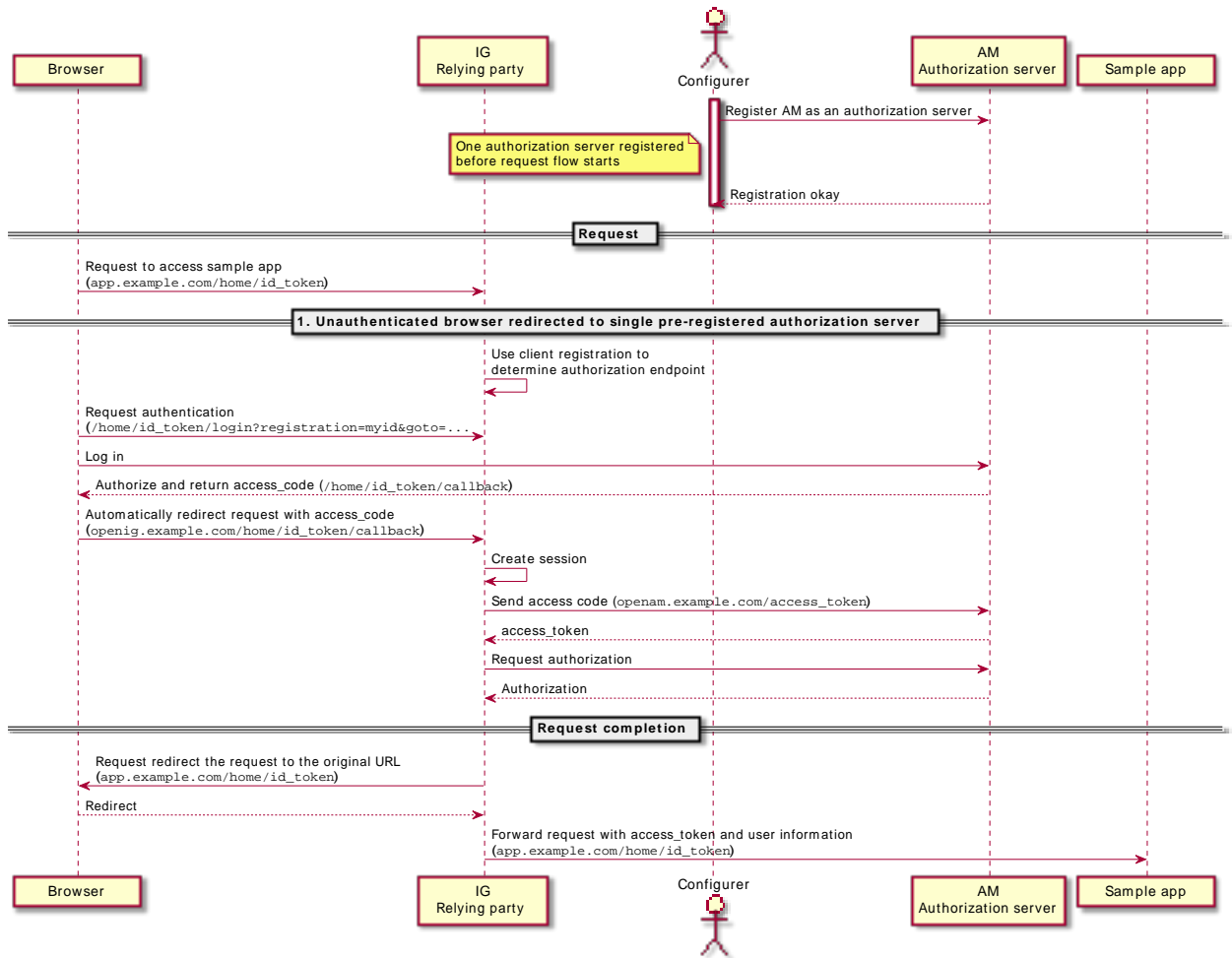
When IG acts as an OpenID Connect relying party, its role is to retrieve user information from the OpenID provider, and then to inject that information into the context for use by subsequent filters and handlers.

## Using AM As a Single OpenID Connect Provider

This section gives an example of how to set up AM as an OpenID Connect identity provider, and IG as a relying party for browser requests to the home page of the sample application.

The following sequence diagram shows the flow of information for a request to access the home page of the sample application, using AM as a single, preregistered OpenID Connect identity provider, and IG as a relying party:




Information Flow for Requests Using AM as a Single OpenID Connect Identity Provider



## Use AM As a Single OpenID Connect Provider

Before you start, prepare AM, IG, and the sample application as described in "Example Installation for This Guide".

1. Set Up AM as an OpenID Connect provider:
  - a. Select Identities, and add a user with the following values:

- ID/username: `george`
  - First name: `george`
  - Last name: `costanza`
  - Password: `C0stanza`
  - Email Address: `george@example.com`
  - Employee number: `123`
- b. (For AM 6.5.3 and later versions) Select  Services > Add a Service, and add a Validation Service with the following Valid goto URL Resources:
- `http://openig.example.com:8080/*`
  - `http://openig.example.com:8080/*?*`
- c. Create an OAuth 2.0 Authorization Server:
- i. Select  Services > Add a Service > OAuth2 Provider.
  - ii. Add a service with the default values.
- d. Create an OAuth 2.0 Client to request OAuth 2.0 access\_tokens:
- i. Select  Applications > OAuth 2.0 > Clients.
  - ii. 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`
- e. (From AM 6.5) On the Advanced tab, select the following values:
- Grant Types: `Authorization Code` and `Resource Owner Password Credentials`
- f. On the Signing and Encryption tab, change ID Token Signing Algorithm to `HS256`, `HS384`, or `HS512`. The algorithm must be HMAC.
- g. Log out of AM.
2. Set up IG:

- a. Set an environment variable for `oidc_client`, and then restart IG:

```
$ export OIDC_SECRET_ID='cGFzc3dvcmQ='
```

- b. Add the following route to IG, to serve `.css` and other static resources for the sample application:

Linux

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

Windows

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

- c. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/07-openid.json
```

Windows

```
%appdata%\OpenIG\config\routes\07-openid.json
```

```
{
  "name": "07-openid",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/id_token')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    }
  ],
  "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": "Error in OAuth 2.0 setup."
  },
  "registrations": [
    {
      "name": "oidc-user-info-client",
      "type": "ClientRegistration",
      "config": {
        "clientId": "oidc_client",
        "clientSecretId": "oidc.secret.id",
        "issuer": {
          "name": "Issuer",
          "type": "Issuer",
          "config": {
            "wellKnownEndpoint": "http://openam.example.com:8088/openam/oauth2/.well-known/openid-configuration"
          }
        },
        "scopes": [
          "openid",
          "profile",
          "email"
        ],
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "tokenEndpointAuthMethod": "client_secret_basic"
      }
    }
  ],
  "requireHttps": false,
  "cacheExpiration": "disabled"
}
},
"handler": "ReverseProxyHandler"
}
}
}

```

For information about how to set up the IG route in Studio, see "OpenID Connect Relying Party in Structured Editor" in the *Studio User Guide*.

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 "Using AM As a Single OpenID Connect Provider".
- The filter has a base client endpoint of `/home/id_token`, which creates the following service URIs:
  - Requests to `/home/id_token/login` start the delegated authorization process.

- Requests to `/home/id_token/callback` are expected as redirects from the OAuth 2.0 Authorization Server (OpenID Connect provider). This is why the redirect URI in the client profile in AM is set to `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.

### 3. Test the setup:

- a. If you are logged in to AM, log out and clear any cookies.
- b. Go to `http://openig.example.com:8080/home/id_token`.

The AM login page is displayed.

- c. Log in to AM as user `george`, password `C0stanza`, and then allow the application to access user information.

The home page of the sample application is displayed.

## Authenticating Automatically to the Sample Application

To authenticate automatically to the sample application, change the last name of the user `george` to match the password `C0stanza`, and add a `StaticRequestFilter` like the following to the end of the chain in `07-openid.json`:

```
{
  "type": "StaticRequestFilter",
  "config": {
    "method": "POST",
    "uri": "http://app.example.com:8081/login",
    "form": {
      "username": [
        "${attributes.openid.user_info.sub}"
      ],
      "password": [
        "${attributes.openid.user_info.family_name}"
      ]
    }
  }
}
```

The `StaticRequestFilter` retrieves the username and password from the context, and replaces the original HTTP GET request with an HTTP POST login request containing credentials.

## Using Multiple OpenID Connect Providers

This section builds on the example in "Using AM As a Single OpenID Connect Provider" to give an example of using OpenID Connect with two identity providers.

The client registration for the AM provider is declared in the heap, and a second client registration defines Google as an alternative identity provider. The Nascar page helps the user to choose an identity provider.

### Set Up Multiple OpenID Connect Providers

1. Set up AM as the first OpenID Connect provider, as described in "Use AM As a Single OpenID Connect Provider".
2. Set up Google as the second OpenID Connect identity provider, using the following hints:
  1. Go to <https://console.cloud.google.com/apis/credentials>.
  2. Create credentials for an OAuth 2.0 client ID with the following options:
    - Application type: [Web application](#)
    - Authorized redirect URI: [http://openid.example.com:8080/home/id\\_token/callback](http://openid.example.com:8080/home/id_token/callback)
  3. Make a note of the ID and password for the Google identity provider.

### Set Up IG for Multiple OpenID Connect Providers

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

Linux

```
$HOME/.openid/config/routes/static-resources.json
```

Windows

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

{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

2. Add the following route to IG:

Linux



```
$HOME/.openig/config/routes/07-openid-nascar.json
```

#### Windows

```
%appdata%\OpenIG\config\routes\07-openid-nascar.json
```

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "openam",
      "type": "ClientRegistration",
      "config": {
        "clientId": "oidc_client",
        "clientSecretId": "oidc.secret.id",
        "issuer": {
          "name": "Issuer",
          "type": "Issuer",
          "config": {
            "wellKnownEndpoint": "http://openam.example.com:8088/openam/oauth2/.well-known/openid-configuration"
          }
        },
        "scopes": [
          "openid",
          "profile",
          "email"
        ],
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "tokenEndpointAuthMethod": "client_secret_basic"
      }
    },
    {
      "name": "google",
      "type": "ClientRegistration",
      "config": {
        "clientId": "googleClientId",
        "clientSecretId": "google.secret.id",
        "issuer": {
          "name": "accounts.google.com",
          "type": "Issuer",
          "config": {
            "wellKnownEndpoint": "https://accounts.google.com/.well-known/openid-configuration"
          }
        },
        "scopes": [
          "openid",
          "profile"
        ],
        "secretsProvider": "SystemAndEnvSecretStore-1"
      }
    },
    {
      "name": "NascarPage",
      "type": "StaticResponseHandler",
    }
  ]
}
```

```

        "config": {
            "status": 200,
            "headers": {
                "Content-Type": [ "text/html" ]
            },
            "entity": "<html><body><p><a href='/home/id_token/login?
registration=oidc_client&issuer=Issuer&goto=${urlEncodeQueryParameterNameOrValue('http://
openig.example.com:8080/home/id_token')}>AM Login</a></p><p><a href='/home/
id_token/login?registration=googleClientId&issuer=accounts.google.com&goto=
${urlEncodeQueryParameterNameOrValue('http://openig.example.com:8080/home/id_token')}>Google Login</
a></p></body></html>"
        }
    },
    "name": "07-openid-nascar",
    "baseURI": "http://app.example.com:8081",
    "condition": "${matches(request.uri.path, '^/home/id_token')}",
    "handler": {
        "type": "Chain",
        "config": {
            "filters": [
                {
                    "type": "OAuth2ClientFilter",
                    "config": {
                        "clientEndpoint": "/home/id_token",
                        "failureHandler": {
                            "type": "StaticResponseHandler",
                            "config": {
                                "comment": "Trivial failure handler for debugging only",
                                "status": 500,
                                "reason": "Error",
                                "headers": {
                                    "Content-Type": [ "text/plain" ]
                                },
                                "entity": "${attributes.openid}"
                            },
                            "loginHandler": "NascarPage",
                            "registrations": [ "openam", "google" ],
                            "requireHttps": false,
                            "cacheExpiration": "disabled"
                        }
                    },
                    "handler": "ReverseProxyHandler"
                }
            ]
        }
    }
}

```

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

- The heap objects `openam` and `google` define two client registrations to authenticate IG to identity providers.
- The heap object `NascarPage` is a `StaticResponseHandler` that provides links to the two client registrations.

- The OAuth2ClientFilter uses a `loginHandler` that refers to `NascarPage` to allow users to choose from the two client registrations.
3. In the route, replace both occurrences of `googleClientId` by the Google identity provider ID retrieved in "Set Up Multiple OpenID Connect Providers".
  4. Set environment variables for the identity providers' passwords:
    - a. Set an environment variable for the password of the AM identity provider, `oidc_client`:

```
$ export OIDC.SECRET.ID='cGFzc3dvcmQ='
```
    - b. Set an environment variable for the password of the Google identity provider, using the password retrieved in "Set Up Multiple OpenID Connect Providers":

```
$ export GOOGLE.SECRET.ID='base64-encoded-google-client-password'
```

The passwords are retrieved by the default `SystemAndEnvSecretStore`, and must be base64-encoded.

### Test the Setup

1. Log out of AM.
2. Go to `http://openig.example.com:8080/home/id_token`.

The Nascar page offers the choice of identity provider.

3. Select a provider, log in with your credentials, and then allow the application to access user information.

For AM, use the following credentials: username `george`, password `C0stanza`. For the Google identity provider, use the Google credentials.

The home page of the sample application is displayed.

## Discovering and Dynamically Registering With OpenID Connect Providers

OpenID Connect defines mechanisms for discovering and dynamically registering with an identity provider that is not known in advance, as specified in the following publications: OpenID Connect Discovery, OpenID Connect Dynamic Client Registration, and RFC 7591 - OAuth 2.0 Dynamic Client Registration Protocol.

In dynamic registration, issuer and client registrations are generated dynamically. They are held in memory and can be reused, but do not persist when IG is restarted.



This section builds on the example in "Using AM As a Single OpenID Connect Provider" to give an example of discovering and dynamically registering with an identity provider that is not known in advance. In this example, the client sends a signed JWT to the authorization server.

To facilitate the example, a WebFinger service is embedded in the sample application. In a normal deployment, the WebFinger server is likely to be a service on the issuer's domain.

## Dynamic Registration With OpenID Connect Providers

1. Create a key `/path/to/keystore.jks`:

```
$ keytool -genkey \  
-alias myprivatekeyalias \  
-keyalg RSA \  
-keysize 2048 \  
-keystore /path/to/keystore.jks \  
-storepass keystore \  
-storetype JKS \  
-keypass keystore \  
-validity 360 \  
-dname "CN=openig.example.com, OU=example, O=com, L=fr, ST=fr, C=fr"
```

2. Set up AM:
  - a. Set up AM as described in "Use AM As a Single OpenID Connect Provider".
  - b. Select the user `george`, and change the last name to `C0stanza`. Note that, for this example, the last name must be the same as the password.
  - c. Configure the OAuth 2.0 Authorization Server for dynamic registration:
    - i. Select  Services > OAuth2 Provider.
    - ii. On the Advanced tab, add the following scopes to Client Registration Scope Whitelist: `openid, profile, email`.
    - iii. On the Client Dynamic Registration tab, select these settings:
      - Allow Open Dynamic Client Registration: Enabled
      - Generate Registration Access Tokens: Disabled
  - d. Configure the authentication method for the OAuth 2.0 Client:
    - i. Select  Applications > OAuth 2.0 > Clients.
    - ii. Select `oidc_client`, and on the Advanced tab, select Token Endpoint Authentication Method: `private_key_jwt`.
3. Set up IG:

- a. In the IG configuration, set an environment variable for the KeyStore password, and then restart IG:

```
$ export KEYSTORE_SECRET_ID='a2V5c3RvcnU='
```

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

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

Linux

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

Windows

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition" : "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

- c. Add the following script to IG:

Linux

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

Windows

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

```
/*
 * OIDC discovery with the sample application
 */
response = new Response(Status.OK)
response.getHeaders().put(ContentTypeHeader.NAME, "text/html");
response.entity = ""
<!doctype html>
<html>
  <head>
    <title>OpenID Connect Discovery</title>
    <meta charset='UTF-8'>
  </head>
  <body>
    <form id='form' action='/discovery/login?'>
      Enter your user ID or email address:
      <input type='text' id='discovery' name='discovery'
        placeholder='george or george@example.com' />
      <input type='hidden' name='goto'
        value='${contexts.router.originalUri}' />
    </form>
  </script>
```

```
// Make sure sampleAppUrl is correct for your sample app.
window.onload = function() {
  document.getElementById('form').onsubmit = function() {
    // Fix the URL if not using the default settings.
    var sampleAppUrl = 'http://app.example.com:8081/';
    var discovery = document.getElementById('discovery');
    discovery.value = sampleAppUrl + discovery.value.split('@', 1)[0];
  };
};
</script>
</body>
</html>"" as String
return response
```

The script transforms the input into a **discovery** value for IG. This is not a requirement for deployment, only a convenience for the purposes of this example. Alternatives are described in the discovery protocol specification.

