Gateway Guide

OpenIG 3.1

Paul Bryan
Mark Craig
Jamie Nelson
Guillaume Sauthier

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

Table of Contents

Preface

This guide shows you how to install and configure OpenIG, a high-performance reverse proxy server with specialized session management and credential replay functionality.

1. Who Should Use this Guide

This guide is written for access management designers and administrators who develop, build, deploy, and maintain OpenIG deployments for their organizations. This guide covers the tasks you might perform once or repeat throughout the life cycle of an OpenIG release.

You do not need to be an expert to learn something from this guide, though a background in HTTP, access management web applications can help. You do need some background in managing services on your operating systems and in your application servers. You can nevertheless get started with this guide, and then learn more as you go along.

2. Formatting Conventions

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

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

Command-line, terminal sessions are formatted as follows:

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

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

Program listings are formatted as follows:

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

3. Accessing Documentation Online

ForgeRock publishes comprehensive documentation online:

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

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

- ForgeRock product documentation, such as this document, aims to be technically accurate and complete with respect to the software documented. It is visible to everyone and covers all product features and examples of how to use them.

4. Using the ForgeRock.org Site

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

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

Chapter 1

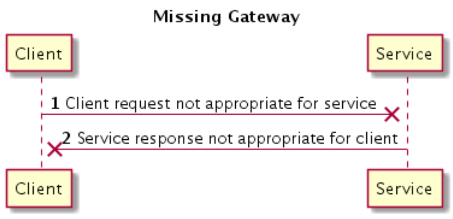
Understanding OpenIG

This chapter introduces OpenIG, briefly covering essential concepts.

1.1. About OpenIG

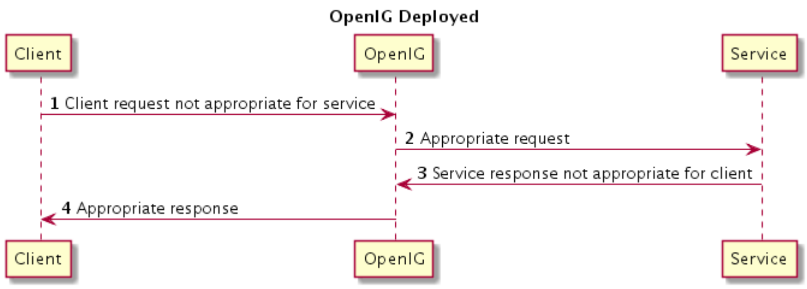
Most organizations have valuable existing services that they cannot easily integrate into newer architectures. Often, however, they also cannot change the existing services. Without a gateway to bridge the gap, some client applications cannot communicate with these existing services.

Figure 1.1. Missing Gateway


Missing Gateway
Client
Service
1 Client request not appropriate for service
2 Service response not appropriate for client
Client
Service

OpenIG works as an HTTP gateway, also known as a reverse proxy. You deploy OpenIG on the network so that it intercepts both client requests and also server responses.

Figure 1.2. OpenIG Deployed



Clients then exchange with protected servers through OpenIG. This means that without touching either clients or servers, you can configure OpenIG to add new capabilities to existing services.

The list that follows names some of what you can add by using OpenIG.

- Access management integration
- Application & API security
- Credential replay
- OAuth 2.0 support
- OpenID Connect 1.0 support
- Network traffic control
- Proxy with request & response capture
- Request & response rewriting
- SAML 2.0 federation support
- Single sign-on

OpenIG supports these capabilities as out-of-the-box configuration options. Once you understand the essential concepts covered in this chapter, try the demonstrations in this guide to see for yourself how to add these features by using OpenIG.

1.2. The Exchange

OpenIG handles HTTP requests and responses with a wrapper called the *exchange*.

The OpenIG exchange encapsulates HTTP requests and responses that pass through OpenIG, as well as the principal associated with the request, the session context associated with the client, and any other state information needed. OpenIG makes it easy to access arbitrary state information through the exchange so that you can use it throughout the duration of the exchange, even when the exchange calls for interaction with additional services.

In addition, OpenIG includes information in the exchange about the client that made the incoming request, such as the client host, port, and IP address, and also the user-agent description, the login of the user making the request, and the X.509 certificates presented by the client when these are available as part of the request.

1.3. The Configuration

OpenIG represents its configuration in JSON format, which is store in flat files.[1] You configure OpenIG by editing the JSON flat files.

[1] OpenIG also uses additional file formats for SAML 2.0, but the primary configuration files are in JSON format.

After installation, you add at least one configuration file. Each configuration file holds a JSON object. At minimum, the JSON object specifies a "handler" to deal with the exchange.

The following very simple configuration routes exchanges to be handled according to separate route configurations.

```
{
    "handler": {
        "type": "Router"
    }
}
```

The "handler" indicates which object OpenIG invokes first. A handler is an object responsible for producing a response to a request, therefore every route must call a handler.

Notice in this case that the "handler" field takes an object as its value. This is an inline declaration. If you only use the object once where it is declared, then it makes sense to use an inline declaration.

To change the definition of an object defined by default or when you need to declare an object once and use it multiple times, you declare object definitions in the "heap", and then reference the objects by "name". You can also use this technique instead of inline declarations.

The following example declares an identical "Router" object as above and references it by its name.

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

Notice that the "heap" takes an array. As the "heap" holds configuration objects all at the same level, you can impose any hierarchy or order that you like when referencing objects. Yet, 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.

Each configuration object has a "type", a "name", and an optional "config".

- The "type" must be the type name of the configuration object.[2]

  As you see in the rest of this guide and in the *Reference*, OpenIG defines many types for different purposes.

[2] For built-in objects, you can use the short name alias. If the object has no alias, use the fully qualified class name of the Java class implementing the object.

- The "name" takes a string that is unique in the list of objects.

  You can omit this field when declaring objects inline.

- The contents of the "config" object depend on the "type".

  When all the configuration settings for the type are optional, the "config" field is also optional, as in this example. If all configuration settings are optional, then omitting the "config" field, or setting the "config" field to an empty object, "config": {}, or setting "config": null all signify that the object uses default settings.

The configuration can specify additional objects as well. For example, you can configure an "HttpClient" that OpenIG uses to connect to servers. The following "HttpClient" configuration uses defaults for all settings, except "hostnameVerifier", which it configures to verify host names in SSL certificates in the same way as most browsers.

```
{
    "name": "HttpClient",
    "type": "HttpClient",
    "config": {
        "hostnameVerifier": "BROWSER_COMPATIBLE"
    }
}
```

Decorators are additional heap objects that let you extend what another object can do. For example, a "CaptureDecorator" enables filters and handlers to log requests and responses. A "TimerDecorator" logs processing times. You decorate the configuration of other objects with the names of decorators as field names. OpenIG defines both a "CaptureDecorator" named "capture" and also a "TimerDecorator" named "timer" by default. You can therefore log requests, responses, and processing times by adding decorations as shown in the following example.

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

In addition to decorators, OpenIG also creates other utility objects with default settings, including "HttpClient", "LogSink", and "TemporaryStorage". You can reference these objects by name without configuring them unless you need to override the default configurations.

Routes, mentioned here and described in more detail in Section 1.4, "Routing", inherit settings from their parent configurations. This means that you can configure global objects in the "heap" of the base configuration for example, and then reference the objects by name in any other OpenIG configuration.

1.4. Routing

OpenIG routing lets you use multiple configuration files. Routing also lets OpenIG reload configurations that you change at runtime without restarting OpenIG.

You can use routing where OpenIG protects multiple services, or multiple different endpoints of the same service. You can also use routing when handling an exchange involves multiple steps, for example because you must redirect the client to authenticate with an identity provider before accessing the service.

A *router*, as shown in Section 1.3, "The Configuration", takes responsibility for managing the routes used, periodically reloading changed routes unless configured to load them only at startup.

Notice in the example that the router does not specify any routes. Instead, routes optionally specify their own "condition". If a route "condition" is true, then the route handles the exchange.

The following example specifies a condition that is true when the incoming request path is `/login`.

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

If the route has no "condition", or if the value of the condition is null, then the route matches any exchange. Furthermore, OpenIG orders routes lexicographically by name.