- d. Add the following route to IG, replacing `/path/to/keystore.jks` with your path:

Linux

```
$HOME/.openig/config/routes/07-discovery.json
```

Windows

```
%appdata%\OpenIG\config\routes\07-discovery.json
```

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "SecretsProvider-1",
      "type": "SecretsProvider",
      "config": {
        "stores": [
          {
            "type": "KeyStoreSecretStore",
            "config": {
              "file": "/path/to/keystore.jks",
              "mappings": [
                {
                  "aliases": [ "myprivatekeyalias" ],
                  "secretId": "private.key.jwt.signing.key"
                }
              ]
            },
            "storePassword": "keystore.secret.id",
            "storeType": "JKS",
            "secretsProvider": "SystemAndEnvSecretStore-1"
          }
        ]
      }
    }
  ],
  "secretsProvider": "SystemAndEnvSecretStore-1"
}
```

```

        "name": "DiscoveryPage",
        "type": "ScriptableHandler",
        "config": {
            "type": "application/x-groovy",
            "file": "discovery.groovy"
        }
    }
},
"name": "07-discovery",
"baseURI": "http://app.example.com:8081",
"condition": "${matches(request.uri.path, '^/discovery')}",
"handler": {
    "type": "Chain",
    "config": {
        "filters": [
            {
                "name": "DynamicallyRegisteredClient",
                "type": "OAuth2ClientFilter",
                "config": {
                    "clientEndpoint": "/discovery",
                    "requireHttps": false,
                    "requireLogin": true,
                    "target": "${attributes.openid}",
                    "failureHandler": {
                        "type": "StaticResponseHandler",
                        "config": {
                            "comment": "Trivial failure handler for debugging only",
                            "status": 500,
                            "reason": "Error",
                            "headers": {
                                "Content-Type": [ "text/plain" ]
                            },
                            "entity": "${attributes.openid}"
                        }
                    }
                }
            },
            "loginHandler": "DiscoveryPage",
            "discoverySecretId": "private.key.jwt.signing.key",
            "tokenEndpointAuthMethod": "private_key_jwt",
            "secretsProvider": "SecretsProvider-1",
            "metadata": {
                "client_name": "My Dynamically Registered Client",
                "redirect_uris": [ "http://openid.example.com:8080/discovery/callback" ],
                "scopes": [ "openid", "profile", "email" ]
            }
        }
    }
},
{
    "type": "StaticRequestFilter",
    "config": {
        "method": "POST",
        "uri": "http://app.example.com:8081/login",
        "form": {
            "username": [
                "${attributes.openid.user_info.sub}"
            ],
            "password": [
                "${attributes.openid.user_info.family_name}"
            ]
        }
    }
}

```

```
    }  
  },  
  "handler": "ReverseProxyHandler"  
}  
}
```

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

- The route matches requests to `/discovery`.
- The OAuth2ClientFilter uses `DiscoveryPage` as the login handler, and specifies metadata to prepare the dynamic registration request.
- `DiscoveryPage` uses a `ScriptableHandler` and script to provide the `discovery` parameter and `goto` parameter.

If there is a match, then it can use the issuer's registration endpoint and avoid an additional request to look up the user's issuer using the WebFinger protocol.

If there is no match, IG uses the `discovery` value as the `resource` for a WebFinger request using OpenID Connect Discovery protocol.

- IG uses the `discovery` parameter to find an identity provider. IG extracts the domain host and port from the value, and attempts to find a match in the `supportedDomains` lists for issuers configured for the route.
- When `discoverySecretId` is set, the `tokenEndpointAuthMethod` is always `private_key_jwt`. Clients send a signed JWT to the authorization server.

Redirects IG to the end user's browser, using the `goto` parameter, after the process is complete and IG has injected the OpenID Connect user information into the context.

#### 4. Test the setup:

- a. Log out of AM.
- b. Go to `http://openig.example.com:8080/discovery`.
- c. Enter the following email address: `george@example.com`. The AM login page is displayed.
- d. Log in as user `george`, password `C0stanza`, and then allow the application to access user information. The sample application returns George's page.



## Chapter 12

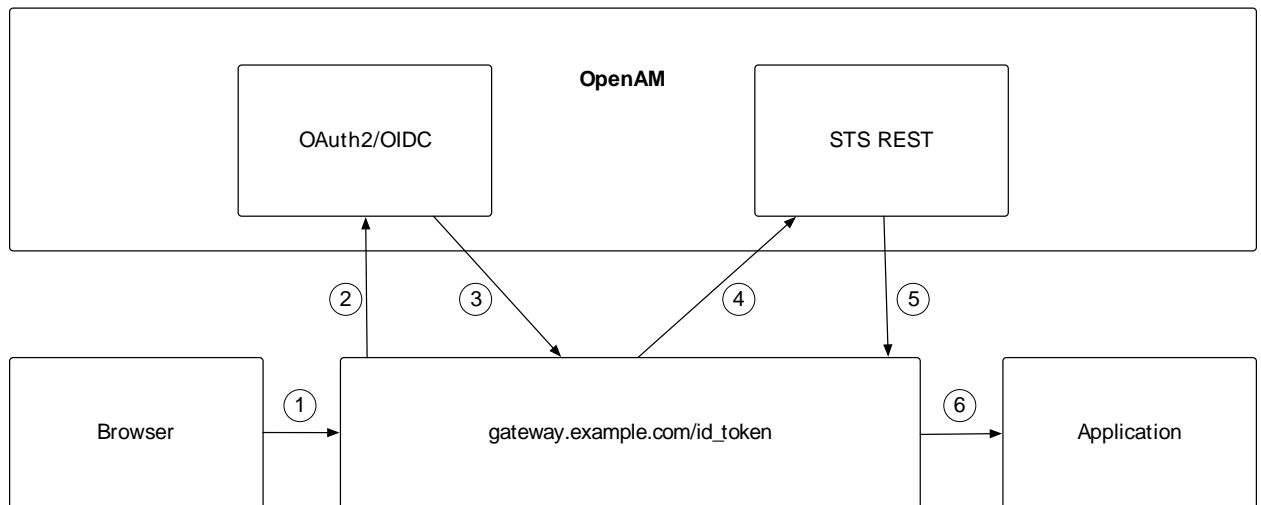
# Transforming OpenID Connect ID Tokens Into SAML Assertions

This chapter builds on the example in *"Acting As an OpenID Connect Relying Party"* to transform OpenID Connect ID tokens into SAML 2.0 assertions.

Many enterprises use existing or legacy, SAML 2.0-based SSO, but many mobile and social applications are managed by OpenID Connect. Use the IG TokenTransformationFilter to bridge the gap between OpenID Connect and SAML 2.0 frameworks.

The following figure illustrates the data flow. For a more detailed view of the flow, see *"Flow of Events"*.

*Token Transformation*



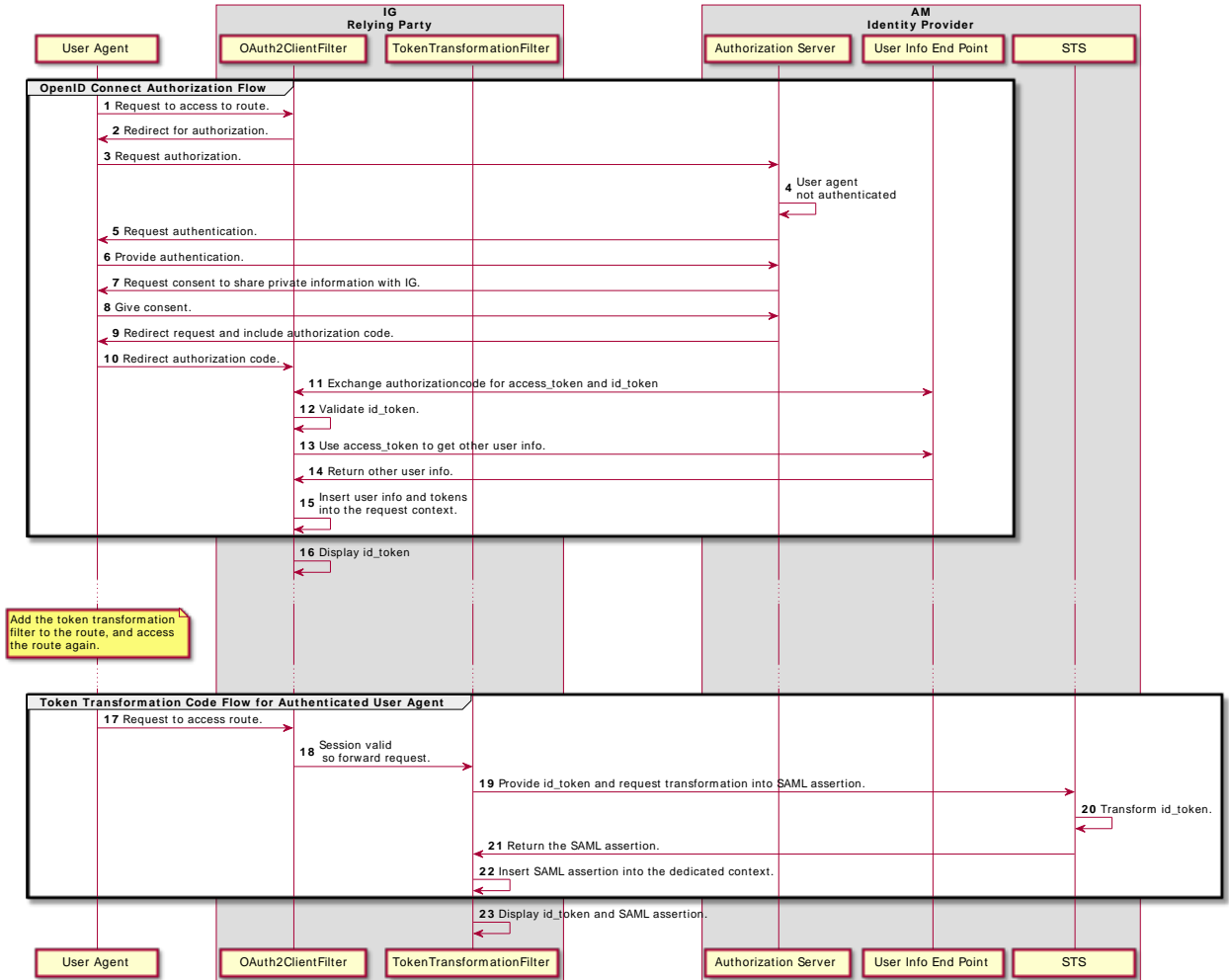
1. A user tries to access to a protected resource.
2. If the user is not authenticated, the OAuth2ClientFilter redirects the request to AM. After authentication, AM asks for the user's consent to give IG access to private information.

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.

The following sequence diagram shows a more detailed view of the flow:

## Flow of Events

Token Transformation Code Flow for Unauthenticated User Agent



## Transform OpenID Connect ID Tokens Into SAML Assertions

1. Set up an AM Security Token Service (STS), where the subject confirmation method is Bearer. For more information about setting up a REST STS instance, see AM's *Security Token Service (STS) Guide*.
  - a. Set up AM as described in "Use AM As a Single OpenID Connect Provider".


b. Select Applications > Agents > Identity Gateway, add an agent with the following values:

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

Leave all other values as default.

For AM 6.5.x and earlier versions, set up an agent as described in "Set Up an IG Agent in AM 6.5 and Earlier".

c. Create a Bearer Module:

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

- Module name: `oidc`
- Type: `OpenID Connect id_token bearer`

ii. In the configuration page, enter the following values:

- OpenID Connect validation configuration type: `Client Secret`
- OpenID Connect validation configuration value: `password`  
This is the password of the OAuth 2.0/OpenID Connect client.
- Client secret: `password`
- Name of OpenID Connect ID Token Issuer: `http://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`

Leave all other values as default, and save your settings.

d. Create an instance of STS REST.

i. In the top level realm, select STS, and add a Rest STS instance with the following values:

- Deployment URL Element: `openig`

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

- SAML2 Token

- SAML2 issuer Id: `OpenAM`
- Service Provider Entity Id: `openid_sp`
- NameIdFormat: Select `urn:oasis:names:tc:SAML:2.0: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 "Using AM As a Single OpenID Connect Provider", for example, `HMAC SHA 256`
  - Client Secret: `password`
  - Issued Tokens Audience: `oidc_client`

ii. On the SAML 2 Token tab, add the following Attribute Mappings:

- Key: `userName`, Value: `uid`
- Key: `password`, Value: `mail`

e. Log out of AM.

2. Set up IG:

a. Set an environment variable for `oidc_client` and `ig_agent`, and then restart IG:

```
$ export OIDC_SECRET_ID='cGFzc3dvcmQ='
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

b. Add the following route to IG:

*Linux*

```
$HOME/.openid/config/routes/50-idtoken.json
```

*Windows*

```
%appdata%\OpenIG\config\routes\50-idtoken.json
```

```
{
  "name": "50-idtoken",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/id_token')}",
  "heap": [
    {
```

```

        "name": "SystemAndEnvSecretStore-1",
        "type": "SystemAndEnvSecretStore"
    },
    {
        "name": "AmService-1",
        "type": "AmService",
        "config": {
            "agent": {
                "username": "ig_agent",
                "passwordSecretId": "agent.secret.id"
            },
            "secretsProvider": "SystemAndEnvSecretStore-1",
            "url": "http://openam.example.com:8088/openam/",
            "version": "7"
        }
    }
],
"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",
                    "clientSecretId": "oidc.secret.id",
                    "secretsProvider": "SystemAndEnvSecretStore-1",
                    "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": {
    "idToken": "${attributes.openid.id_token}",
    "instance": "openid",
    "amService": "AmService-1"
  }
},
],
"handler": {
  "type": "StaticResponseHandler",
  "config": {
    "reason": "Found",
    "status": 200,
    "headers": {
      "Content-Type": [ "text/plain" ]
    },
    "entity": "{$\"id_token\":\n\n\"${attributes.openid.id_token}\"} \n\n\n{\"saml_assertions\n\":\n\n\"${contexts.sts.issuedToken}\"}"
  }
}
}
}
}
```

For information about how to set up the IG route in Studio, see "Token Transformation in Structured Editor" in the *Studio User Guide*.

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 "Using AM As a Single OpenID Connect Provider", and used by IG to connect with AM.
- The TokenTransformationFilter transforms an id\_token into a SAML assertion:

- The `id_token` parameter defines where this filter gets the id\_token created by the `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}`.

### 3. Test the setup:

- a. Go to `http://openig.example.com:8080/home/id_token`.

The AM login screen is displayed.

- b. Log in to AM as username `george`, password `C0stanza`.

An OpenID Connect request to access private information is displayed.

- c. Select Allow.

The id\_token and SAML assertions are displayed:

```
{"id_token": "eyJhdHlwIjogIkpXVCIsICJhbGciOiAiSFMyNTYiIH0.eyJhYXRfaGFzaCI6I6ICJ . . ."}
{"saml_assertions":
<"saml:Assertion xmlns:saml="urn:oasis:names:tc:SAML:2.0:assertion" Version= . . ."}

```



## Chapter 13

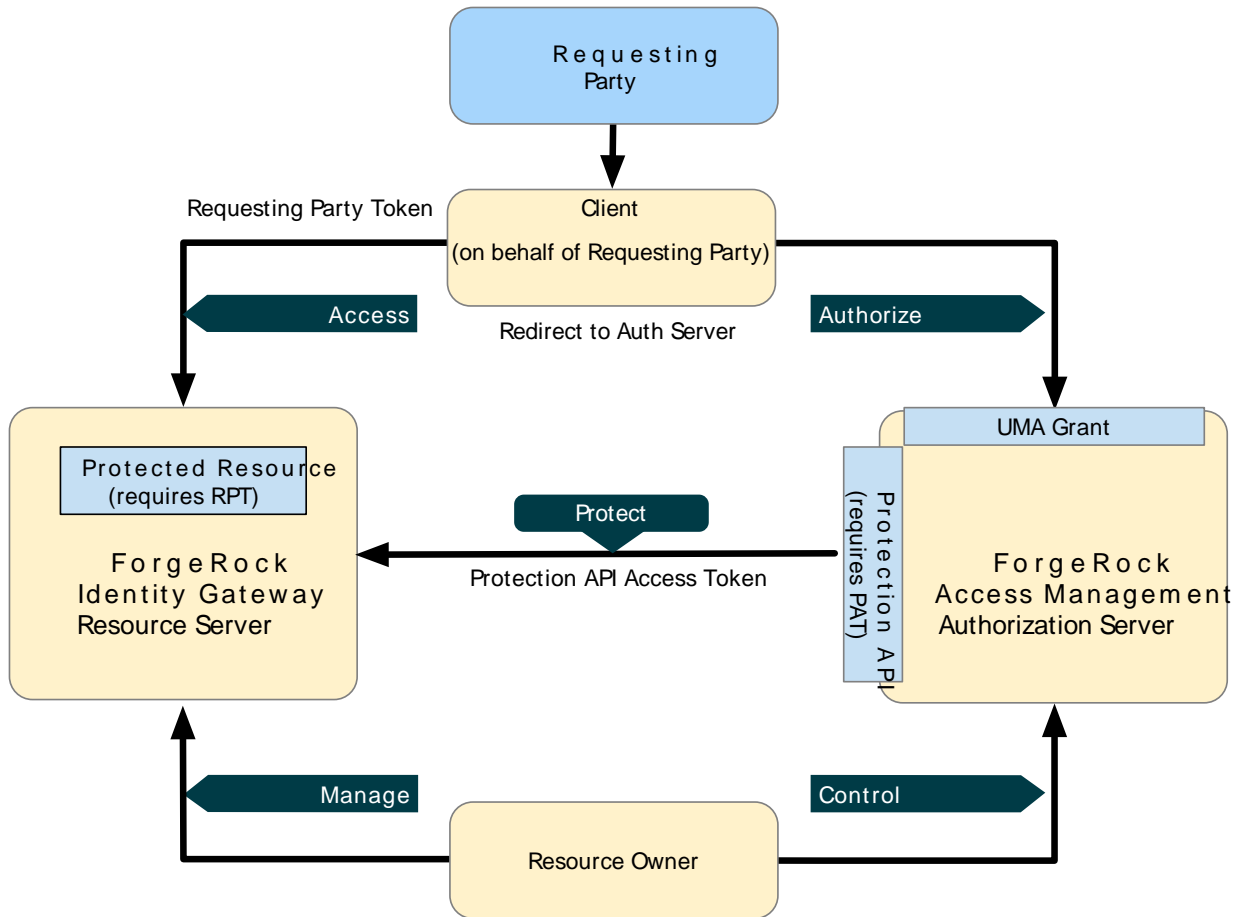
# Supporting UMA Resource Servers

IG includes support for User-Managed Access (UMA) 2.0 Grant for OAuth 2.0 Authorization specifications. This chapter describes UMA 2.0, and applies to IG 5.5 and later versions, used with AM 5.5 and later versions. For earlier versions, see their documentation.

- "About IG As an UMA Resource Server"
- "Limitations Of IG As an UMA Resource Server"
- "Setting Up the UMA Example"
- "Editing the Example to Match Custom Settings"
- "Understanding the UMA API With an API Descriptor"

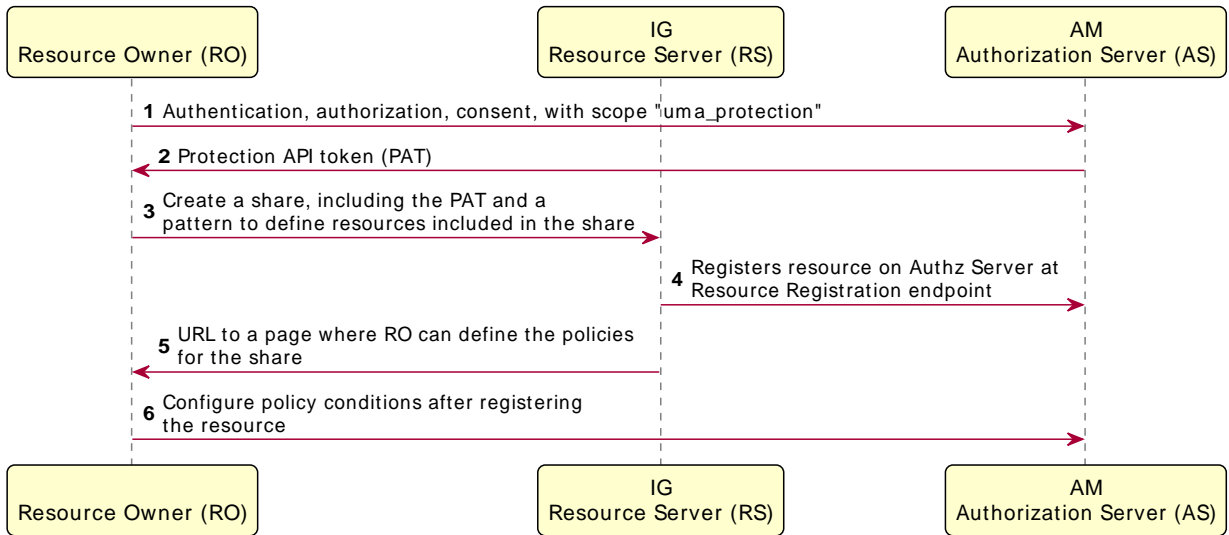
## About IG As an UMA Resource Server

The following figure shows an UMA environment, with IG protecting a resource, and AM acting as an authorization server. For information about UMA, see AM's *User-Managed Access (UMA) 2.0 Guide*.



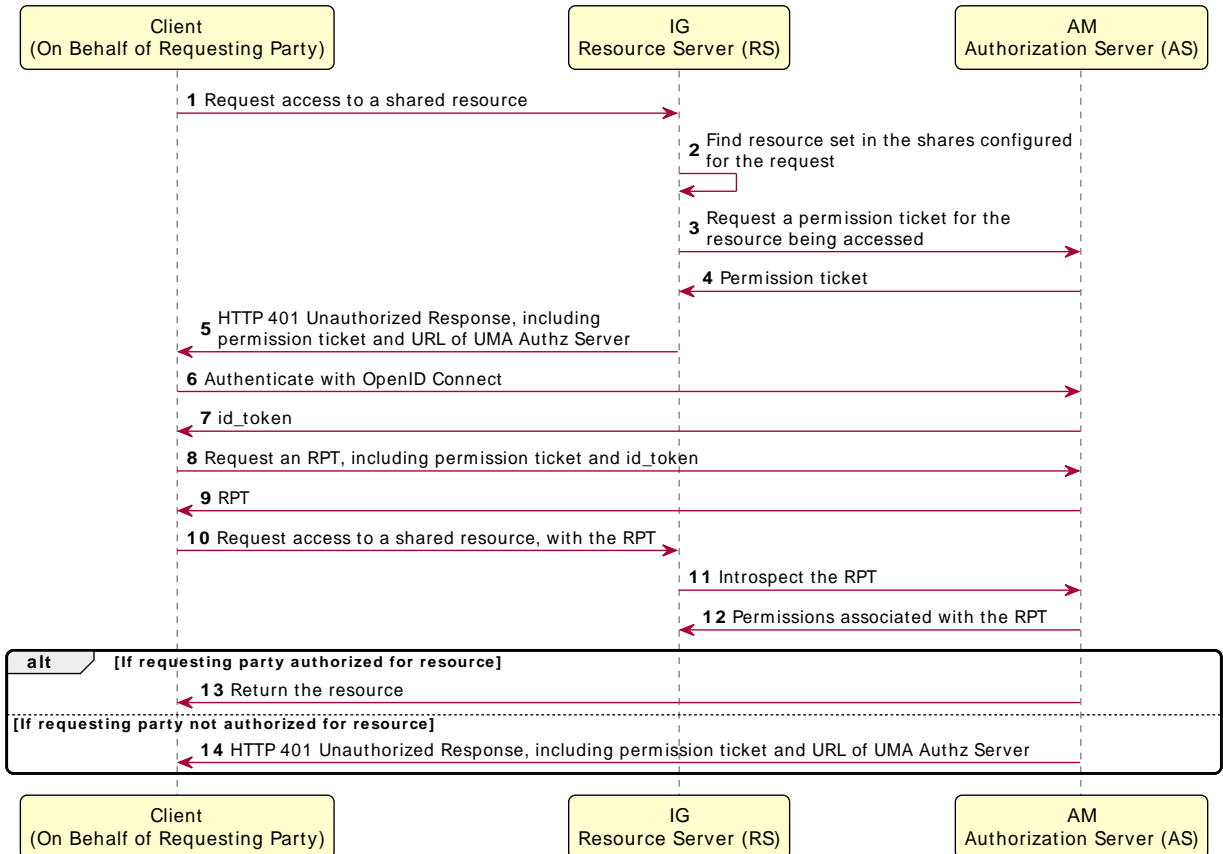
The following figure shows the data flow when the resource owner registers a resource with AM, and sets up a share using a Protection API Token (PAT):

### UMA 2.0 Flow for Protecting a Resource



The following figure shows the data flow when the client accesses the resource, using a Requesting Party Token (RPT):

### UMA 2.0 Grant Flow



For information about CORS support, see [Configuring CORS Support in AM's Security Guide](#). This procedure describes how to modify the AM configuration to allow cross-site access.

## Limitations Of IG As an UMA Resource Server

When using IG as an UMA resource server, note the following points:

- IG depends on the resource owner for the PAT.

When a PAT expires, no refresh token is available to IG. The resource owner must repeat the entire share process with a new PAT in order to authorize access to protected resources. The resource owner should delete the old resource and create a new one.

- Data about PATs and shared resources is held in memory.

IG has no mechanism for persisting the data across restarts. When IG stops and starts again, the resource owner must repeat the entire share process.

- UMA client applications for sharing and accessing protected resources must deal with UMA error conditions and IG error conditions.
- By default, the REST API to manage share objects exposed by IG is protected only by CORS.
- When matching protected resource paths with share patterns, IG takes the longest match.

For example, if resource owner Alice shares `/photos/*.*` with Bob, and `/photos/vacation.png` with Charlie, and then Bob attempts to access `/photos/vacation.png`, IG applies the sharing permissions for Charlie, not Bob. As a result, Bob can be denied access.

## Setting Up the UMA Example

This section describes tasks to set up AM as an authorization server:

- Enabling cross-origin resource sharing (CORS) support in AM
- Configuring AM as an authorization server
- Registering UMA client profiles with AM
- Setting up a resource owner (Alice) and requesting party (Bob)

### Caution

The settings in this section are suggestions for this tutorial. They are not intended as instructions for setting up AM CORS support on a server in production.

If you need to accept all origins, by allowing the use of `Access-Control-Allow-Origin=*`, do not allow `Content-Type` headers. Allowing the use of both types of headers exposes AM to cross-site request forgery (CSRF) attacks.

### *Enable CORS Support for AM*

Before you start, prepare AM, IG, and the sample application as described in "Example Installation for This Guide".

If you use different settings for the sample application, see "Editing the Example to Match Custom Settings".


## 1. Set up AM:

- a. Find the name of the AM session cookie:


```
$ curl http://openam.example.com:8088/openam/json/serverinfo/* | jq .cookieName
```

The rest of the steps in this procedure assume that you are using the default AM session cookie, `iPlanetDirectoryPro`. If not, substitute the value in the procedure.


- b. Create an OAuth 2.0 Authorization Server:

- i. Select  Services > Add a Service > OAuth2 Provider.
- ii. Add a service with the default values.

- c. Configure an UMA Authorization Server:

- i. Select  Services > Add a Service > UMA Provider.
- ii. Add a service with the default values.

- d. Add an OAuth 2.0 client for UMA protection:


- i. Select  Applications > OAuth 2.0 > Clients.
- ii. Add a client with these values:

- Client ID: `OpenIG`
- Client secret: `password`
- Scope: `uma_protection`



- iii. (From AM 6.5) On the Advanced tab, select the following option:

- Grant Types: `Resource Owner Password Credentials`

- e. Add an OAuth 2.0 client for accessing protected resources:

- i. Select  Applications > OAuth 2.0 > Clients.
- ii. Add a client with these values:

- Client ID: `UmaClient`
- Client secret: `password`
- Scope: `openid`

- iii. (From AM 6.5) On the Advanced tab, select the following option:
  - Grant Types: **Resource Owner Password Credentials** and **UMA**
- f. Select  Identities, and add an identity for a resource owner, with the following values:
  - ID: **alice**
  - Password: **UMAexample**
- g. Select  Identities, and add an identity for a requesting party, with the following values:
  - ID: **bob**
  - Password: **UMAexample**
- h. Enable the CORS filter on AM:
  - i. In a terminal window, retrieve an `access_token` from AM:

```
$ mytoken=$(curl --request POST \  
--header "Accept-API-Version: resource=2.1" \  
--header "X-OpenAM-Username: amadmin" \  
--header "X-OpenAM-Password: password" \  
--header "Content-Type: application/json" \  
--data "{}" \  
http://openam.example.com:8088/openam/json/authenticate | jq -r ".tokenId")
```