You can use these features to have both optional and default routes. For example, you could name your routes to check conditions in order: `01-login.json`, `02-protected.json`, `99-default.json`. Alternatively, you can name routes using the "name" property on the route.

A router configuration can specify where to look for route files. As a "Router" is a kind of "handler", routes can have routers, too.

1.5. Filters, Handlers, & Chains

Routing only delegates exchange handling. It does not actually deal with exchanges. To deal with an exchange, you chain together filters and handlers.

- A *handler* either delegates to another handler, or it produces a response.

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

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

- A *filter* transforms something in the exchange.

  A filter can leave the exchange unchanged. Alternatively a filter can even replace the request or the response, for example generating a static request that replaces the client request. Other filters only add or change some of the data in the exchange.

- A *chain* takes a list of filters and one handler. (The list of filters can be empty.)

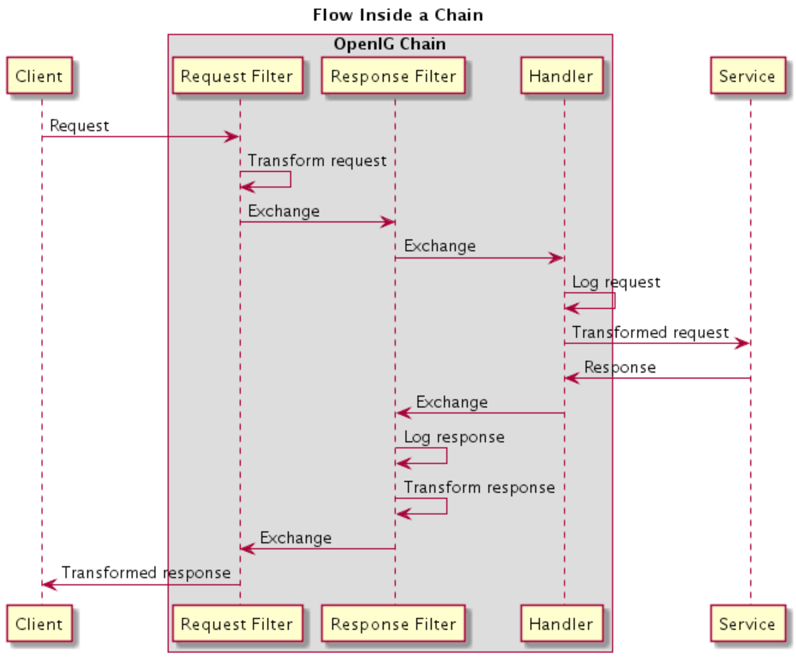
  Like a router, a chain itself is technically a handler. You can therefore place a chain anywhere in the configuration that you can place a handler.

  The chain dispatches processing to the filters in order, and then to the handler.

  Initially the filters process the incoming exchange, with each filter handing off to the next filter and finally the handler. Then after the handler produces the response, the filters process the outgoing exchange on its way to the client. The same filter can process the incoming request and the outgoing response. Many filters, however, either process the request or the response.

The following diagram shows the flow inside a chain that has a request filter transforming the request, a response filter transforming the response, and a handler sending a request to a service to get a response. Notice how the flow traverses the filters in reverse order when the outgoing exchange comes back from the handler.

Figure 1.3. Flow Inside a Chain



The following route configuration demonstrates the flow, but without an external service.

```
{
    "handler": {
        "type": "Chain",
        "comment": "Base configuration defines the capture decorator",
        "config": {
            "filters": [
                {
                    "type": "HeaderFilter",
                    "comment": "Same header on all requests",
                    "config": {
                        "messageType": "REQUEST",
                        "add": {
                            "X-MyHeaderFilter": [
                                "Added by HeaderFilter to request"
                            ]
                        }
                    }
                },
                {
                    "type": "HeaderFilter",
                    "comment": "Remove X-Powered-By from response",
                    "capture": "response",
                    "config": {
                        "messageType": "RESPONSE",
                        "remove": [
                            "X-Powered-By"
                        ]
                    }
                }
            ],
            "handler": {
                "type": "StaticResponseHandler",
                "comment": "Same response to all requests",
                "capture": "request",
                "config": {
                    "status": 200,
                    "reason": "OK",
                    "headers": {
                        "X-Powered-By": [
                            "OpenIG"
                        ]
                    },
                    "entity": "<html><p>Hello, World!</p></html>"
                }
            }
        }
    }
}
```

When the "Chain" gets the request, it processes the exchange as follows.

1. The first "HeaderFilter" adds a header to the incoming request.

2. The second "HeaderFilter" deals with responses, so it simply passes the exchange to the handler.

3. The "StaticResponseHandler" captures (logs) the request.
4. The "StaticResponseHandler" itself produces a response having an entity body and a header.
5. The second "HeaderFilter" captures (logs) the response.
6. The second "HeaderFilter" removes the header added to the response.
7. The first "HeaderFilter" deals with requests, so it simply passes the outgoing exchange back to OpenIG.

Suppose this chain configuration is the only route active for the request. In that case, the flow produces the following.

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

### Captured incoming request (inside OpenIG exchange)
GET / HTTP/1.1
X-MyHeaderFilter: Added by HeaderFilter to request
Accept: */*
Host: www.example.com:8080

### Captured outgoing response (inside OpenIG exchange)
HTTP/1.1 200 OK
Content-Length: 33
X-Powered-By: OpenIG

<html><p>Hello, World!</p></html>

### Final response to user-agent
HTTP/1.1 200 OK
Content-Length: 33

<html><p>Hello, World!</p></html>
```

1.6. Comments in OpenIG Configuration Files

JSON does not specify a notation for comments.

However, when OpenIG does not recognize a JSON field name, it ignores the field. This makes it possible to use comments in configuration files.

To make your configuration files easier to read, use the following conventions when commenting.

"comment"

Use the "comment" fields to add text comments.

The following "CaptureDecorator" configuration includes a text comment.

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

"_field-name"

Use an underscore (`_`) to comment a field temporarily.

The following "CaptureDecorator" configuration has `"captureEntity": true` commented out. As a result, it uses the default setting (`"captureEntity": false`).

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

1.7. Where To Go From Here

Now that you understand the essential concepts, start using OpenIG with the help of the following chapters.

Getting Started

This chapter shows you how to get OpenIG up and running quickly.

Installation in Detail

This chapter covers more advanced installation procedures.

Getting Login Credentials From Data Sources

This chapter shows you how to configure OpenIG to look up credentials in external sources, such as a file or a database.

Getting Login Credentials From OpenAM

This chapter walks you through an OpenAM integration with OpenAM's password capture and replay feature.

OpenIG as a SAML 2.0 Service Provider

This chapter shows how to configure OpenIG as a SAML 2.0 Identity Provider.

OpenIG as an OAuth 2.0 Resource Server

This chapter explains how OpenIG acts as an OAuth 2.0 Resource Server, and follows with a tutorial that shows you how to use OpenIG as a resource server.

OpenIG as an OAuth 2.0 Client

This chapter explains how OpenIG acts as an OAuth 2.0 client or OpenID Connect 1.0 relying party, and follows with a tutorial that shows you how to use OpenIG as an OpenID Connect 1.0 relying party.

Configuring Routes

This chapter shows how to configure OpenIG to allow dynamic configuration changes and route to multiple applications.

Configuration Templates

This chapter provides sample OpenIG configuration files for common use cases.

ForgeRock can also help you succeed in your projects involving OpenIG. You can purchase OpenIG support subscriptions and training courses from ForgeRock and from consulting partners around the world and in your area. To contact ForgeRock, send mail to info@forgerock.com. To find a partner in your area, see http://forgerock.com/partners/find-a-partner/.

Chapter 2
Getting Started

This chapter provides instructions to get OpenIG up and running on Jetty, configured to serve as reverse proxy to a minimal HTTP server for use when following along with the documentation. This allows you to quickly see how OpenIG works, and provides hands on experience with a few key features. For more general installation and configuration instructions, start with the chapter on *Installation in Detail*.

2.1. Before You Begin

Make sure you have a supported Java Development Kit installed. For details, see the *Release Notes* section, *JDK Version* in the *Release Notes*.

2.2. Install OpenIG

You install OpenIG in the root context of a web application container. In this chapter, you use Jetty server as the web application container.

To perform initial installation, follow these steps.

1. Download and unzip a supported version of Jetty server.

   Supported versions are listed in the *Release Notes* section, *Web Application Containers* in the *Release Notes*.

2. Download the OpenIG war file.

3. Deploy OpenIG in the root context.

   Copy the OpenIG war file as `root.war` to the `/path/to/jetty/webapps/`.

   ```
   $ cp OpenIG-3.1.0.war /path/to/jetty/webapps/root.war
   ```

   Jetty automatically deploys OpenIG in the root context on startup.

4. Start Jetty in the background:

```
$ /path/to/jetty/bin/jetty.sh start
```

Or start Jetty in the foreground:

```
$ cd /path/to/jetty/
$ java -jar start.jar
```

5. Verify that you can see the OpenIG welcome page at http://localhost:8080.

   When you start OpenIG without a configuration, requests to OpenIG default to a welcome page with a link to the documentation.

6. Stop Jetty in the background:

```
$ /path/to/jetty/bin/jetty.sh stop
```

   Or stop Jetty in the foreground by entering Ctrl+C in the terminal where Jetty is running.

2.3. Install an Application to Protect

Now that OpenIG is installed, set up a sample application to protect.

Follow these steps.

1. Download and run the minimal HTTP server .jar to use as the application to protect.

```
$ java -jar openig-doc-samples-3.1.0-jar-with-dependencies.jar
Jun 11, 2014 4:32:42 PM org.forgerock.openig.doc.SampleServer runServer
INFO: Starting HTTP server on port 8081
Jun 11, 2014 4:32:42 PM org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [0.0.0.0:8081]
Jun 11, 2014 4:32:42 PM org.glassfish.grizzly.http.server.HttpServer start
INFO: [HttpServer] Started.
Jun 11, 2014 4:32:42 PM org.forgerock.openig.doc.SampleServer runServer
INFO: Press Ctrl+C to stop the server.
```

   By default, this server listens on port 8081. If that port is not free, specify another port.

```
$ java -jar openig-doc-samples-3.1.0-jar-with-dependencies.jar 8888
Jun 11, 2014 4:33:04 PM org.forgerock.openig.doc.SampleServer runServer
INFO: Starting HTTP server on port 8888
Jun 11, 2014 4:33:04 PM org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [0.0.0.0:8888]
Jun 11, 2014 4:33:04 PM org.glassfish.grizzly.http.server.HttpServer start
INFO: [HttpServer] Started.
Jun 11, 2014 4:33:04 PM org.forgerock.openig.doc.SampleServer runServer
INFO: Press Ctrl+C to stop the server.
```

2. Now access the minimal HTTP server through a browser at http://localhost:8081.

   Login with username `demo`, password `changeit`. You should see a page that includes the username, `demo`, and some information about your browser request.

2.4. Configure OpenIG

Now that you have installed both OpenIG and also a sample application to protect, and configure OpenIG.

Follow these steps to configure OpenIG to proxy traffic to the sample application.

1. Prepare the OpenIG configuration.

   Add the following base configuration file as `$HOME/.openig/config/config.json`. By default, OpenIG looks for `config.json` in the `$HOME/.openig/config` directory.

```
{
    "handler": {
        "type": "Router",
        "audit": "global",
        "capture": "all"
    },
    "heap": [
        {
            "name": "LogSink",
            "type": "ConsoleLogSink",
            "config": {
                "level": "DEBUG"
            }
        },
        {
            "name": "JwtSession",
            "type": "JwtSession"
        },
        {
            "name": "ClientHandler",
            "type": "ClientHandler"
        },
        {
            "name": "capture",
            "type": "CaptureDecorator",
            "config": {
                "captureEntity": true,
                "_captureExchange": true
            }
        }
    ],
    "baseURI": "http://www.example.com:8081"
}
```

```
$ mkdir -p $HOME/.openig/config
$ vi $HOME/.openig/config/config.json
```

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

If you adapt this base configuration for production use, make sure to adjust the log level, and to deactivate the "CaptureDecorator" that generates several log message lines for each request and response. Also consider editing the router based on recommendations in the section, *Locking Down Route Configurations*.

2. Add the following default route configuration file as `$HOME/.openig/config/routes/99-default.json`. By default, the Router defined in the base configuration file looks for routes in the `$HOME/.openig/config/routes` directory.

```
{
    "handler": "ClientHandler"
}
```

```
$ mkdir $HOME/.openig/config/routes
$ vi $HOME/.openig/config/routes/99-default.json
```

On Windows, the file name should be `%appdata%\OpenIG\config\routes\99-default.json`.

3. Start Jetty in the background:

```
$ /path/to/jetty/bin/jetty.sh start
```

Or start Jetty in the foreground:

```
$ cd /path/to/jetty/
$ java -jar start.jar
```

2.5. Configure the Network

So far you have deployed OpenIG in the root context of Jetty on port 8080. Since OpenIG is a reverse proxy you must make sure that all traffic from your browser to the protected application goes through OpenIG. In other words, the network must be configured so that the browser goes to OpenIG instead of going directly to the protected application.

Although if you followed the installation steps you are running both OpenIG and the minimal HTTP server on the same host as your browser (probably your laptop or desktop), keep in mind that network configuration is an important deployment step. To encourage you to keep this in mind, the sample configuration for this chapter expects the minimal HTTP server to be running on `www.example.com`, rather than `localhost`.

The quickest way to configure the network locally is to add an entry to your `/etc/hosts` file on UNIX systems or `%SystemRoot%\system32\drivers\etc\hosts` on Windows. See the Wikipedia entry, *Hosts (file)*, for more information on host files. If you are indeed running all servers in this chapter on the same host, add the following entry to the hosts file.

```
127.0.0.1    www.example.com
```

If you are running the browser and OpenIG on separate hosts, add the IP address of the host running OpenIG to the hosts file on the system running the browser, where the host name matches that of protected application. For example, if OpenIG is running on a host with IP address 192.168.0.15:

```
192.168.0.15    www.example.com
```

If OpenIG is on a different host from the protected application, also make sure that the host name of the protected application resolves correctly for requests from OpenIG to the application.

Tip

Some browsers cache IP address resolutions, even after clearing all browsing data. Restart the browser after changing the IP addresses of named hosts.

The simplest way to make sure you have configured your DNS or host settings properly for remote systems is to stop OpenIG and then to make sure you cannot reach the target application with the host name and port number of OpenIG. If you can still reach it, double check your host settings.

Also make sure name resolution is configured to check host files before DNS. This configuration can be found in `/etc/nsswitch.conf` for most UNIX systems. Make sure `files` is listed before `dns`.

2.6. Try It Out

http://www.example.com:8080/ should take you to the home page of the minimal HTTP server.

What just happened?

When your browser goes to `http://www.example.com:8080/`, it is actually connecting to OpenIG deployed in Jetty. OpenIG proxies all traffic it receives to the protected application at `http://www.example.com:8080/`, and returns responses from the application to your browser. It does this based on the configuration that you set up.

Consider the base configuration file first, `config.json`. The base configuration file specifies a router handler named "Router". OpenIG calls this handler when it receives an incoming request. In addition, it uses the "LogSink" to log debug messages to the console. Alternatively, to send log messages to a file you can use a "FileLogSink" in the *Reference*, rather than a "ConsoleLogSink".

The "baseURI" setting in turn changes the request URI to point the request to the sample application to protect. The "Router" captures the request on the way in, and captures the response on the way out.

The "Router" routes processing to separate route configurations.

For now the only route available is the the default route you added, `99-default.json`. The default route calls a "ClientHandler" with the default configuration. This "ClientHandler" simply proxies the request to and the response from the sample application to protect without changing either the request or the response. Therefore, the browser request is sent unchanged to the sample application and the response from the sample application is returned unchanged to your browser.

Now change the OpenIG configuration to log you in automatically with hard-coded credentials.

1. Add a route to automatically log you in as username `demo`, password `password`.

   Add the following route configuration file as `$HOME/.openig/config/routes/01-static.json`.