- ii. Using the token retrieved in the previous step, enable the CORS filter on AM, by using the use the `/global-config/services/CorsService` REST endpoint:

```
$ curl \  
--request PUT \  
--header "Content-Type: application/json" \  
--header "iPlanetDirectoryPro: $mytoken" http://openam.example.com:8088/openam/json/global-  
config/services/CorsService/configuration/CorsService \  
--data '{  
  "acceptedMethods": [  
    "POST",  
    "GET",  
    "PUT",  
    "DELETE",  
    "PATCH",  
    "OPTIONS"  
  ],  
  "acceptedOrigins": [  
    "http://app.example.com:8081",  
    "http://openig.example.com:8080",  
    "http://openam.example.com:8088/openam"  
  ],  
  "allowCredentials": true,  
  "acceptedHeaders": [  
    "Authorization",  
    "Content-Type",  
    "iPlanetDirectoryPro",
```

```

        "X-OpenAM-Username",
        "X-OpenAM-Password",
        "Accept",
        "Accept-Encoding",
        "Connection",
        "Content-Length",
        "Host",
        "Origin",
        "User-Agent",
        "Accept-Language",
        "Referer",
        "Dnt",
        "Accept-API-Version",
        "If-None-Match",
        "Cookie",
        "X-Requested-With",
        "Cache-Control",
        "X-Password",
        "X-Username",
        "X-NoSession"
    ],
    "exposedHeaders": [
        "Access-Control-Allow-Origin",
        "Access-Control-Allow-Credentials",
        "Set-Cookie",
        "WWW-Authenticate"
    ],
    "maxAge": 600,
    "enabled": true,
    "allowCredentials": true
  },
  {
    "_id": "CorsService",
    "_rev": "300529028",
    "maxAge": 600,
    "exposedHeaders": ["Access-Control-Allow-Origin", "Access-Control-Allow-Credentials", "WWW-Authenticate", "Set-Cookie"],
    "acceptedOrigins": ["http://openig.example.com:8080", "http://app.example.com:8081", "http://openam.example.com:8088/openam"],
    "acceptedMethods": ["DELETE", "POST", "GET", "OPTIONS", "PUT", "PATCH"],
    "acceptedHeaders": ["Cookie", "Origin", "X-Username", "Accept", "X-Requested-With", "Connection", "User-Agent", "Referer", "Host", "Dnt", "X-NoSession", "Accept-Encoding", "iPlanetDirectoryPro", "If-None-Match", "Authorization", "Cache-Control", "X-OpenAM-Username", "X-Password", "Accept-Language", "Content-Length", "X-OpenAM-Password", "Accept-API-Version", "Content-Type"],
    "enabled": true,
    "allowCredentials": true,
    "_type": {
      "_id": "CorsService",
      "name": "CORS Service",
      "collection": true
    }
  }
}

```

### Tip

To delete the CORS configuration and create another, first run the following command:



```
$ curl \
--request DELETE \
--header "X-Requested-With: XMLHttpRequest" \
--header "iplanetDirectoryPro: $mytoken" \
http://openam.example.com:8088/openam/json/global-config/services/CorsService/CorsService/
configuration/CorsService
```

## 2. Set up IG as an UMA resource server:

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

*Linux*

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

*Windows*

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

- b. Add the following route to IG:

*Linux*

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

*Windows*

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

*Standalone mode*

```

{
  "prefix": "openig",
  "connectors": [
    { "port" : 8080 }
  ],
  "heap": [
    {
      "name": "ClientHandler",
      "type": "ClientHandler"
    },
    {
      "name": "ApiProtectionFilter",
      "type": "CorsFilter",
      "config": {
        "policies": [
          {
            "origins": [ "http://app.example.com:8081" ],
            "acceptedMethods": [ "GET", "POST", "DELETE" ],
            "acceptedHeaders": [ "Content-Type" ]
          }
        ]
      }
    }
  ]
}

```

#### Web container mode

```

{
  "prefix": "openig",
  "heap": [
    {
      "name": "ClientHandler",
      "type": "ClientHandler"
    },
    {
      "name": "ApiProtectionFilter",
      "type": "CorsFilter",
      "config": {
        "policies": [
          {
            "origins": [ "http://app.example.com:8081" ],
            "acceptedMethods": [ "GET", "POST", "DELETE" ],
            "acceptedHeaders": [ "Content-Type" ]
          }
        ]
      }
    }
  ]
}

```

Notice the following feature of the route:

- The default `ApiProtectionFilter` is overridden by the `CorsFilter`, which allows requests from the origin `http://app.example.com:8081`. For information, see "`AdminHttpApplication (admin.json)`" in the *Configuration Reference*.

c. Add the following route to IG:

Linux

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

Windows

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

```
{
  "name": "00-uma",
  "condition": "${request.uri.host == 'app.example.com'}",
  "heap": [
    {
      "name": "UmaService",
      "type": "UmaService",
      "config": {
        "protectionApiHandler": "ClientHandler",
        "wellKnownEndpoint": "http://openam.example.com:8088/openam/uma/.well-known/uma2-configuration",
        "resources": [
          {
            "comment": "Protects all resources matching the following pattern.",
            "pattern": ".*",
            "actions": [
              {
                "scopes": [
                  "#read"
                ],
                "condition": "${request.method == 'GET'}"
              },
              {
                "scopes": [
                  "#create"
                ],
                "condition": "${request.method == 'POST'}"
              }
            ]
          }
        ]
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "CorsFilter",
          "config": {
            "policies": [
              {
                "origins": [ "http://app.example.com:8081" ],
                "acceptedMethods": [ "GET" ],
                "acceptedHeaders": [ "Authorization" ],
                "exposedHeaders": [ "WWW-Authenticate" ],
                "allowCredentials": true
              }
            ]
          }
        }
      ]
    }
  }
}
```

```
    }
  ]
},
{
  "type": "UmaFilter",
  "config": {
    "protectionApiHandler": "ClientHandler",
    "umaService": "UmaService"
  }
},
],
"handler": "ReverseProxyHandler"
}
}
```

Notice the following features of the route:

- The route matches requests from [app.example.com](http://app.example.com).
- The UmaService describes the resources that a resource owner can share, using AM as the authorization server. It provides a REST API to manage sharing of resource sets.
- The CorsFilter defines the policy for cross-origin requests, listing the methods and headers that the request can use, the headers that are exposed to the frontend JavaScript code, and whether the request can use credentials.
- The UmaFilter manages requesting party access to protected resources, using the UmaService. Protected resources are on the sample application, which responds to requests on port 8081.

d. Restart IG to reload the configuration.

3. Test the setup:

a. If necessary, log out of AM, and then go to <http://app.example.com:8081/uma/>.

b. Share resources:

i. Select Alice shares resources.

ii. On Alice's page, select Share with Bob. The following items are displayed:

- The PAT that Alice receives from AM.
- The metadata for the resource set that Alice registers through IG.
- The result of Alice authenticating with AM in order to create a policy.
- The successful result when Alice configures the authorization policy attached to the shared resource.

**Tip**

If the step fails, run the following command to get an access token for Alice:

```
$ curl -X POST \  
-H "Cache-Control: no-cache" \  
-H "Content-Type: application/x-www-form-urlencoded" \  
-d \  
'grant_type=password&scope=uma_protection&username=alice&password=UMAexample&client_id=OpenIG&client_id=OpenIG' \  
\ \  
http://openam.example.com:8088/openam/oauth2/access_token \  
{ "access_token": "AQI...QAA*", "scope": "uma_protection", "token_type": "Bearer", "expires_in": 3599 }
```

If you fail to get an access token, check that AM is configured as described in "Setting Up the UMA Example". 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/>.

- c. Access resources:
  - i. Go back to the first page, and select Bob accesses resources.
  - ii. On Bob's page, select Get Alice's resources. The following items are displayed:
    - The WWW-Authenticate Header.
    - The OpenID Connect Token that Bob gets to obtain the RPT.
    - The RPT that Bob gets in order to request the resource again.
    - The final response containing the body of the resource.

## 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 "*Downloading and Starting the Sample Application*" in the *Getting Started Guide* to temporary folder:

```
$ mkdir /tmp/uma
$ cd /tmp/uma
$ jar -xvf /path/to/IG-sample-application-7.0.2.jar webroot-uma
created: webroot-uma/
inflated: webroot-uma/bob.html
inflated: webroot-uma/common.js
inflated: webroot-uma/alice.html
inflated: webroot-uma/index.html
inflated: webroot-uma/style.css
```

2. Edit the configuration in `common.js`, `alice.html`, and `bob.html` to match your settings.
3. Repack the UMA sample client files and then restart the sample application:

```
$ jar -uvf /path/to/IG-sample-application-7.0.2.jar webroot-uma
adding: webroot-uma/(in = 0) (out= 0)(stored 0%)
adding: webroot-uma/bob.html(in = 26458) (out= 17273)(deflated 34%)
adding: webroot-uma/common.js(in = 3652) (out= 1071)(deflated 70%)
adding: webroot-uma/alice.html(in = 27775) (out= 17512)(deflated 36%)
adding: webroot-uma/index.html(in = 22046) (out= 16060)(deflated 27%)
adding: webroot-uma/style.css(in = 811) (out= 416)(deflated 48%)
updated module-info: module-info.class
```

4. If necessary, adjust the CORS settings for AM.

## 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 14

# Configuring Routers and Routes

The following sections provide an overview of how IG uses routers and routes to handle requests and their context:

- "Configuring Routers"
- "Configuring Routes"
- "Creating and Editing Routes Through Common REST"
- "Preventing the Reload of Routes"
- "Accessing Reserved Routes"

For information about creating routes in Studio, see the [Studio User Guide](#).

## 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 "Monitoring Endpoints" in the *Configuration Reference*. For information about the parameters of a router, see "Router" in the *Configuration Reference*.

## 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" 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" in the *Configuration Reference*.

### 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.

## 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 a characteristic of the request, context, or IG runtime environment, such as system properties or environment variables. Conditions are defined using IG expressions, as described in "Expressions" 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:

### Example Conditions and Requests

Condition	Requests That Meet the Condition
<code>"\${matches(request.uri.path, '^/login')}"</code>	<code>http://app.example.com/login, ...</code>
<code>"\${request.uri.host == 'api.example.com'}"</code>	<code>http://api.example.com/, https://api.example.com/home, http://api.example.com:8080/home, ...</code>
<code>"\${matches(contexts.client.remoteAddress, '127.0.0.1')}"</code>	<code>http://localhost:8080/keygen, http://127.0.0.1:8080/keygen, ...</code>  Where <code>/keygen</code> is not mandatory and could be anything else.

Condition	Requests That Meet the Condition
<code>"\${matches(request.uri.query, 'demo=simple')}"</code>	<a href="http://openig.example.com:8080/login?demo=simple">http://openig.example.com:8080/login?demo=simple</a> , ...  For information about URI query, see <a href="#">query</a> in "URI" in the <i>Configuration Reference</i> .
<code>"\${request.uri.scheme == 'http'}"</code>	<a href="http://openig.example.com:8080">http://openig.example.com:8080</a> , ...
<code>"\${matches(request.uri.path, '^/dispatch') or matches(request.uri.path, '^/mylogin')}"</code>	<a href="http://openig.example.com:8080/dispatch">http://openig.example.com:8080/dispatch</a> , <a href="http://openig.example.com:8080/mylogin">http://openig.example.com:8080/mylogin</a> , ...
<code>"\${request.uri.host == 'sp1.example.com' and not matches(request.uri.path, '^/saml')}"</code>	<a href="http://sp1.example.com:8080/">http://sp1.example.com:8080/</a> , <a href="http://sp1.example.com/mypath">http://sp1.example.com/mypath</a> , ...  Not <a href="http://sp1.example.com:8080/saml">http://sp1.example.com:8080/saml</a> , <a href="http://sp1.example.com/saml">http://sp1.example.com/saml</a> , ...
<code>"condition": "\${matches (request.uri.path, '&amp;{uriPath}')}"</code>	<a href="http://openig.example.com:8080/hello">http://openig.example.com:8080/hello</a> , when the following property is configured: <pre>{   "properties": {     "uriPath": "hello"   } }</pre> For information about including properties in the configuration, see " <i>Properties</i> " in the <i>Configuration Reference</i> .
<code>"condition": "\${request.headers['X-Forwarded-Host'][0] == 'service.example.com'}"</code>	Requests with the header <code>X-Forwarded-Host</code> , whose first value is <a href="#">service.example.com</a> .

## 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`.

## 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 information about switching to development mode, see "Switching from Production Mode to Development Mode" in the *Getting Started Guide*.

**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, see the Java Agent's *User 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 *Getting Started Guide*.

1. Add the following route to IG:

*Linux*

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

*Windows*

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

```
{
  "name": "crest",
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "reason": "OK",
      "headers": {
        "Content-Type": [ "text/plain" ]
      },
      "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:

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

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

## 3. To add a route through Common REST:

- 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, where `$HOME/.openig` is the instance directory. To double check, go to <http://openig.example.com:8080/crest>. You should get an HTTP 404 error.
- Enter the following command in a terminal window:

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

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

- Check in `$HOME/.openig/logs/route-system.log` that the route has been added to configuration, where `$HOME/.openig` is the instance directory. To double-check, go to <http://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:

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

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

- Check in `$HOME/.openig/logs/route-system.log` that the route has been updated, where `$HOME/.openig` is the instance directory. To double-check, go to `http://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:

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

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

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

- Enter the following command in a terminal window:

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

The list of loaded routes is displayed.

## 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"
  }
}
```

## 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 "Setting Up the UMA Example".

## Chapter 15

# Proxying WebSocket Traffic

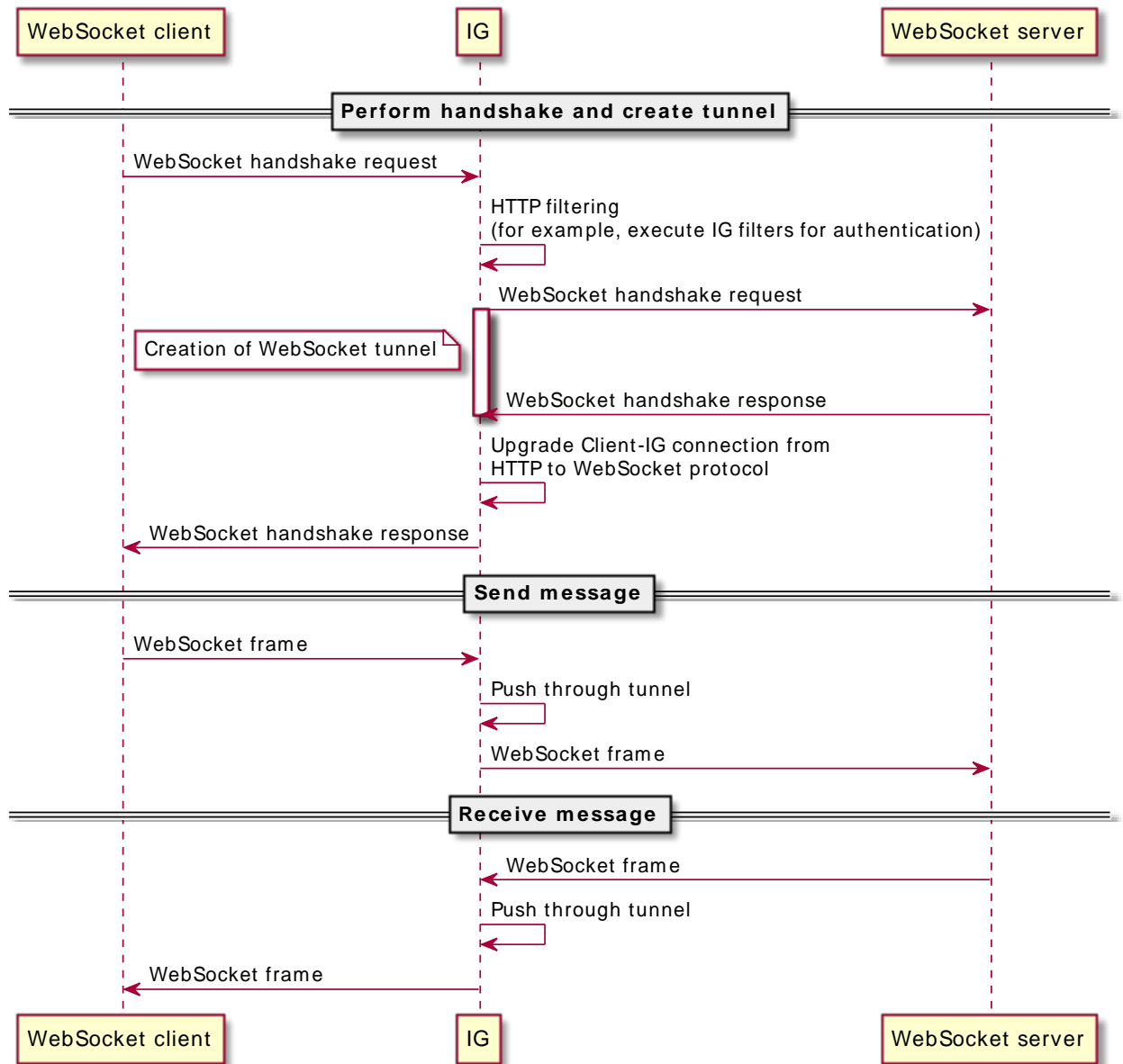
When a user agent requests an upgrade from HTTP or HTTPS to the WebSocket protocol, IG detects the request and performs an HTTP handshake request between the user agent and the protected application.

If the handshake is successful, IG upgrades the connection and provides a dedicated tunnel to route WebSocket traffic between the user agent and the protected application. IG does not intercept messages to or from the WebSocket server.

The tunnel remains open until it is closed by the user agent or protected application. When the user agent closes the tunnel, the connection between IG and the protected application is automatically closed.

The following sequence diagram shows the flow of information when IG proxies WebSocket traffic:

Flow of Information to Proxy WebSocket Traffic







To set up IG to proxy WebSocket traffic, configure the `websocket` property of `ReverseProxyHandler`. By default, IG does not proxy WebSocket traffic. For more information, see "ReverseProxyHandler" in the *Configuration Reference*.

## Configure Proxying for WebSocket Traffic

### 1. Set up AM:

- a. (For AM 6.5.x and earlier versions) Select  Identities > demo, and set the demo user password to `Ch4ng31t`.
- b. (For AM 6.5.3 and later versions) Select  Services > Add a Service, and add a Validation Service with the following Valid goto URL Resources:
  - `http://openig.example.com:8080/*`
  - `http://openig.example.com:8080/*?*`
- c. Select Applications > Agents > Identity Gateway, add an agent with the following values:
  - Agent ID: `ig_agent`
  - Password: `password`

Leave all other values as default.

For AM 6.5.x and earlier versions, set up an agent as described in "Set Up an IG Agent in AM 6.5 and Earlier".

### 2. Set up IG:

- a. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

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

- b. Add the following route to IG, to serve `.css` and other static resources for the sample application:

Linux

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

Windows

```
%appdata%\openIG\config\routes\static-resources.json
```

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

c. Add the following route to IG:

*Linux*

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

*Windows*

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

```
{
  "name": "websocket",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/websocket')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://openam.example.com:8088/openam/",
        "version": "7"
      }
    }
  ],
  {
    "name": "ReverseProxyHandler",
    "type": "ReverseProxyHandler",
    "config": {
      "websocket": {
        "enabled": true
      }
    }
  }
],
"handler": {
  "type": "Chain",
  "config": {
    "filters": [
      {
        "name": "SingleSignOnFilter-1",
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      }
    ]
  }
}
```

```
    }  
  },  
  "handler": "ReverseProxyHandler"  
}  
}
```

For information about how to set up the route in Studio, see "Proxy for WebSocket Traffic in Structured Editor" in the *Studio User Guide*.

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.

### 3. Test the setup:

- a. If you are logged in to AM, log out.
- b. Go to `http://openig.example.com:8080/websocket`.

The `SingleSignOnFilter` redirects the request to AM for authentication.

- c. Log in to AM as user `demo`, password `Ch4ng31t`.

AM authenticates the user, creates an SSO token, and redirects the request back to the original URI, with the token in a cookie.

The request then passes to the `ReverseProxyHandler`, which routes the request to the HTML page `/websocket/index.html` of the sample app. The page initiates the HTTP handshake for connecting to the WebSocket endpoint `/websocket/echo`.

- d. Enter text on the WebSocket echo screen, and note that the text is echoed back.

## Chapter 16

# Implementing Not-Enforced URIs for Authentication

By default, IG routes protect resources (such as a websites or applications) from all requests on the route's condition path. Some parts of the resource, however, do not need to be protected. For example, it can be okay for unauthenticated requests to access the welcome page of a web site, or an image or favicon.

The following sections give examples of routes that do not enforce authentication for a specific request URL or URL pattern, but enforce authentication for other request URLs:


- "Implementing Not-Enforced URIs With a SwitchFilter"
- "Implementing Not-Enforced URIs With a Dispatcher"

## Implementing Not-Enforced URIs With a SwitchFilter

### *Not Enforce URIs by Using a SwitchFilter*

Before you start:

- Prepare IG and the sample app as described in Getting Started Guide
  - Install and configure AM on `http://openam.example.com:8088/openam`, using the default configuration.
1. On your system, add the following data in a comma-separated value file called `/tmp/userfile` (on Windows `C:\Temp\userfile`):

```
username,password,fullname,email
george,C0stanza,George Costanza,george@example.com
kramer,N3wman12,Kramer,kramer@example.com
bjensen,H1falutin,Babs Jensen,bjensen@example.com
demo,Ch4ng31t,Demo User,demo@example.com
kvaughan,B5ibery12,Kirsten Vaughan,kvaughan@example.com
scarter,S9rain12,Sam Carter,scarter@example.com
```
  2. Set up AM:
    - a. (For AM 6.5.x and earlier versions) Select  Identities > demo, and set the demo user password to `Ch4ng31t`.

- b. (For AM 6.5.3 and later versions) Select  Services > Add a Service, and add a Validation Service with the following Valid goto URL Resources:

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

- c. Select Applications > Agents > Identity Gateway, add an agent with the following values:

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

Leave all other values as default.

For AM 6.5.x and earlier versions, set up an agent as described in "Set Up an IG Agent in AM 6.5 and Earlier".

### 3. Set up IG:

- a. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

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

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

*Linux*

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

*Windows*

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

- c. Add the following route to IG:

*Linux*

```
$HOME/.openig/config/routes/not-enforced-switch.json
```

*Windows*

```
%appdata%\OpenIG\config\routes\not-enforced-switch.json
```

```
{
```

```

"properties": {
  "notEnforcedPathPatterns": "^/home|^/favicon.ico|^/css"
},
"heap": [
  {
    "name": "SystemAndEnvSecretStore-1",
    "type": "SystemAndEnvSecretStore"
  },
  {
    "name": "AmService-1",
    "type": "AmService",
    "config": {
      "agent": {
        "username": "ig_agent",
        "passwordSecretId": "agent.secret.id"
      },
      "secretsProvider": "SystemAndEnvSecretStore-1",
      "url": "http://openam.example.com:8088/openam/",
      "version": "7"
    }
  }
],
"name": "not-enforced-switch",
"condition": "${matches(request.uri.path, '^/')}",
"baseURI": "http://app.example.com:8081",
"handler": {
  "type": "Chain",
  "config": {
    "filters": [
      {
        "name": "SwitchFilter-1",
        "type": "SwitchFilter",
        "config": {
          "onRequest": [{
            "condition": "${matches(request.uri.path, '&{notEnforcedPathPatterns}')}",
            "handler": "ReverseProxyHandler"
          }]
        }
      },
      {
        "type": "SingleSignOnFilter",
        "config": {
          "amService": "AmService-1"
        }
      },
      {
        "type": "PasswordReplayFilter",
        "config": {
          "loginPage": "${true}",
          "credentials": {
            "type": "FileAttributesFilter",
            "config": {
              "file": "/tmp/userfile",
              "key": "email",
              "value": "${contexts.ssoToken.info.uid}@example.com",
              "target": "${attributes.credentials}"
            }
          }
        }
      }
    ],
    "request": {

```

```
    "method": "POST",
    "uri": "http://app.example.com:8081/login",
    "form": {
      "username": [
        "${attributes.credentials.username}"
      ],
      "password": [
        "${attributes.credentials.password}"
      ]
    }
  },
},
],
"handler": "ReverseProxyHandler"
}
```

Notice the following features of the route:

- The route condition is `/`, so the route matches all requests.
- The SwitchFilter passes requests on the path `~/home`, `~/favicon.ico`, and `~/css` directly to the ReverseProxyHandler. All other requests continue the along the chain to the SingleSignOnFilter.
- If the request does not have a valid AM session cookie, the SingleSignOnFilter redirects the request to AM for authentication. The SingleSignOnFilter stores the cookie value in an `SsoTokenContext`.
- Because the PasswordReplayFilter detects that the response is a login page, it uses the FileAttributesFilter to replay the password, and logs the request into the sample application.

#### 4. Test the setup:

- a. If you are logged in to AM, log out and clear any cookies.
- b. Access the route on the not-enforced URL `http://openig.example.com:8080/home`. The home page of the sample app is displayed without authentication.
- c. Access the route on the enforced URL `http://openig.example.com:8080/profile`. The SingleSignOnFilter redirects the request to AM for authentication.
- d. Log in to AM as user `demo`, password `Ch4ng31t`. The PasswordReplayFilter replays the credentials for the demo user. The request is passed to the sample app's profile page for the demo user.

## Implementing Not-Enforced URIs With a Dispatcher

To use a Dispatcher for not-enforced URIs, replace the route in "Implementing Not-Enforced URIs With a SwitchFilter" with the following route. If the request is on the path `~/home`, `~/favicon.ico`, or `~/css`, the Dispatcher sends it directly to the ReverseProxyHandler, without authentication. It passes all other requests into the Chain for authentication.

```
{
  "properties": {
    "notEnforcedPathPatterns": "^/home|^/favicon.ico|^/css"
  },
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        },
        "secretsProvider": "SystemAndEnvSecretStore-1",
        "url": "http://openam.example.com:8088/openam/",
        "version": "7"
      }
    }
  ],
  "name": "not-enforced-dispatch",
  "condition": "${matches(request.uri.path, '^/')}",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "Dispatcher",
    "config": {
      "bindings": [
        {
          "condition": "${matches(request.uri.path, '&{notEnforcedPathPatterns}')}",
          "handler": "ReverseProxyHandler"
        },
        {
          "handler": {
            "type": "Chain",
            "config": {
              "filters": [
                {
                  "type": "SingleSignOnFilter",
                  "config": {
                    "amService": "AmService-1"
                  }
                },
                {
                  "type": "PasswordReplayFilter",
                  "config": {
                    "loginPage": "${true}",
                    "credentials": {

```



```
    "type": "FileAttributesFilter",
    "config": {
      "file": "/tmp/userfile",
      "key": "email",
      "value": "${contexts.ssoToken.info.uid}@example.com",
      "target": "${attributes.credentials}"
    },
  },
  "request": {
    "method": "POST",
    "uri": "http://app.example.com:8081/login",
    "form": {
      "username": [
        "${attributes.credentials.username}"
      ],
      "password": [
        "${attributes.credentials.password}"
      ]
    }
  }
},
],
"handler": "ReverseProxyHandler"
}
}
}
}
}
}
}
```

## Chapter 17

# Configuration Templates

This chapter contains template routes for common configurations. To use a template, set up IG as described in *Getting Started Guide*, and modify the template for your deployment. Before you use a route in production, review the points in "General Security Considerations" in the *Maintenance Guide*.

- "Proxy and Capture"
- "Simple Login Form"
- "Login Form With Cookie From Login Page"
- "Login Form With Password Replay and Cookie Filters"
- "Login Which Requires a Hidden Value From the Login Page"
- "HTTP and HTTPS Application"
- "AM Integration With Headers"

## Proxy and Capture

If you installed and configured IG with a router and default route as described in *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": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            }
          }
        }
      }
    }
  ]
}
```

```

    }
  },
  "hostnameVerifier": "ALLOW_ALL"
}
],
"handler": {
  "type": "DispatchHandler",
  "config": {
    "bindings": [
      {
        "condition": "${request.uri.path == '/login'}",
        "handler": "ReverseProxyHandler",
        "baseURI": "https://app.example.com:8444"
      },
      {
        "condition": "${request.uri.scheme == 'http'}",
        "handler": "ReverseProxyHandler",
        "baseURI": "http://app.example.com:8081"
      },
      {
        "handler": "ReverseProxyHandler",
        "baseURI": "https://app.example.com:8444"
      }
    ]
  }
},
"condition": "${matches(request.uri.query, 'demo=capture')}"
}

```

To try this example with the sample application:

1. Add the following route to IG:

*Linux*

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

*Windows*

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

2. Add the following route to serve static resources, such as .css, for the sample application:

*Linux*

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

*Windows*

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

```

{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}

```

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

Configure the `ReverseProxyHandler` to strictly verify hostnames for outgoing SSL connections.

In production, do not use `TrustAllManager` for `TrustManager`, or `ALLOW_ALL` for hostname verification. For information, see "`ReverseProxyHandler`" 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.

## 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": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            }
          }
        },
        "hostnameVerifier": "ALLOW_ALL"
      }
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
```

```

    "config": {
      "loginPage": "${request.uri.path == '/login'}",
      "request": {
        "method": "POST",
        "uri": "https://app.example.com:8444/login",
        "form": {
          "username": [
            "MY_USERNAME"
          ],
          "password": [
            "MY_PASSWORD"
          ]
        }
      }
    },
    "handler": "ReverseProxyHandler"
  },
  "condition": "${matches(request.uri.query, 'demo=simple')}"
}

```

To try this example with the sample application:

1. Add the following route to IG:

*Linux*

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

*Windows*

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

2. Replace `MY_USERNAME` with `demo`, and `MY_PASSWORD` with `Ch4ng31t`.
3. Add the following route to serve static resources, such as `.css`, for the sample application:

*Linux*

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

*Windows*

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

```

{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}

```

4. Go 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:8444
connection: Keep-Alive
user-agent: Apache-HttpClient/4.1.2 (Java/1.8.0_144)
```

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

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

Configure the `ReverseProxyHandler` to strictly verify hostnames for outgoing SSL connections.

In production, do not use `TrustAllManager` for `TrustManager`, or `ALLOW_ALL` for hostname verification. For information, see "`ReverseProxyHandler`" 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.

## 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`" 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": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            }
          }
        }
      }
    }
  ]
}
```

```

    },
    "hostnameVerifier": "ALLOW_ALL"
  }
}
],
"handler": {
  "type": "Chain",
  "config": {
    "filters": [
      {
        "type": "PasswordReplayFilter",
        "config": {
          "loginPage": "${request.uri.path == '/login'}",
          "request": {
            "method": "POST",
            "uri": "https://app.example.com:8444/login",
            "form": {
              "username": [
                "MY_USERNAME"
              ],
              "password": [
                "MY_PASSWORD"
              ]
            }
          }
        }
      },
      {
        "type": "CookieFilter"
      }
    ]
  },
  "handler": "ReverseProxyHandler"
}
},
"condition": "${matches(request.uri.query, 'demo=cookie')}"
}

```

To try this example with the sample application:

1. Add the following route to IG:

*Linux*

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

*Windows*

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

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

*Linux*

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

*Windows*

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

{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

- Go to <http://openig.example.com:8080/login?demo=cookie>.

The sample application page is displayed.

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

- Refresh your connection to <http://openig.example.com:8080/login?demo=cookie>.

Compared to the example in "Login Form With Cookie From Login Page", this example displays additional information about the session cookie:

```
Cookies  session-cookie=123...
```

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

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

Configure the `ReverseProxyHandler` to strictly verify hostnames for outgoing SSL connections.

In production, do not use `TrustAllManager` for `TrustManager`, or `ALLOW_ALL` for hostname verification. For information, see "`ReverseProxyHandler`" in the *Configuration Reference*.