```
{
    "handler": {
        "type": "Chain",
        "config": {
            "filters": [
                {
                    "type": "StaticRequestFilter",
                    "config": {
                        "method": "POST",
                        "uri": "http://www.example.com:8081",
                        "form": {
                            "username": [
                                "demo"
                            ],
                            "password": [
                                "changeit"
                            ]
                        }
                    }
                }
            ],
            "handler": "ClientHandler"
        }
    },
    "condition": "${matches(exchange.request.uri.path, '^/static')}"
}
```

On Windows, the file name should be `%appdata%\OpenIG\config\routes\01-static.json`.

2. Access the new route, http://www.example.com:8080/static.

   This time, OpenIG logs you in automatically.

Also view the information logged about requests and responses, which shows up in the Jetty log.

What's happening behind the scenes?

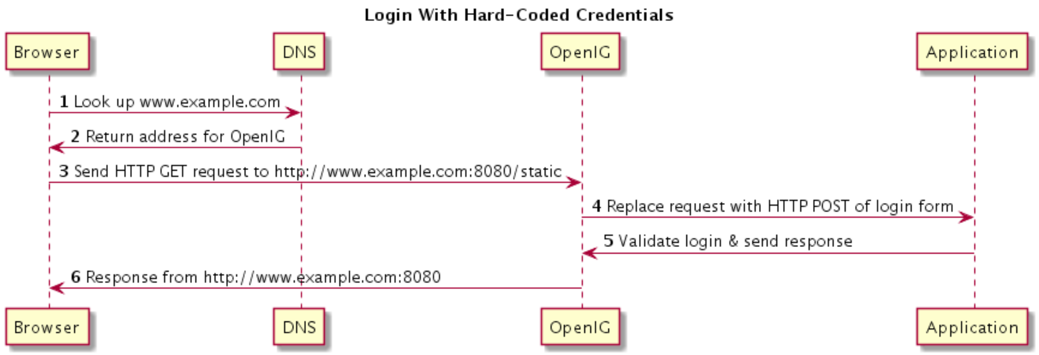
With the original configuration, OpenIG does not change requests or responses, but only proxies requests and responses, and captures request and response information.

After you change the configuration, OpenIG continues to capture request and response data. When your request does not goes to the default route, but instead goes to `/static`, then the condition on the new route you added matches the request. OpenIG therefore uses the new route you added.

Using the route configuration in `01-static.json`, OpenIG replaces your browser's original HTTP GET request with an HTTP POST login request containing credentials to authenticate. As a result, instead of the home page with a login form, OpenIG logs you in directly, and the application responds with the page you see after logging in. OpenIG then returns this response to your browser.

The following sequence diagram shows the steps.

Figure 2.1.


Login With Hard-Coded Credentials
Browser
DNS
OpenIG
Application
1 Look up www.example.com
2 Return address for OpenIG
3 Send HTTP GET request to http://www.example.com:8080/static
4 Replace request with HTTP POST of login form
5 Validate login & send response
6 Response from http://www.example.com:8080
Browser
DNS
OpenIG
Application

1. The browser host makes a DNS request for the IP address of the HTTP server host, `www.example.com`.
2. DNS responds with the address for OpenIG.
3. Browser sends a request to the HTTP server.
4. OpenIG replaces the request with an HTTP POST request, including the login form with hard-coded credentials.
5. HTTP server validates the credentials, and responds with the profile page.
6. OpenIG passes the response back to the browser.

Chapter 3
Installation in Detail

This chapter covers more advanced installation procedures.

- Make sure you have a supported Java version installed.

  See the *Release Notes* section, *JDK Version* in the *Release Notes*, for details.

- Prepare a deployment container.

  For details, see Section 3.1, "Configuring Deployment Containers".

- Prepare the network to use OpenIG as a reverse proxy.

  For details, see Section 3.2, "Preparing the Network".

- Download, deploy, and configure OpenIG.

  For details, see Section 3.3, "Installing OpenIG".

3.1. Configuring Deployment Containers

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

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

For further information on advanced configuration for a particular container, see the container documentation.

3.1.1. About Securing Connections

OpenIG 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 OpenIG is acting as a server, the web application container where OpenIG runs is responsible for setting up TLS connections with client applications that connect to OpenIG. For details, see

Section 3.1.3.2, "Configuring Jetty For HTTPS (Server-Side)" or Section 3.1.2.2, "Configuring Tomcat For HTTPS (Server-Side)".

When OpenIG is acting as a client, the HttpClient in the *Reference* configuration sets up TLS connections from OpenIG to other servers. For details, see Section 3.5, "Configuring OpenIG For HTTPS (Client-Side)".

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 generally costs money.

It is also possible for you to self-sign certificates. The trade off is that although you do not have to pay any money, 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 key store 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 trust store. You might therefore need to install those signing certificates on the client side as well.

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

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

The Java Secure Socket Extension (JSSE), part of the Java environment, provides security services that OpenIG 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*.

3.1.2. Configuring Apache Tomcat For OpenIG

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

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

Configure Tomcat to use the same protocol as the application you are protecting with OpenIG. 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 as well.

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.

3.1.2.1. Configuring Tomcat Cookie Domains

If you use OpenIG for more than a single protected application and the protected applications are on different hosts, then you must configure Tomcat to set domain cookies. To do this, add a session cookie domain context element that specifies the domain to `/path/to/conf/Catalina/`*`server`*`/root.xml`, as in the following example.

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

Restart Tomcat to read the configuration changes.

3.1.2.2. Configuring Tomcat For HTTPS (Server-Side)

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

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

Also create a key store holding a self-signed certificate.

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

Notice the key store file location and the key store 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 Apache Tomcat documentation on configuring HTTPS.

3.1.2.3. Configuring Tomcat to Access MySQL Over JNDI

If OpenIG accesses an SQL database, then you must configure Apache Tomcat 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/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.

3.1.3. Configuring Jetty For OpenIG

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

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

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

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

3.1.3.1. Configuring Jetty Cookie Domains

If you use OpenIG for more than a single protected application and the protected applications are on different hosts, then you must configure Jetty to set domain cookies. To do this, add a session domain handler element that specifies the domain to `/path/to/jetty/etc/jetty.xml`, as in the following example.

```
<Get name="sessionHandler">
    <Get name="sessionManager">
        <Set name="sessionDomain">.example.com</Set>
    </Get>
</Get>
```

Restart Jetty to read the configuration changes.

3.1.3.2. Configuring Jetty For HTTPS (Server-Side)

To get Jetty up quickly on an SSL port, follow the steps in this section.

These steps involve replacing the built-in key store with your own.

1. If you have not done so already, remove the built-in key store.

```
$ rm /path/to/jetty/etc/keystore
```

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

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

3. Find the obfuscated form of the password.

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

4. Edit the SSL Context Factory entry in the Jetty configuration file, `/path/to/jetty/etc/jetty-ssl.xml`.

```
<New id="sslContextFactory" class="org.eclipse.jetty.http.ssl.SslContextFactory">
  <Set name="KeyStore"><Property name="jetty.home" default="." />/etc/keystore</Set>
  <Set name="KeyStorePassword">OBF:1v2j1uum1xtv1zej1zer1xtn1uvk1v1v</Set>
  <Set name="KeyManagerPassword">OBF:1v2j1uum1xtv1zej1zer1xtn1uvk1v1v</Set>
  <Set name="TrustStore"><Property name="jetty.home" default="." />/etc/keystore</Set>
  <Set name="TrustStorePassword">OBF:1v2j1uum1xtv1zej1zer1xtn1uvk1v1v</Set>
</New>
```

5. Uncomment the line specifying that configuration file in `/path/to/jetty/start.ini`.

```
etc/jetty-ssl.xml
```

6. Restart Jetty.

7. Browse https://www.example.com:8443.

   You should see a warning in the browser that the (self-signed) certificate is not recognized.

3.1.3.3. Configuring Jetty to Access MySQL Over JNDI

If OpenIG 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.

3.2. Preparing the Network

In order for OpenIG to function as a reverse proxy, browsers attempting to access the protected application must go through OpenIG instead.

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

Restart the browser after making this change.

3.3. Installing OpenIG

Follow these steps to install OpenIG.

1. Download the OpenIG .war file.

   Browse to https://backstage.forgerock.com/downloads to download the software.

2. Deploy the OpenIG war file *to the root context* of the web application container.

   OpenIG must be deployed to the root context, not below.

3. Prepare your OpenIG configuration files.

   By default, OpenIG files are located under `$HOME/.openig` on Linux, Mac OS X, and UNIX systems, and `%appdata%\OpenIG` on Windows systems. OpenIG uses the following file system directories.

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

   > OpenIG configuration files, where the main configuration file is `config.json`.

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

   > OpenIG route configuration files.
   >
   > See the chapter, *Configuring Routes*, for more information.

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

   > OpenIG SAML 2.0 configuration files.
   >
   > See the chapter, *Using OpenIG Federation*, for more information.

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

   > OpenIG script files, for Groovy scripted filters and handlers.
   >
   > See the chapter, *Extending OpenIG*, for more information.

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

   > OpenIG temporary files.
   >
   > This location can be used for temporary storage.

   You can change `$HOME/.openig` (or `%appdata%\OpenIG`) from the default location in the following ways.

   - Unpack the OpenIG war file, and edit the `WEB-INF/web.xml` application descriptor to set the `openig-base` initialization parameter to the full path to the base location for OpenIG files, as in the following example.

```
<servlet>
  <servlet-name>GatewayServlet</servlet-name>
  <servlet-class>org.forgerock.openig.servlet.GatewayServlet</servlet-class>
  <init-param>
    <param-name>openig-base</param-name>
    <param-value>/path/to/openig</param-value>
  </init-param>
</servlet>
```

- Set the `OPENIG_BASE` environment variable to the full path to the base location for OpenIG files.

  ```
  # On Linux, Mac OS X, and UNIX using Bash
  $ export OPENIG_BASE=/path/to/openig

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

- Set the `openig.base` Java system property to the full path to the base location for OpenIG files when starting the web application container where OpenIG runs, as in the following example that starts Jetty server in the foreground.

  ```
  $ java -Dopenig.base=/path/to/openig -jar start.jar
  ```

If you have not yet prepared configuration files, then start with the configuration from the chapter on *Getting Started*.

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

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

4. Start the web container where OpenIG is deployed.

5. Browse to the protected application.

   OpenIG should now proxy all traffic to the application.

6. Make sure the browser is going through OpenIG.

   Verify this in one of the following ways.

   - a. Stop the OpenIG web container.

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

c. Start the OpenIG web container.

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

- Check the LogSink to see that traffic is going through OpenIG.

  The default ConsoleLogSink is the deployment container log.

3.4. Preparing For Load Balancing & Failover

For a high scale or highly available deployment, you can prepare a pool of OpenIG 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 OpenIG saves in the exchange, or retrieves locally from the OpenIG server system. If information is retrieved locally, then also consider setting up failover so that 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.

OpenIG can save state information in the exchange in several ways.

- Handlers including a SamlFederationHandler in the *Reference* or a custom ScriptableHandler in the *Reference* can store information in the exchange. Most handlers depend on information in the exchange, some of which is first stored by OpenIG.

- Filters including those having types AssignmentFilter in the *Reference*, HeaderFilter in the *Reference*, OAuth2ClientFilter in the *Reference*, OAuth2ResourceServerFilter in the *Reference*, ScriptableFilter in the *Reference*, SqlAttributesFilter in the *Reference*, and StaticRequestFilter in the *Reference* can store information in the exchange. Most filters depend on information in the exchange, some of which is first stored by OpenIG.

OpenIG can also retrieve information locally in several ways.

- Filters and handlers including FileAttributesFilter in the *Reference*, ScriptableFilter in the *Reference*, ScriptableHandler in the *Reference*, and SqlAttributesFilter in the *Reference* can depend on local system files or container configuration.

By default the exchange data resides in memory in the container where OpenIG runs. This includes the default session implementation, which is backed by the HttpSession that the container handles. You can opt to store session data on the user-agent instead, however. For details and to consider whether your data fits, see JwtSession in the *Reference*. When you use the `JwtSession` implementation, be sure to share the encryption keys across all servers, so that any server can read session cookies from any other.

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

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

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

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

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

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

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

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

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

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

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

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

3.5. Configuring OpenIG For HTTPS (Client-Side)

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

If however the server certificates are self-signed or signed by a CA whose certificate is not trusted out of the box, then you can configure a KeyStore in the *Reference*, TrustManager in the *Reference*, and optionally a KeyManager in the *Reference* to reference when configuring an HttpClient in the *Reference* to enable OpenIG to trust servers when acting as a client.

The KeyStore holds the servers' certificates or the CA signing certificate. The TrustManager allows OpenIG to handle the certificates in the KeyStore when deciding whether to trust a server certificate. The optional KeyManager allows OpenIG to present its certificate from the key store when the server must authenticate OpenIG as client. The HttpClient references whatever TrustManager and KeyManager you configure.

You can configure each of these either globally for the OpenIG server, of locally for a particular ClientHandler configuration.

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

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

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

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

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

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

The HttpClient configuration has the following security settings.

"trustManager"

This references your TrustManager.

Recall that you must configure this when your server certificates are not trusted out of the box.

"hostnameVerifier"

This defines how the HttpClient verifies host names in server certificates.

By default, host name verification is turned off.

"keyManager"

This references your optional KeyManager.

Configure this if servers request that OpenIG present its certificate as part of mutual authentication.

In that case, generate a key pair for OpenIG, and have the certificate signed by a well-known CA. See the **keytool** documentation for instructions. You can use a different key store for the KeyManager than you use for the TrustManager.

The following HttpClient configuration references "MyTrustManager" and sets browser-compatible host name verification.

```
{
    "name": "HttpClient",
    "type": "HttpClient",
    "config": {
        "hostnameVerifier": "BROWSER_COMPATIBLE",
        "trustManager": "MyTrustManager"
    }
}
```

3.6. Setting Up Keys For JWT Encryption

You can use JwtSession in the *Reference* to configure OpenIG to store session information in JWT cookies on the user-agent, rather than storing the information in the container where OpenIG runs.

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

Alternatively, OpenIG can use keys from a key store. The following steps describe how to prepare the key store for JWT encryption.

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

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

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

2. Add a KeyStore in the *Reference* to your configuration that references the key store file.

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

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

   ```
   {
       "name": "MyJwtSession",
       "type": "JwtSession",
       "config": {
           "keystore": "MyKeyStore",
           "alias": "jwe-key",
           "password": "changeit",
           "cookieName": "OpenIG"
       }
   }
   ```

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

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

Chapter 4
Getting Login Credentials From Data Sources

In the chapter on *Getting Started*, you learned how to configure OpenIG to proxy traffic and capture request and response data. You also learned how to configure OpenIG to use a static request to log in with hard-coded credentials.

This chapter shows you how OpenIG can look up credentials in external sources. For example, OpenIG can look up credentials in a file or in a relational database.

4.1. Before You Start

Before you start this tutorial, prepare OpenIG and the minimal HTTP server as you did for the chapter on *Getting Started*.

OpenIG should be running in Jetty, configured to access the minimal HTTP server as described in that chapter.

4.2. Login With Credentials From a File

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

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

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

OpenIG looks up the user credentials based on the user's email address. OpenIG relies for this on a `FileAttributesFilter` configuration object.

Follow these steps to set up login with credentials from a file.

1. Add the user file on your system.

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

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

2. Add a new route to the OpenIG configuration to use the `FileAttributesFilter` configuration object.

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

```
{
    "handler": {
        "type": "Chain",
        "config": {
            "filters": [
                {
                    "type": "FileAttributesFilter",
                    "config": {
                        "target": "${exchange.credentials}",
                        "file": "/tmp/userfile",
                        "key": "email",
                        "value": "george@example.com"
                    }
                },
                {
                    "type": "StaticRequestFilter",
                    "config": {
                        "method": "POST",
                        "uri": "http://www.example.com:8081",
                        "form": {
                            "username": [
                                "${exchange.credentials.username}"
                            ],
                            "password": [
                                "${exchange.credentials.password}"
                            ]
                        }
                    }
                }
            ],
            "handler": "ClientHandler"
        }
    },
    "condition": "${matches(exchange.request.uri.path, '^/file')}"
}
```

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

Notice the following features of the new route.

- The "FileAttributesFilter" specifies the file to access, the key and value to look up to retrieve the user's record, and where the exchange stores the search results.
- The "StaticRequestFilter" filter retrieves the username and password from the exchange and replaces your browser's original HTTP GET request with an HTTP POST login request that contains the credentials to authenticate.
- The route matches requests to `/file`.