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

## Login Form With Password Replay and Cookie Filters

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

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



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

The route uses a default `CookieFilter` to manage cookies. In this default configuration, cookies from the protected application are intercepted and stored in the IG session. They are not sent to the browser. For information, see "CookieFilter" 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": [
                  "Ch4ng31t"
                ],
                "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
Set-Cookie: IG_SESSIONID=24446BA29E866F840197C8E0EAD57A89; Path=/; HttpOnly
...

```

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

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

## Login Which Requires a Hidden Value From the Login Page

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

### *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": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            }
          }
        }
      }
    }
  ],
  "hostnameVerifier": "ALLOW_ALL"
}

```

```

    }
  },
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "PasswordReplayFilter",
          "config": {
            "loginPage": "${request.uri.path == '/login'}",
            "loginPageExtractions": [
              {
                "name": "hidden",
                "pattern": "loginToken\\s+value=\"(.*)\""
              }
            ]
          },
          "request": {
            "method": "POST",
            "uri": "${appBaseUri}/login",
            "form": {
              "username": [
                "MY_USERNAME"
              ],
              "password": [
                "MY_PASSWORD"
              ],
              "hiddenValue": [
                "${attributes.extracted.hidden}"
              ]
            }
          }
        }
      ]
    }
  },
  "handler": "ReverseProxyHandler"
}
},
"condition": "${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. Add the following route to IG:

*Linux*

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

*Windows*

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

2. Replace **MY\_USERNAME** with **scarter**, and **MY\_PASSWORD** with **S9rain12**.

3. Add the following route to serve static resources, such as .css, for the sample application:

Linux

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

Windows

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

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

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

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

Configure the `ReverseProxyHandler` to strictly verify hostnames for outgoing SSL connections.

In production, do not use `TrustAllManager` for `TrustManager`, or `ALLOW_ALL` for hostname verification. For information, see "`ReverseProxyHandler`" 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.

## 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": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
```

```

        "trustManager": {
            "type": "TrustAllManager"
        }
    },
    "hostnameVerifier": "ALLOW_ALL"
}
],
"handler": {
    "type": "DispatchHandler",
    "config": {
        "bindings": [
            {
                "condition": "${request.uri.scheme == 'http'}",
                "handler": "ReverseProxyHandler",
                "baseURI": "http://app.example.com:8081"
            },
            {
                "condition": "${request.uri.path == '/login'}",
                "handler": {
                    "type": "Chain",
                    "config": {
                        "comment": "Add one or more filters to handle login.",
                        "filters": [],
                        "handler": "ReverseProxyHandler"
                    }
                },
                "baseURI": "https://app.example.com:8444"
            },
            {
                "handler": "ReverseProxyHandler",
                "baseURI": "https://app.example.com:8444"
            }
        ]
    }
},
"condition": "${matches(request.uri.query, 'demo=https')}"
}

```

To try this example with the sample application:

1. Add the following route to IG:

*Linux*

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

*Windows*

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

2. Add the following route to serve static resources, such as .css, for the sample application:

*Linux*

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

*Windows*

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

```
{
  "name" : "sampleapp_resources",
  "baseURI" : "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/css')}",
  "handler": "ReverseProxyHandler"
}
```

3. Go to `http://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 `ReverseProxyHandler` with one that is configured to trust the application's public key server certificate. Otherwise, use a `ReverseProxyHandler` that references a truststore holding the certificate.

Configure the `ReverseProxyHandler` to strictly verify hostnames for outgoing SSL connections.

In production, do not use `TrustAllManager` for `TrustManager`, or `ALLOW_ALL` for hostname verification. For information, see "`ReverseProxyHandler`" 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.

## 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": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": {
              "type": "TrustAllManager"
            }
          }
        }
      }
    }
  ]
}
```

```

    }
    },
    "hostnameVerifier": "ALLOW_ALL"
  }
}
],
"handler": {
  "type": "Chain",
  "config": {
    "filters": [
      {
        "type": "PasswordReplayFilter",
        "config": {
          "loginPage": "${request.uri.path == '/login'}",
          "request": {
            "method": "POST",
            "uri": "https://app.example.com:8444/login",
            "form": {
              "username": [
                "${request.headers['username']}[0]}"
              ],
              "password": [
                "${request.headers['password']}[0]}"
              ]
            }
          }
        }
      ]
    }
  },
  "handler": "ReverseProxyHandler"
}
},
"condition": "${matches(request.uri.query, 'demo=headers')}"
}

```

To try this example with the sample application:

1. Add the route to IG:

*Linux*

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

*Windows*

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

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

```

$ curl \
--header "username: kvaughan" \
--header "password: B5ibery12" \
http://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 `ReverseProxyHandler` with one that is configured to trust the application's public key server certificate. Otherwise, use a `ReverseProxyHandler` that references a truststore holding the certificate.

Configure the `ReverseProxyHandler` to strictly verify hostnames for outgoing SSL connections.

In production, do not use `TrustAllManager` for `TrustManager`, or `ALLOW_ALL` for hostname verification. For information, see "`ReverseProxyHandler`" 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 18

# Extending IG

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

- "Extending IG Through Scripts"
- "Extending IG Through the Java API"
- "Recording Custom Audit Events"

## Extending IG Through Scripts

The following sections describe how to extend IG through scripts:

- "About Scripting"
- "Scripting Dispatch"
- "Scripting HTTP Basic Authentication"
- "Scripting Authentication to LDAP-Enabled Servers"
- "Scripting SQL Queries"

## About Scripting

### 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.

IG supports the Groovy dynamic scripting language through the use of the scriptable objects. For information about scriptable object types, their configuration, and properties, see "Scripts" in the *Configuration Reference*.

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 [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" in the *Configuration Reference*.

## Using a Reference File Script

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

Linux

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

Windows

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

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

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

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

## Scripting in Studio

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

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

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

For an example of creating a `ScriptableThrottlingPolicy` in Studio, see "Configuring Scriptable Throttling". For information about using Studio, see "Adding Configuration to a Route" in the *Studio User Guide*.

## Scripting Dispatch

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

1. Add the following script to IG:

Linux

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

Windows

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

```
/*
 * Copyright 2014-2020 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:H1falutin, 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
```

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

Linux

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

Windows

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

```
{
  "heap": [
    {
      "name": "DispatchHandler",
      "type": "DispatchHandler",
      "config": {
        "bindings": [{
          "condition": "${matches(request.uri.path, '/mylogin')}",
          "handler": {
            "type": "Chain",
            "config": {
              "filters": [
                {
                  "type": "HeaderFilter",
                  "config": {
                    "messageType": "REQUEST",
                    "add": {
                      "Username": [
                        "bjensen"
                      ]
                    }
                  }
                }
              ]
            }
          }
        }
      ]
    }
  ]
}
```

```

        "Password": [
            "H1falutin"
        ]
    }
}
],
"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')}"
}

```

3. Go to <http://openig.example.com:8080/dispatch>, and click **log in**.

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

## 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.

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

Linux

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

Windows

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

```
/*
 * Copyright 2014-2020 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": "Hlfalutin"
*     }
*   }
* }
*/

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)

```

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

#### Linux

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

#### Windows

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

```
{
  "handler": {
```

```

"type": "Chain",
"config": {
  "filters": [
    {
      "type": "ScriptableFilter",
      "config": {
        "type": "application/x-groovy",
        "file": "BasicAuthFilter.groovy",
        "args": {
          "username": "bjensen",
          "password": "H1falutin"
        }
      },
      "capture": "filtered_request"
    }
  ],
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "reason": "OK",
      "headers": {
        "Content-Type": [ "text/plain" ]
      },
      "entity": "Hello bjensen!"
    }
  }
},
"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.

3. Go 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://app.example.com:8081/basic HTTP/1.1
...
Authorization: Basic Ymp...aW4=

```

## Scripting Authentication to LDAP-Enabled Servers

Many organizations use an LDAP directory service, such as ForgeRock Directory Services (DS), to store user profiles and authentication credentials. This section describes how to authenticate to DS by using a script and a ScriptableFilter.

DS is secure by default, so connections between IG and DS must be configured for TLS. For convenience, this example uses a TrustAllManager to blindly accept any certificate presented by DS.

In a production environment, use a TrustManager that is configured to accept *only* the appropriate certificates.

If the LDAP connection in your deployment is not secured with TLS, you can remove SSL options from the example script, and remove the TrustAllManager from the example route.

For more information about attributes and types for interacting with LDAP, see `AttributeParser` in DS's Javadoc. The `ConnectionFactory` heartbeat is enabled by default. For information about how to disable it, see `LdapConnectionFactory` in DS's Javadoc.

## Authenticate to an LDAP Server

1. Install an LDAP directory server, such as DS, and then generate or import some sample users who can authenticate over LDAP. For information about setting up DS and importing sample data, see [Install DS for Evaluation](#) in DS's *Installation Guide*.
2. Add the following script to IG:

### Linux

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

### Windows

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

```
import org.forgerock.opendj.ldap.*
import org.forgerock.opendj.security.SslOptions;

/* 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 = ldapHost ?: "localhost"
port = ldapPort ?: 1636

// Include options for SSL.
// In this example, the keyManager is not set (no mTLS enabled), and both
// the trustManager and the LDAP secure protocol are specified from the
// script arguments (see 'trustManager' and 'protocols' arguments).
// In a development environment (when there is no TLS), the SslOptions can be removed completely.
ldapOptions = ldap.defaultOptions(context)
SslOptions sslOptions = SslOptions.newSslOptions(null, trustManager)
    .enabledProtocols(protocols);
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

```

Information about the script is given in the script comments. If necessary, adjust the script to match your DS installation.

### 3. Add the following route to IG:

### Linux

```
$HOME/.openig/config/routes/l0-ldap.json
```

### Windows

```
%appdata%\OpenIG\config\routes\l0-ldap.json
```

```
{
  "heap": [
    {
      "name": "DsTrustManager",
      "type": "TrustAllManager"
    }
  ],
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "ScriptableFilter",
          "config": {
            "args": {
              "ldapHost": "localhost",
              "ldapPort": 1636,
              "protocols": "TLSv1.3",
              "trustManager": "${heap['DsTrustManager']}"
            },
            "type": "application/x-groovy",
            "file": "LdapsAuthFilter.groovy"
          }
        }
      ],
      "handler": {
        "type": "ScriptableHandler",
        "config": {
          "type": "application/x-groovy",
          "source": [
            "dn = request.headers['Ldap-User-Dn'].values[0]",
            "entity = '<html><body><p>Ldap-User-Dn: ' + dn + '</p></body></html>'",
            "",
            "response = new Response(Status.OK)",
            "response.entity = entity",
            "return response"
          ]
        }
      }
    }
  },
  "condition": "${matches(request.uri.path, '^/ldap')}"
}
```

Notice the following features of the route:

- The route matches requests to `/ldap`.

- The ScriptableFilter calls `LdapsAuthFilter.groovy` to authenticate the user over a secure LDAP connection, using the username and password provided in the request.
  - The script uses TrustAllManager to blindly accept any certificate presented by DS.
  - The script receives a connection to the DS server, using TLS options. Using the credentials in the request, the script tries to perform an LDAP bind operation. If the bind succeeds (the credentials are accepted by the LDAP server), the request continues to the ScriptableHandler. Otherwise, the request stops with an error.
  - The ScriptableHandler returns the user DN.
4. Go to `http://openig.example.com:8080/ldap?username=abarnes&password=chevron` to specify credentials for the sample user `abarnes`.

The script returns the user DN:

```
Ldap-User-Dn: uid=abarnes,ou=People,dc=example,dc=com
```

## Scripting SQL Queries

This example builds on "Logging In With Credentials From a Database" to use scripts to look up credentials in a database, set the credentials in headers, and set the scheme in HTTPS to protect the request.

1. Set up and test the example in "Logging In With Credentials From a Database".
2. Add the following script to IG, to look up user credentials in the database, by email address, and set the credentials in the request headers for the next handler:

*Linux*

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

*Windows*

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

```

/*
 * Copyright 2014-2020 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(dataSource)
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)

```

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

#### Linux

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

#### Windows

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

```

/*
 * Copyright 2014-2020 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 {

    // DataSource supplied as constructor parameter.
    def sql

```

```

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

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

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

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

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

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

```

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

Linux

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

Windows

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

```
{
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "JdbcDataSource-1",
      "type": "JdbcDataSource",

```

```

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

```

Notice the following features of the route:

- The route matches requests to `/db`.
- The `JdbcDataSource` in the heap sets up the connection to the database.
- The `ScriptableFilter` calls `SqlAccessFilter.groovy` to look up credentials over SQL.

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

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

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

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

The sample application profile page for George is displayed.

## Extending IG Through the Java API

### 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.

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" in the *Configuration Reference*. The following sections describe how to extend IG through the Java API:

- "Key Extension Points"
- "Implementing a Customized Sample Filter"
- "Implementing a Class Alias Resolver"
- "Configuring the Heap Object for the Customization"
- "Embedding the Customization in IG"

### Key Extension Points

Interface Stability: Evolving, as defined in "ForgeRock Product Stability Labels" 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. Add custom libraries, as described in "Embedding the Customization in IG".

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.

## 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:

```
package org.forgerock.openig.doc.examples;

import org.forgerock.http.Filter;
import org.forgerock.http.Handler;
import org.forgerock.http.protocol.Request;
import org.forgerock.http.protocol.Response;
import org.forgerock.openig.heap.GenericHeaplet;
import org.forgerock.openig.heap.HeapException;
import org.forgerock.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.examples.SampleFilter",
  "config": {
    "name": "X-Greeting",
    "value": "Hello world"
  }
}
```

Note how `type` is configured with the fully qualified class name for `SampleFilter`. To simplify the configuration, implement a class alias resolver, as described in "Implementing a Class Alias Resolver".