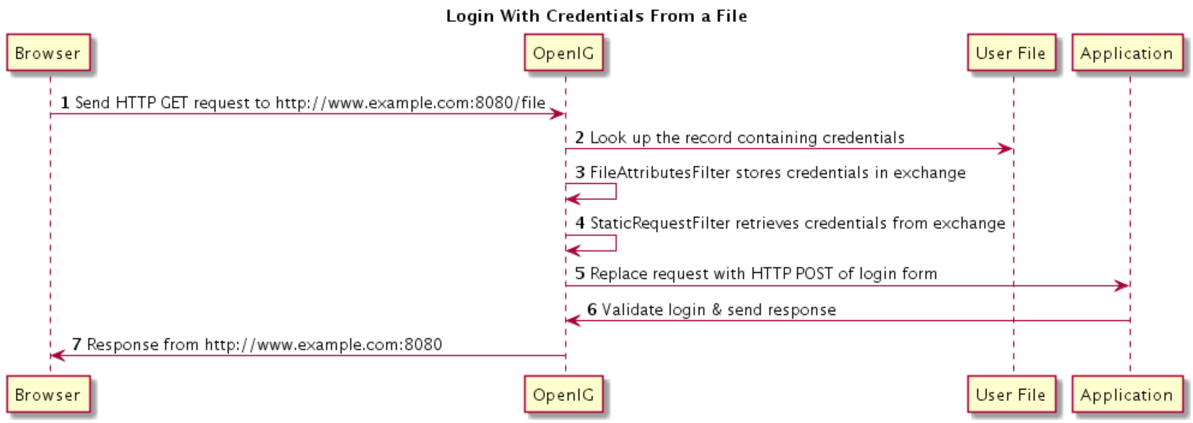
3. On Windows systems, edit the path name to the user file.

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

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

What's happening behind the scenes?

Figure 4.1.


Login With Credentials From a File
Browser
OpenIG
User File
Application
1 Send HTTP GET request to http://www.example.com:8080/file
2 Look up the record containing credentials
3 FileAttributesFilter stores credentials in exchange
4 StaticRequestFilter retrieves credentials from exchange
5 Replace request with HTTP POST of login form
6 Validate login & send response
7 Response from http://www.example.com:8080
Browser
OpenIG
User File
Application

OpenIG intercepts your browser's HTTP GET request. The request matches the new route configuration. The OpenIG "FileAttributesFilter" looks up credentials in a file, and stores the credentials it finds in the exchange. OpenIG then calls the next filter in the chain, "StaticRequestFilter", passing the exchange object that now holds the credentials. The "StaticRequestFilter" filter pulls the credentials out of the exchange, builds the login form, and performs the HTTP POST request to the HTTP server. The HTTP server validates the credentials, and responds with a profile page. OpenIG then passes the response from the HTTP server to your browser.

4.3. Login With Credentials From a Database

This sample shows you how to configure OpenIG to get credentials from H2. This sample was developed with Jetty and with H2 1.4.178.

Although this sample uses H2, OpenIG also works with other database software. OpenIG relies on the application server where it runs to connect to the database. Configuring OpenIG to retrieve data from a database is therefore a question of configuring the application server to connect to the database, and configuring OpenIG to choose the appropriate data source, and to send the appropriate SQL request to the database. As a result, the OpenIG configuration depends more on the data structure than on any particular database drivers or connection configuration.

Procedure 4.1. Preparing the Database

Follow these steps to prepare the database.

1. On the system where OpenIG runs, download and unpack H2 database.

2. Start H2.

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

   H2 starts, listening on port 8082, and opens a browser console page.

3. In the browser console page, select Generic H2 (Server) under Saved Settings. This sets the Driver Class, `org.h2.Driver`, the JDBC URL, `jdbc:h2:tcp://localhost/~/test`, the User Name, `sa`.

   In the Password field, type `password`.

   Then click Connect to access the console.

4. Run a statement to create a users table based on the user file from Section 4.2, "Login With Credentials From a File".

   If you have not created the user file on your system, put the following content in `/tmp/userfile`.

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

   Then create the users table through the H2 console:

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

On success, the table should contain the same users as the file. You can check this by running `SELECT * FROM users;` in the H2 console.

Procedure 4.2. Preparing Jetty's Connection to the Database

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

1. Configure Jetty for JNDI as described in the Jetty documentation on *Configuring JNDI*.

   For the version of Jetty used in this sample, stop Jetty and add the following lines to `/path/to/jetty/start.ini`.

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

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

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

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

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

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

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

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

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

Procedure 4.3. Preparing the OpenIG Configuration

Add a new route to the OpenIG configuration to look up credentials in the database.

- To add the route, add the following route configuration file as `$HOME/.openig/config/routes/03-sql.json`.

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

On Windows, the file name should be `%appdata%\OpenIG\config\routes\03-sql.json`.

Notice the following features of the new route.

- 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 the exchange stores the search results.
- The "StaticRequestFilter" retrieves the username and password from the exchange and replaces your browser's original HTTP GET request with an HTTP POST login request that contains the credentials to authenticate.

  Notice that the request is for `username, password`, and that H2 returns the fields as `USERNAME` and `PASSWORD`. The configuration reflects this difference.
- The route matches requests to `/sql`.

Procedure 4.4. To Try Logging In With Credentials From a Database

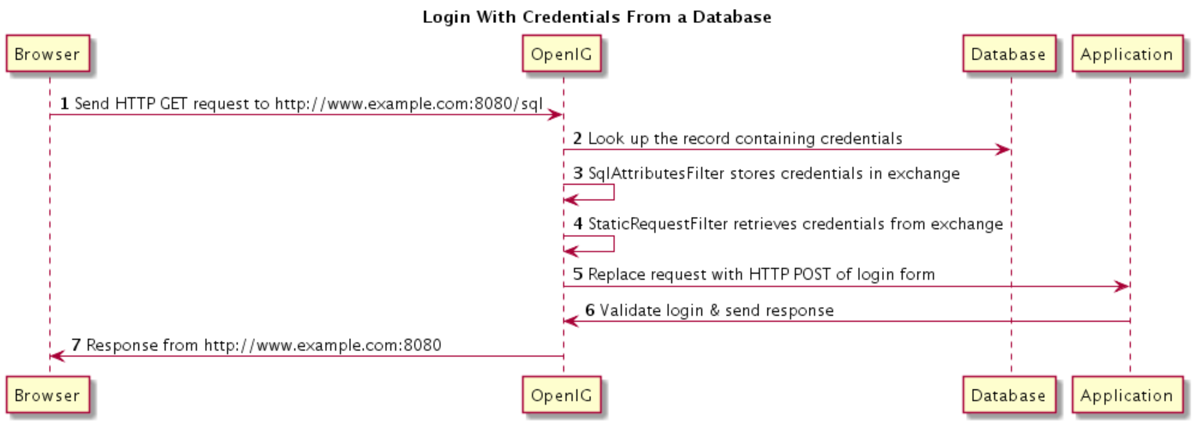
With H2, Jetty, and OpenIG correctly configured, you can try it out.

- Access the new route, http://www.example.com:8080/sql.

  OpenIG logs you in automatically as George.

What's happening behind the scenes?

Figure 4.2.


Login With Credentials From a Database
Browser
OpenIG
Database
Application
1 Send HTTP GET request to http://www.example.com:8080/sql
2 Look up the record containing credentials
3 SqlAttributesFilter stores credentials in exchange
4 StaticRequestFilter retrieves credentials from exchange
5 Replace request with HTTP POST of login form
6 Validate login & send response
7 Response from http://www.example.com:8080
Browser
OpenIG
Database
Application

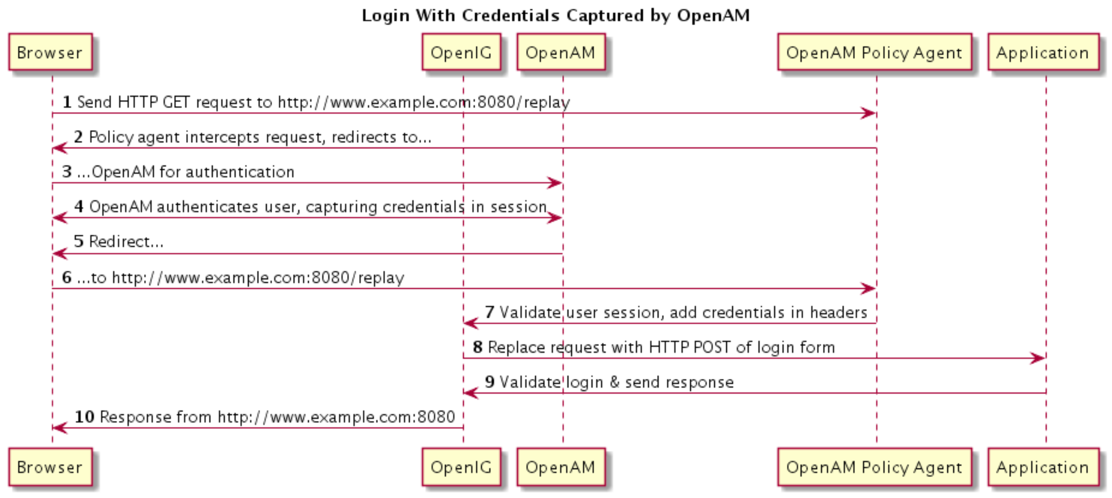
OpenIG intercepts your browser's HTTP GET request. The request matches the new route configuration. The OpenIG "SqlAttributesFilter" looks up credentials in H2, and stores the credentials it finds in the exchange. OpenIG then calls the next filter in the chain, "StaticRequestFilter", passing the exchange object that now holds the credentials. The "StaticRequestFilter" filter pulls the credentials out of the exchange, builds the login form, and performs the HTTP POST request to the HTTP server. The HTTP server validates the credentials, and responds with a profile page. OpenIG then passes the response from the HTTP server to your browser.

Chapter 5
Getting Login Credentials From OpenAM

This chapter walks you through an OpenAM integration with OpenAM's password capture and replay feature. This feature of OpenAM is typically used to integrate with Microsoft Outlook Web Access (OWA) or SharePoint by capturing the password during OpenAM authentication, encrypting it, and adding to the session, which is later decrypted and used for Basic Authentication to OWA or SharePoint. This tutorial shows how you can configure OpenIG to use the user name and password from the OpenAM Authentication to log the user an application. This is also how you would achieve OWA or SharePoint integration.

5.1. Detailed Flow

The figure below illustrates the flow of requests for a user who is not yet logged in to OpenAM accessing a protected application. After successful authentication, the user is logged into the application with the username and password from the OpenAM login session.


Login With Credentials Captured by OpenAM
Browser
OpenIG
OpenAM
OpenAM Policy Agent
Application
1 Send HTTP GET request to http://www.example.com:8080/replay
2 Policy agent intercepts request, redirects to...
3 ...OpenAM for authentication
4 OpenAM authenticates user, capturing credentials in session
5 Redirect...
6 ...to http://www.example.com:8080/replay
7 Validate user session, add credentials in headers
8 Replace request with HTTP POST of login form
9 Validate login & send response
10 Response from http://www.example.com:8080
Browser
OpenIG
OpenAM
OpenAM Policy Agent
Application

1. The user sends a browser request to access a protected application.
2. The OpenAM policy agent protecting OpenIG intercepts the request.
3. The policy agent redirects the browser to OpenAM.

4. OpenAM authenticates the user, capturing the login credentials, storing the password in encrypted form in the user's session.
5. After authentication, OpenAM redirects the browser...
6. ...back to the protected application.
7. The OpenAM policy agent protecting OpenIG intercepts the request, validates the user session with OpenAM (not shown here), adds the username and encrypted password to headers in the request, and passes the request to OpenIG.
8. OpenIG retrieves the credentials from the headers, and uses the username and decrypted password to replace the request with an HTTP POST of the login form.
9. The application validates the login credentials, and sends a response back to OpenIG.
10. OpenIG passes the response from the application back to the user's browser.

5.2. Setup Summary

This tutorial calls for you to set up several different software components.

- OpenAM is installed on `http://openam.example.com:8088/openam`.
- Download and run the minimal HTTP server .jar to use as the application to protect.

  The openig-doc-samples-3.1.0-jar-with-dependencies.jar application listens at `http://www.example.com:8081`. The minimal HTTP server is run with the **java -jar openig-doc-samples-3.1.0-jar-with-dependencies.jar** command, as described in the chapter on *Getting Started*.
- OpenIG is deployed in Jetty as described in the chapter on *Getting Started*. OpenIG listens at `http://www.example.com:8080`.
- OpenIG is protected by an OpenAM Java EE policy agent also deployed in Jetty. The policy agent is configured to add username and encrypted password headers to the HTTP requests.

5.3. Setup Details

In this section, it is assumed that you are familiar with the components involved. For OpenAM documentation, see https://backstage.forgerock.com/docs/am.

5.3.1. Setting Up OpenAM Server

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

Create a sample user Subject in the top level realm with username `george` and password `costanza`. Test that you can login to OpenAM with this username and password.

5.3.2. Preparing the Policy Agent Profile

Create the Java EE agent profile in the top level realm with the following settings:

- Server URL: `http://openam.example.com:8088/openam`
- Agent URL: `http://www.example.com:8080/agentapp`

Edit the policy agent profile to add these settings, making sure to save your work when you finish.

- On the Global settings tab page under General, change the Agent Filter Mode from `ALL` to `SSO_ONLY`.
- On the Application tab page under Session Attributes Processing, change the Session Attribute Fetch Mode from `NONE` to `HTTP_HEADER`.
- Also on the Application tab page under Session Attributes Processing, add `UserToken=username` and `sunIdentityUserPassword=password` to the Session Attribute Mapping list.

5.3.3. Configuring Password Capture

Configure password capture in OpenAM as follows.

- In the OpenAM console under Access Control > / (Top Level Realm) > Authentication, click All Core Settings, and then add `com.sun.identity.authentication.spi.ReplayPasswd` to the Authentication Post Processing Classes.
- Run OpenAM's **com.sun.identity.common.DESGenKey** command to generate a shared key for the OpenAM Authentication plugin and for OpenIG.

  To run this command using the **java** command, you must add OpenAM .jar file libraries to the Java class path. The library names depend on the version of OpenAM that you use.

  - When using OpenAM 12.0.0, the libraries are `forgerock-util-1.3.5.jar` `openam-core-12.0.0.jar`, and `openam-shared-12.0.0.jar`.

    As an example, if OpenAM 12.0.0 is installed in Apache Tomcat under `/openam` you would run the command **java -classpath /path/to/tomcat/webapps/openam/WEB-INF/lib/forgerock-util-1.3.5.jar:/path/to/tomcat/webapps/openam/WEB-INF/lib/openam-core-12.0.0.jar:/path/to/tomcat/webapps/openam/WEB-INF/lib/openam-shared-12.0.0.jar com.sun.identity.common.DESGenKey**.

  - When using OpenAM 11.0.0 for example, the libraries are `forgerock-util-1.1.0.jar` `openam-core-11.0.0.jar`, and `openam-shared-11.0.0.jar`.

    As an example, if OpenAM 11.0.0 is installed in Apache Tomcat under `/openam` you would run the command **java -classpath /path/to/tomcat/webapps/openam/WEB-INF/lib/forgerock-util-1.1.0.jar:/**

path/to/tomcat/webapps/openam/WEB-INF/lib/openam-core-11.0.0.jar:/path/to/tomcat/webapps/openam/WEB-INF/lib/openam-shared-11.0.0.jar com.sun.identity.common.DESGenKey.

- When using OpenAM 10 and earlier, the libraries are `amserver.jar` and `opensso-sharedlib.jar`.

  As an example, if OpenAM 10 is installed in Apache Tomcat under `/openam` you would run the command **java -classpath /path/to/tomcat/webapps/openam/WEB-INF/lib/amserver.jar:/path/to/tomcat/webapps/openam/WEB-INF/lib/opensso-sharedlib.jar com.sun.identity.common.DESGenKey**.