## 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.examples;

import java.util.HashMap;
import java.util.Map;
```

```
import org.forgerock.openig.alias.ClassAliasResolver;

/**
 * Allow use of short name aliases in configuration object types.
 *
 * This allows a configuration with {@code "type": "SampleFilter"}
 * instead of {@code "type": "org.forgerock.openig.doc.examples.SampleFilter"}.
 */
public class SampleClassAliasResolver implements ClassAliasResolver {

    private static final Map<String, Class<?>> ALIASES =
        new HashMap<>();

    static {
        ALIASES.put("SampleFilter", SampleFilter.class);
    }

    /**
     * Get the class for a short name alias.
     *
     * @param alias Short name alias.
     * @return The class, or null if the alias is not defined.
     */
    @Override
    public Class<?> resolve(final String alias) {
        return ALIASES.get(alias);
    }
}
```

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:

```
{
  "name": "SampleFilter",
  "type": "SampleFilter",
  "config": {
    "name": "X-Greeting",
    "value": "Hello world"
  }
}
```

To create a customized `ClassAliasResolver`, add a services file with the following characteristics:

- Name the file after the class resolver interface.
- Store the file under `META-INF/services/org.forgerock.openig.alias.ClassAliasResolver`, in the customization .jar file.

If you build your project using Maven, you can add the file under the `src/main/resources` directory.

- In your `ClassAliasResolver` file, add a line for the fully qualified class name of your resolver as follows:

```
org.forgerock.openig.doc.examples.SampleClassAliasResolver
```

If you have more than one resolver in your .jar file, add one line for each fully qualified class name.

## 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.

## Embedding the Customization in IG

After building your customizations into a `.jar` file, add it to the configuration as follows:

- For IG installed in standalone mode, create the directory `$HOME/.openig/extra`, where `$HOME/.openig` is the instance directory, and add the `.jar` file to the directory.
- For IG installed in web container mode, include the `.jar` file in the IG `.war` file, as follows:
  1. Unpack `IG-7.0.2.war`
  2. Include the `.jar` library in `WEB-INF/lib`
  3. Create a new `.war` file

The following example adds the `.jar` file `sample-filter` to `custom.war`:

```
$ mkdir root && cd root
$ jar -xf ~/Downloads/IG-7.0.2.war
$ cp ~/Documents/sample-filter/target/sample-filter-1.0.0-SNAPSHOT.jar WEB-INF/lib
$ jar -cf ../custom.war *
```

Deploy `custom.war` in the same way as you deploy `IG-7.0.2.war`.

## Recording Custom Audit Events

This section describes how to record a custom audit event to standard output. The example is based on the example in "Validating Access\_Tokens Through the Introspection Endpoint", adding an audit event for the custom topic `OAuth2AccessTopic`.

To record custom audit events to other outputs, adapt the route in the following procedure to use another audit event handler.

For information about how to configure supported audit event handlers, and exclude sensitive data from log files, see "Auditing Your Deployment" in the *Maintenance Guide*. For more information about audit event handlers, see "Audit Framework" in the *Configuration Reference*.

## Record Custom Audit Events to Standard Output

Before you start, prepare IG and the sample application as described in [Getting Started Guide](#).

1. Set up AM as described in ["Validating Access\\_Tokens Through the Introspection Endpoint"](#).
2. Define the schema of an event topic called `0Auth2AccessTopic` by adding the following route to IG:

Linux

```
$HOME/.openig/audit-schemas/0Auth2AccessTopic.json
```

Windows

```
%appdata%\OpenIG\audit-schemas/0Auth2AccessTopic.json
```

```
{
  "schema": {
    "$schema": "http://json-schema.org/draft-04/schema#",
    "id": "0Auth2Access",
    "type": "object",
    "properties": {
      "_id": {
        "type": "string"
      },
      "timestamp": {
        "type": "string"
      },
      "transactionId": {
        "type": "string"
      },
      "eventName": {
        "type": "string"
      },
      "accessToken": {
        "type": "object",
        "properties": {
          "scopes": {
            "type": "array",
            "items": {
              "type": "string"
            }
          },
          "expiresAt": "number",
          "sub": "string"
        },
        "required": [ "scopes" ]
      },
      "resource": {
        "type": "object",
        "properties": {
          "path": {
            "type": "string"
          },
          "method": {
            "type": "string"
          }
        }
      }
    }
  }
}
```

```

    }
  }
},
"filterPolicies": {
  "field": {
    "includeIf": [
      "_id",
      "/timestamp",
      "/eventName",
      "/transactionId",
      "/accessToken",
      "/resource"
    ]
  }
},
"required": [ "_id", "timestamp", "transactionId", "eventName" ]
}

```

Notice that the schema includes the following fields:

- Mandatory fields `_id`, `timestamp`, `transactionId`, and `eventName`.
  - `accessToken`, to include the `access_token` scopes, expiry time, and the subject.
  - `resource`, to include the path and method.
  - `filterPolicies`, to specify additional event fields to include in the logs.
3. Define a script to generate audit events on the topic named `OAuth2AccessTopic`, by adding the following file to the IG configuration as:

Linux

```
$HOME/.openig/scripts/groovy/OAuth2Access.groovy
```

Windows

```
%appdata%\OpenIG\scripts\groovy\OAuth2Access.groovy
```

```

import static org.forgerock.json.resource.Requests.newCreateRequest;
import static org.forgerock.json.resource.ResourcePath.resourcePath;

// Helper functions
def String transactionId() {
    return contexts.transactionId.transactionId.value;
}

def JsonValue auditEvent(String eventName) {
    return json(object(field('eventName', eventName),
        field('transactionId', transactionId()),
        field('timestamp', clock.instant().toEpochMilli())));
}

def auditEventRequest(String topicName, JsonValue auditEvent) {
    return newCreateRequest(resourcePath("/") + topicName, auditEvent);
}

```

```

def accessTokenInfo() {
  def accessTokenInfo = contexts.oauth2.accessToken;
  return object(field('scopes', accessTokenInfo.scopes as List),
    field('expiresAt', accessTokenInfo.expiresAt),
    field('sub', accessTokenInfo.info.sub));
}

def resourceEvent() {
  return object(field('path', request.uri.path),
    field('method', request.method));
}

// -----

// Build the event
JsonValue auditEvent = auditEvent('OAuth2AccessEvent')
  .add('accessToken', accessTokenInfo())
  .add('resource', resourceEvent());

// Send the event
auditService.handleCreate(context, auditEventRequest("OAuth2AccessTopic", auditEvent));

// Continue onto the next filter
return next.handle(context, request)

```

The script generates audit events named `OAuth2AccessEvent`, on a topic named `OAuth2AccessTopic`. The events conform to the topic schema.

4. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

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

5. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/30-custom.json
```

Windows

```
%appdata%\OpenIG\config\routes\30-custom.json
```

```

{
  "name": "30-custom",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/rs-introspect-audit')}",
  "heap": [
    {
      "name": "AuditService-1",
      "type": "AuditService",
      "config": {
        "config": {},
        "eventHandlers": [
          {
            "class": "org.forgerock.audit.handlers.json.stdout.JsonStdoutAuditEventHandler",

```



```

        "config": {
            "name": "jsonstdout",
            "elasticsearchCompatible": false,
            "topics": [
                "0Auth2AccessTopic"
            ]
        }
    },
    {
        "name": "SystemAndEnvSecretStore-1",
        "type": "SystemAndEnvSecretStore"
    },
    {
        "name": "AmService-1",
        "type": "AmService",
        "config": {
            "agent": {
                "username": "ig_agent",
                "passwordSecretId": "agent.secret.id"
            },
            "secretsProvider": "SystemAndEnvSecretStore-1",
            "url": "http://openam.example.com:8088/openam/",
            "version": "7"
        }
    }
],
"handler": {
    "type": "Chain",
    "config": {
        "filters": [
            {
                "name": "0Auth2ResourceServerFilter-1",
                "type": "0Auth2ResourceServerFilter",
                "config": {
                    "scopes": [
                        "mail",
                        "employeenumber"
                    ],
                    "requireHttps": false,
                    "realm": "OpenIG",
                    "accessTokenResolver": {
                        "name": "token-resolver-1",
                        "type": "TokenIntrospectionAccessTokenResolver",
                        "config": {
                            "amService": "AmService-1",
                            "providerHandler": {
                                "type": "Chain",
                                "config": {
                                    "filters": [
                                        {
                                            "type": "HttpBasicAuthenticationClientFilter",
                                            "config": {
                                                "username": "ig_agent",
                                                "passwordSecretId": "agent.secret.id",
                                                "secretsProvider": "SystemAndEnvSecretStore-1"
                                            }
                                        }
                                    ]
                                }
                            }
                        }
                    }
                }
            }
        ]
    }
}

```

```

    }
    ],
    "handler": "ForgeRockClientHandler"
  }
}
}
},
{
  "type": "ScriptableFilter",
  "config": {
    "type": "application/x-groovy",
    "file": "0Auth2Access.groovy",
    "args": {
      "auditService": "${heap['AuditService-1']}",
      "clock": "${heap['Clock']}"
    }
  }
},
{
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/html" ]
      },
      "entity": "<html><body><h2>Decoded access_token: ${contexts.oauth2.accessToken.info}</h2></body></html>"
    }
  }
}
}
}
}
}

```

Notice the following features of the route:

- The route matches requests to `/rs-introspect-audit`.
- The `accessTokenResolver` uses the token introspection endpoint to validate the `access_token`.
- The `HttpBasicAuthenticationClientFilter` adds the credentials to the outgoing token introspection request.
- The `ScriptableFilter` uses the Groovy script `0Auth2Access.groovy` to generate audit events named `0Auth2AccessEvent`, with a topic named `0Auth2AccessTopic`.
- The audit service publishes the custom audit event to the `JsonStdoutAuditEventHandler`. A single line per audit event is published to standard output.

## Test the Setup

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

```
$ mytoken=$(curl -s \  
--user "client-application:password" \  
--data "grant_type=password&username=george&password=C0stanza&scope=mail%20employeenumber" \  
http://openam.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
```

2. Access the route, with the `access_token` returned in the previous step:

```
$ curl -v http://openig.example.com:8080/rs-introspect-audit --header "Authorization: Bearer  
${mytoken}"
```

Information about the decoded `access_token` is returned.

3. Search the standard output for an audit message like the following example, that includes an audit event on the topic `OAuth2AccessTopic`:

```
{  
  "_id": "fa2...-14",  
  "timestamp": 155...541,  
  "eventName": "OAuth2AccessEvent",  
  "transactionId": "fa2...-13",  
  "accessToken": {  
    "scopes": ["employeenumber", "mail"],  
    "expiresAt": 155...000,  
    "sub": "george"  
  },  
  "resource": {  
    "path": "/rs-introspect-audit",  
    "method": "GET"  
  },  
  "source": "audit",  
  "topic": "OAuth2AccessTopic",  
  "level": "INFO"  
}
```

## Chapter 19

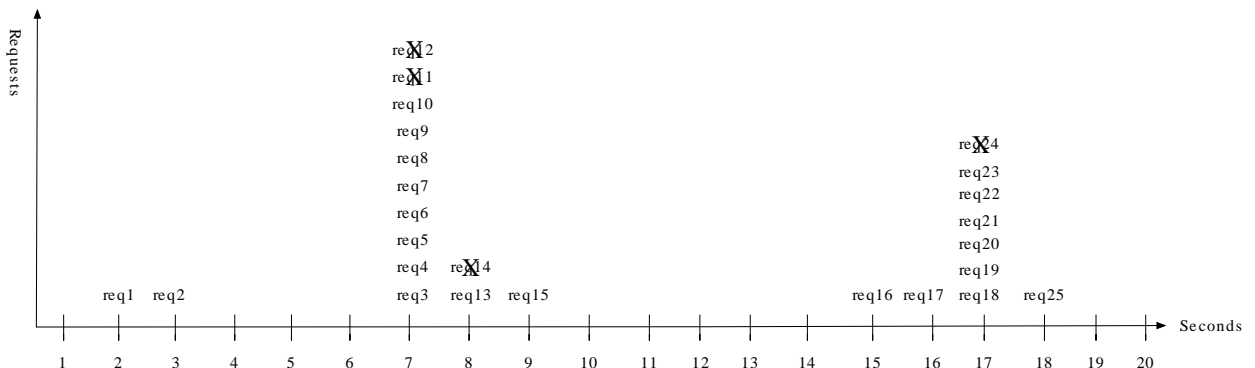
# Throttling the Rate of Requests to Protected Applications

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 following sections describe how to set up simple, mapped, and scriptable throttling filters:

- "About Throttling"
- "Configuring Simple Throttling"
- "Configuring Mapped Throttling"
- "Configuring Scriptable Throttling"

## About Throttling

The throttling filter uses the token bucket algorithm, allowing some unevenness or bursts in the request flow. The following image shows how IG manages requests for a throttling rate of 10 requests/10 seconds:



- At 7 seconds, 2 requests have previously passed when there is a burst of 9 requests. IG allows 8 requests, but disregards the 9th because the throttling rate for the 10-second throttling period has been reached.

- At 8 and 9 seconds, although 10 requests have already passed in the 10-second throttling period, IG allows 1 request each second.
- At 17 seconds, 4 requests have passed in the previous 10-second throttling period, and IG allows another burst of 6 requests.

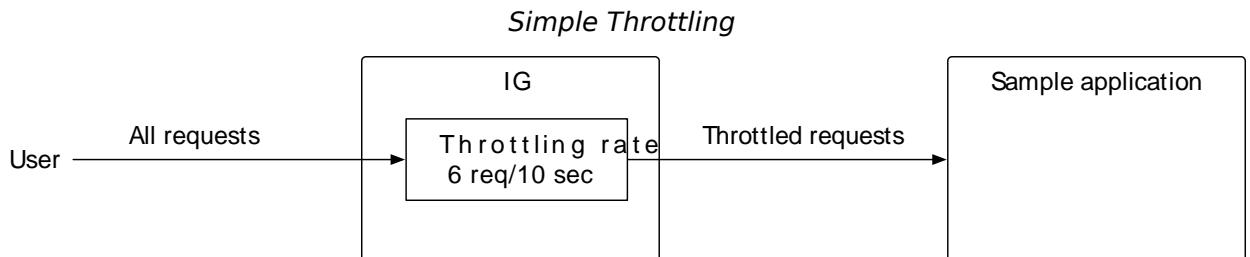
When the throttling rate is reached, IG issues an HTTP status code 429 `Too Many Requests` and a `Retry-After` header like the following, where the value is the number of seconds to wait before trying the request again:

```
GET http://openig.example.com:8080/home/throttle-scriptable HTTP/1.1
. . .

HTTP/1.1 429 Too Many Requests
Retry-After: 10
```

## Configuring Simple Throttling

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.



### Configure Simple Throttling

1. Add the following route to IG:

*Linux*

```
$HOME/.openig/config/routes/00-throttle-simple.json
```

*Windows*

```
%appdata%\OpenIG\config\routes\00-throttle-simple.json
```

```
{
  "name": "00-throttle-simple",
  "baseURI": "http://app.example.com:8081",
  "condition": "${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"
  }
}
```

For information about how to set up the IG route in Studio, see "Simple Throttling Filter in Structured Editor" in the *Studio User Guide*.

Notice the following features of the route:

- The route matches requests to `/home/throttle-simple`.
- The `ThrottlingFilter` contains a request grouping policy that is blank. This means that all requests are in the same group.
- The rate defines the number of requests allowed to access the sample application in a given time.

## 2. Test the setup:

- a. 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
```

- b. 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.

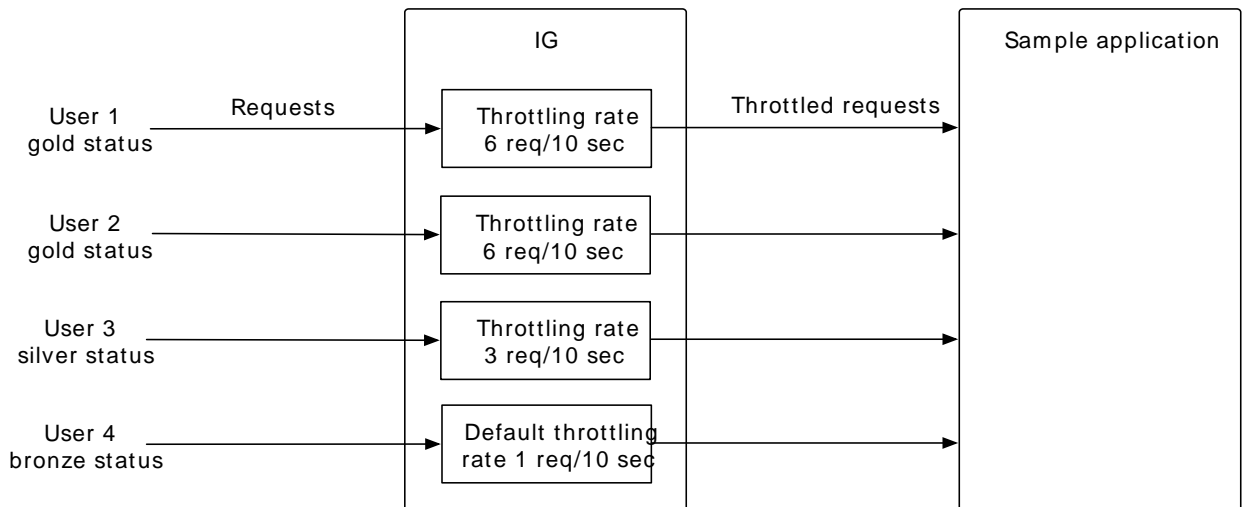
## Configuring Mapped Throttling

This section describes how to configure a mapped throttling policy, where the grouping policy defines criteria to group requests, and the rate policy defines the criteria by which rates are mapped.

The following image illustrates how different throttling rates can be applied to users.

The following image illustrates how each user with a **gold** status has a throttling rate of 6 requests/10 seconds, and each user with a **silver** status has 3 requests/10 seconds. The **bronze** status is not mapped to a throttling rate, and so a user with the **bronze** status has the default rate.

*Mapped Throttling*



### Configure Mapped Throttling

1. Set up AM:
  - a. Set up AM as described in "Validating Access\_Tokens Through the Introspection Endpoint".

- b. Select </> Scripts > OAuth2 Access Token Modification Script, and replace the default script as follows:

```
import org.forgerock.http.protocol.Request
import org.forgerock.http.protocol.Response

def attributes = identity.getAttributes(["mail", "employeeNumber"].toSet())
accessToken.setField("mail", attributes["mail"][0])
def mail = attributes['mail'][0]
if (mail.endsWith('@example.com')) {
    status = "gold"
} else if (mail.endsWith('@other.com')) {
    status = "silver"
} else {
    status = "bronze"
}
accessToken.setField("status", status)
```

The AM script adds user profile information to the `access_token`, and defines the content of the users `status` field according to the email domain.

## 2. Set up IG:

- a. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcmQ='
```

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

- b. Add the following route to IG:

*Linux*

```
$HOME/.openig/config/routes/00-throttle-mapped.json
```

*Windows*

```
%appdata%\OpenIG\config\routes\00-throttle-mapped.json
```

```
{
  "name": "00-throttle-mapped",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/throttle-mapped')}",
  "heap": [
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    },
    {
      "name": "AmService-1",
      "type": "AmService",
      "config": {
        "agent": {
          "username": "ig_agent",
          "passwordSecretId": "agent.secret.id"
        }
      }
    }
  ]
}
```



```

    },
    "secretsProvider": "SystemAndEnvSecretStore-1",
    "url": "http://openam.example.com:8088/openam/",
    "version": "7"
  }
}
],
"handler": {
  "type": "Chain",
  "config": {
    "filters": [
      {
        "name": "OAuth2ResourceServerFilter-1",
        "type": "OAuth2ResourceServerFilter",
        "config": {
          "scopes": [
            "mail",
            "employeenumber"
          ],
          "requireHttps": false,
          "realm": "OpenIG",
          "accessTokenResolver": {
            "name": "token-resolver-1",
            "type": "TokenIntrospectionAccessTokenResolver",
            "config": {
              "amService": "AmService-1",
              "providerHandler": {
                "type": "Chain",
                "config": {
                  "filters": [
                    {
                      "type": "HttpBasicAuthenticationClientFilter",
                      "config": {
                        "username": "ig_agent",
                        "passwordSecretId": "agent.secret.id",
                        "secretsProvider": "SystemAndEnvSecretStore-1"
                      }
                    }
                  ],
                  "handler": "ForgeRockClientHandler"
                }
              }
            }
          }
        }
      }
    ]
  }
},
{
  "name": "ThrottlingFilter-1",
  "type": "ThrottlingFilter",
  "config": {
    "requestGroupingPolicy": "${contexts.oauth2.accessToken.info.mail}",
    "throttlingRatePolicy": {
      "name": "MappedPolicy",
      "type": "MappedThrottlingPolicy",
      "config": {
        "throttlingRateMapper": "${contexts.oauth2.accessToken.info.status}",
        "throttlingRatesMapping": {
          "gold": {
            "numberOfRequests": 6,

```



c. Search the output file to see the result:

```
$ grep "< HTTP/1.1" /tmp/george.txt | sort | uniq -c
6 < HTTP/1.1 200
4 < HTTP/1.1 429
```

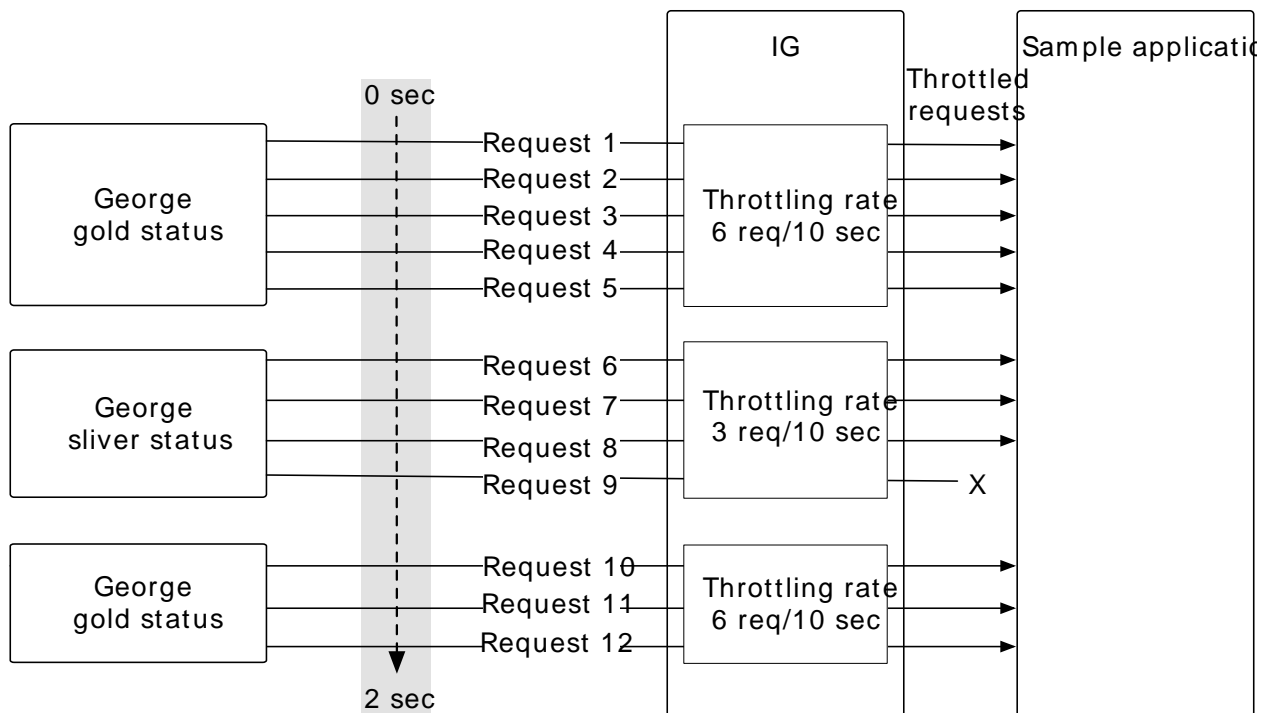
Notice that with a **gold** status, George can access the route 6 times in 10 seconds.

d. In AM, change George's email to **george@other.com**, and then run the last two steps again to see how the access is reduced.

## Considerations for Dynamic Throttling

The following image illustrates what can happen when the throttling rate defined by **throttlingRateMapping** changes frequently or quickly:

*Dynamic Throttling Rate*



In the image, George starts out with a **gold** status. In a two second period, he 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, George has almost reached his throttling rate. When his status is downgraded to silver, those requests are disregarded and the full throttling rate for **silver** is applied. George can now make three more requests even though he had nearly reached his throttling rate with a **gold** status.

After making three requests with a **silver** status, George has reached his throttling rate. When he makes a fourth request, the request is refused.

George is now upgraded back to **gold** and can now make six more requests even though he had reached his throttling rate with a **silver** status.

When you configure `requestGroupingPolicy` and `throttlingRateMapper`, bear in mind what happens when the throttling rate defined by the `throttlingRateMapper` is changed.

## Configuring Scriptable Throttling

This section builds on the example in "Configuring Mapped Throttling". It creates a scriptable throttling filter, where the script applies a throttling rate of 6 requests/10 seconds to requests from gold status users. For all other requests, the script returns `null`, and applies the default rate of 1 request/10 seconds.

### Configure Scriptable Throttling

1. Set up AM as described in "Configure Mapped Throttling".
2. Set up IG:
  - a. Set an environment variable for the IG agent password, and then restart IG:

```
$ export AGENT_SECRET_ID='cGFzc3dvcnQ='
```

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

- b. Add the following route to IG:

Linux

```
$HOME/.openig/config/routes/00-throttle-scriptable.json
```

Windows

```
%appdata%\OpenIG\config\routes\00-throttle-scriptable.json
```

```
{
  "name": "00-throttle-scriptable",
  "baseURI": "http://app.example.com:8081",
  "condition": "${matches(request.uri.path, '^/home/throttle-scriptable')}"
}
```

```

"heap": [
  {
    "name": "SystemAndEnvSecretStore-1",
    "type": "SystemAndEnvSecretStore"
  },
  {
    "name": "AmService-1",
    "type": "AmService",
    "config": {
      "agent": {
        "username": "ig_agent",
        "passwordSecretId": "agent.secret.id"
      },
      "secretsProvider": "SystemAndEnvSecretStore-1",
      "url": "http://openam.example.com:8088/openam/",
      "version": "7"
    }
  }
],
"handler": {
  "type": "Chain",
  "config": {
    "filters": [
      {
        "name": "OAuth2ResourceServerFilter-1",
        "type": "OAuth2ResourceServerFilter",
        "config": {
          "scopes": [
            "mail",
            "employeenumber"
          ],
          "requireHttps": false,
          "realm": "OpenIG",
          "accessTokenResolver": {
            "name": "token-resolver-1",
            "type": "TokenIntrospectionAccessTokenResolver",
            "config": {
              "amService": "AmService-1",
              "providerHandler": {
                "type": "Chain",
                "config": {
                  "filters": [
                    {
                      "type": "HttpBasicAuthenticationClientFilter",
                      "config": {
                        "username": "ig_agent",
                        "passwordSecretId": "agent.secret.id",
                        "secretsProvider": "SystemAndEnvSecretStore-1"
                      }
                    }
                  ],
                  "handler": "ForgeRockClientHandler"
                }
              }
            }
          }
        }
      }
    ]
  }
}

```

```

"name": "ThrottlingFilter-1",
"type": "ThrottlingFilter",
"config": {
  "requestGroupingPolicy": "${contexts.oauth2.accessToken.info.mail}",
  "throttlingRatePolicy": {
    "type": "DefaultRateThrottlingPolicy",
    "config": {
      "delegateThrottlingRatePolicy": {
        "name": "ScriptedPolicy",
        "type": "ScriptableThrottlingPolicy",
        "config": {
          "type": "application/x-groovy",
          "source": [
            "if (contexts.oauth2.accessToken.info.status == status) {",
            "  return new ThrottlingRate(rate, duration)",
            "} else {",
            "  return null",
            "}"
          ],
          "args": {
            "status": "gold",
            "rate": 6,
            "duration": "10 seconds"
          }
        }
      },
      "defaultRate": {
        "numberOfRequests": 1,
        "duration": "10 s"
      }
    }
  }
},
"handler": "ReverseProxyHandler"
}

```

For information about how to set up the IG route in Studio, see "Scriptable Throttling Filter in Structured Editor" in the *Studio User Guide*.

Notice the following features of the route, compared to `00-throttle-mapped.json`:

- The route matches requests to `/home/throttle-scriptable`.
- The `DefaultRateThrottlingPolicy` delegates the management of throttling to the `ScriptableThrottlingPolicy`.
- The script applies a throttling rate to requests from users with gold status. For all other requests, the script returns null and the default rate is applied.

### 3. Test the setup:

- Get an `access_token` for George from AM:

```
$ george_token=$(curl -s \  
--user "client-application:password" \  
--data "grant_type=password&username=george&password=C0stanza&scope=mail%20employeeenumber" \  
http://openam.example.com:8088/openam/oauth2/access_token | jq -r ".access_token")
```

- b. Using the `access_token` for authentication, access the route multiple times. The following example accesses the route 10 times, and writes the output to a file:

```
$ curl -v http://openig.example.com:8080/home/throttle-scriptable/[01-10\] --header  
"Authorization:Bearer ${george_token}" > /tmp/george.txt 2>&1
```

- c. Search the output file to see the result:

```
$ grep "< HTTP/1.1" /tmp/george.txt | sort | uniq -c  
6 < HTTP/1.1 200  
4 < HTTP/1.1 429
```

Notice that with a `gold` status, George can access the route 6 times in 10 seconds.

- d. In AM, change George's email to `george@other.com`, and then run the last two steps again to see how the access is reduced.

## Chapter 20

# SAML 2.0 and Multiple Applications

The chapter extends the example in "Acting As a SAML 2.0 Service Provider" with the service provider `sp`, to add a second service provider.

The new service provider has entity ID `sp2` and runs on the host `sp2.example.com`. To prevent unwanted behavior, the service providers must have different values.

1. Add `sp2.example.com` to your `/etc/hosts` file:

```
127.0.0.1 localhost openam.example.com openig.example.com app.example.com sp.example.com
sp2.example.com
```

2. In IG, configure the service provider files for `sp2`, using the files you created in Step 2:

- a. In `fedlet.cot`, add `sp2` to the list of `sun-fm-trusted-providers`:

```
cot-name=Circle of Trust
sun-fm-cot-status=Active
sun-fm-trusted-providers=openam, sp, sp2
sun-fm-saml2-readerservice-url=
sun-fm-saml2-writerservice-url=
```

- b. Copy `sp.xml` to `sp2.xml`, and copy `sp-extended.xml` to `sp2-extended.xml`.
  - c. In both files, search and replace the following strings:
    - `entityID="sp"`: replace with `entityID="sp2"`
    - `sp.example.com`: replace with `sp2.example.com`
    - `metaAlias="/sp"`: replace with `metaAlias="/sp2"`
    - `/metaAlias/sp`: replace with `/metaAlias/sp2`
  - d. Restart IG.
3. In AM, set up a remote service provider for `sp2`, as in Step 3:
    - a. Select **Applications > Federation > Entity Providers**.
    - b. Drag in or import `sp2.xml` created in the previous step.
    - c. Select Circles of Trust: `Circle of Trust`
  4. Add the following routes to IG:



*Linux*

```
$HOME/.openig/config/routes/saml-sp2.json
```

*Windows*

```
%appdata%\OpenIG\config\routes\saml-sp2.json
```

```
{
  "name": "saml-sp2",
  "condition": "${matches(request.uri.host, 'sp2.example.com') and matches(request.uri.path, '^/saml')}",
  "handler": {
    "type": "SamlFederationHandler",
    "config": {
      "comment": "Use unique session properties for this SP.",
      "assertionMapping": {
        "sp2Username": "cn",
        "sp2Password": "sn"
      },
      "authnContext": "sp2AuthnContext",
      "sessionIndexMapping": "sp2SessionIndex",
      "subjectMapping": "sp2SubjectName",
      "redirectURI": "/sp2"
    }
  }
}
```

*Linux*

```
$HOME/.openig/config/routes/federate-sp2.json
```

*Windows*

```
%appdata%\OpenIG\config\routes\federate-sp2.json
```

```
{
  "name": "federate-sp2",
  "condition": "${matches(request.uri.host, 'sp2.example.com') and not matches(request.uri.path, '^/saml')}",
  "baseURI": "http://app.example.com:8081",
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [
        {
          "condition": "${empty session.sp2Username}",
          "handler": {
            "type": "StaticResponseHandler",
            "config": {
              "status": 302,
              "reason": "Found",
              "headers": {
                "Location": [
                  "http://sp2.example.com:8080/saml/SPInitiatedSSO?metaAlias=/sp2"
                ]
              }
            }
          }
        ]
      }
    }
  }
}
```

```
    },
    {
      "handler": {
        "type": "Chain",
        "config": {
          "filters": [
            {
              "type": "HeaderFilter",
              "config": {
                "messageType": "REQUEST",
                "add": {
                  "x-username": ["${session.sp2Username[0]}"],
                  "x-password": ["${session.sp2Password[0]}"]
                }
              }
            }
          ],
          "handler": "ReverseProxyHandler"
        }
      }
    }
  ]
}
```

5. Test the setup:

- a. Log out of AM, and test the setup with the following links:
  - IDP-initiated SSO
  - SP-initiated SSO
- b. Log in to AM with username **george** and password **C0stanza**.

IG returns the response page showing that the George has logged in.