The output of the command shows the generated key, as in the following example for OpenAM 11.0.0.

```
$ cd /path/to/tomcat/webapps/openam/WEB-INF/lib
$ java -classpath \
 forgerock-util-1.1.0.jar:openam-core-11.0.0.jar:openam-shared-11.0.0.jar \
 com.sun.identity.common.DESGenKey
Key ==> ipvvZF2Mj0k
```

- In the OpenAM console under Configuration > Servers and Sites, click on the server name link, go to the Advanced tab and add `com.sun.am.replaypasswd.key` with the value of the key generated in the previous step.

  Restart the OpenAM server after adding the Advanced property for the change to take effect.

5.3.4. Installing OpenIG

Install OpenIG in Jetty and run the minimal HTTP server as described in the chapter on *Getting Started*.

When you finish, OpenIG should be running in Jetty, configured to access the minimal HTTP server as described in that chapter.

The initial OpenIG configuration file should look like the one used to proxy requests through to the HTTP server and to capture request and response data, as you used in the chapter on *Getting Started*.

To test your setup, access the HTTP server home page through OpenIG at http://www.example.com:8080. Login as username `george`, password `costanza`. You should see a page showing the username and some information about the request.

5.3.5. Installing the Policy Agent

Install the OpenAM Java EE policy agent alongside OpenIG in Jetty, listening at `http://www.example.com:8080`, using the following hints.

- Jetty Server Config Directory : `/path/to/jetty/etc`

- Jetty installation directory. : `/path/to/jetty`
- OpenAM server URL : `http://openam.example.com:8088/openam`
- Agent URL : `http://www.example.com:8080/agentapp`
- After copying `agentapp.war` into `/path/to/jetty/webapps/`, also add the following filter configuration to `/path/to/jetty/etc/webdefault.xml`.

```
<filter>
  <filter-name>Agent</filter-name>
  <display-name>Agent</display-name>
  <description>OpenAM Policy Agent Filter</description>
  <filter-class>com.sun.identity.agents.filter.AmAgentFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>Agent</filter-name>
  <url-pattern>/replay</url-pattern>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>ERROR</dispatcher>
</filter-mapping>
```

To test the configuration, start Jetty, and then browse to http://www.example.com:8080/replay. You should be redirected to OpenAM for authentication.

Do not log in, however. You have not yet configured a route to handle requests to `/replay`.

5.3.6. Configuring OpenIG

Add a new route to the OpenIG configuration to handle OpenAM password capture & replay.

To add the route, add the following route configuration file as `$HOME/.openig/config/routes/04-replay.json`.

```
{
    "handler": {
        "type": "Chain",
        "config": {
            "filters": [
                {
                    "type": "CryptoHeaderFilter",
                    "config": {
                        "messageType": "REQUEST",
                        "operation": "DECRYPT",
                        "algorithm": "DES/ECB/NoPadding",
                        "key": "DESKEY",
                        "keyType": "DES",
                        "charSet": "utf-8",
                        "headers": [
                            "password"
                        ]
                    }
                },
                {
                    "type": "StaticRequestFilter",
                    "config": {
                        "method": "POST",
                        "uri": "http://www.example.com:8081",
                        "form": {
                            "username": [
                                "${exchange.request.headers['username'][0]}"
                            ],
                            "password": [
                                "${exchange.request.headers['password'][0]}"
                            ]
                        }
                    }
                },
                {
                    "type": "HeaderFilter",
                    "config": {
                        "messageType": "REQUEST",
                        "remove": [
                            "password",
                            "username"
                        ]
                    }
                }
            ],
            "handler": "ClientHandler"
        }
    },
    "condition": "${matches(exchange.request.uri.path, '^/replay')}"
}
```

On Windows, the file name should be `%appdata%\OpenIG\config\routes\04-replay.json`.

Change `DESKEY` to the actual key value that you generated in Section 5.3.3, "Configuring Password Capture".

Notice the following features of the new route.

- The "CryptoHeaderFilter" decrypts the password that OpenAM captured and encrypted, and that the OpenAM policy agent included in the headers for the request.

  The resulting "CryptoHeaderFilter" should look something like this, but using the "key" value that you generated:

```
{
    "type": "CryptoHeaderFilter",
    "config": {
        "messageType": "REQUEST",
        "operation": "DECRYPT",
        "algorithm": "DES/ECB/NoPadding",
        "key": "ipvvZF2Mj0k",
        "keyType": "DES",
        "charSet": "utf-8",
        "headers": [
            "password"
        ]
    }
}
```

- The "StaticRequestFilter" retrieves the username and password from the exchange and replaces your browser's original HTTP GET request with an HTTP POST login request that contains the credentials to authenticate.
- The "HeaderFilter" removes the username and password headers before continuing to process the exchange.
- The route matches requests to `/replay`.

5.4. Trying It Out

Log out of OpenAM if you are logged in already.

Access the new route, http://www.example.com:8080/replay.

If you are not already logged into OpenAM you should be redirected to the OpenAM login page. Login with username `george`, password `costanza`. After login you should be redirected back to the application.

Chapter 6 OpenIG as a SAML 2.0 Service Provider

This chapter has two aims. First, it aims to help you understand how OpenIG works as a SAML 2.0 service provider, and what that entails in terms of setup and configuration. Second, it aims to show you how to configure OpenIG as a SAML 2.0 federation service provider, logging users in to a protected application with information from a SAML assertion.

6.1. About SAML 2.0 Federation

The Federation component of OpenIG is a standards based authentication service used by OpenIG to validate a user and retrieve key attributes of the user in order to log them in to applications that OpenIG protects. The Federation component implements Security Assertion Markup Language 2.0.

Security Assertion Markup Language (SAML) 2.0 is a standard for exchanging security information across organizational boundaries. SAML 2.0 enables web single sign-on (SSO), for example, where the service managing the user's identity does not necessarily belong to the same organization and does not necessarily use the same software as the service that the user wants to access.

In SAML 2.0, the service managing the user's identity is called the *Identity Provider* (IDP). The service that the user wants to access is called the *Service Provider* (SP). Provider organizations agree on the security information they want to exchange, and then they mutually configure access to each others' services, so that the SAML 2.0 federation capability is ready for use. The group of providers sets up a *circle of trust*, which is a list of services participating in the federation. In order to be able to configure access to services in the circle of trust, the providers share SAML 2.0 *metadata* describing their services in an XML format defined by the SAML 2.0 standard.

OpenIG plays the role of SAML 2.0 SP. You must therefore configure OpenIG as SP to access IDP services in order for the Federation component to be operational.

For SAML 2.0 web SSO, the user authenticates with the IDP. This can start either with the user visiting the IDP site and logging in, or with the user visiting the SP site and being directed to the IDP to log in. On successful authentication, the IDP sends an assertion statement about the authentication to the SP. This assertion statement attests which user the IDP authenticated, when the authentication succeeded, how long the assertion is valid, and so forth. It can optionally contain attribute values for the user who authenticated. (OpenIG can then, for example, use the attribute values to log a user into a protected application.) The assertion can optionally be signed and encrypted.

There are two ways that the OpenIG federation component can be invoked:

1. IDP initiated SSO, where the remote Identity Provider sends an unsolicited authentication statement to OpenIG

2. SP initiated SSO, where OpenIG calls the Federation component to initiate federated SSO with the Identity Provider

In both cases, the job of the Federation component is to validate the user and to pass the required attributes to OpenIG so that it can log the user into protected applications.

6.2. Installation Overview

This section summarizes the steps needed to prepare OpenIG to act as a SAML 2.0 SP for your target application.

- Install the OpenIG war file.
- Configure OpenIG to proxy successfully, and even log a user in, to the target application. Getting this to work before configuring Federation makes the process much simpler to troubleshoot if anything goes wrong.
- Add Federation configuration to the OpenIG configuration.
- Include the assertion mapping, redirect URI, and any optional configuration settings you choose in the Federation configuration.
- Export the Identity Provider metadata from the remote IDP, or use the metadata from an OpenAM-generated Fedlet. (An OpenAM Fedlet is a small web application that can act as SP.)
- Import OpenIG metadata to your Identity Provider.

If you intend to protect multiple service provider applications first read this chapter and work through the samples. Then consider the explanation in the appendix, *SAML 2.0 & Multiple Applications*.

6.3. Configuration File Overview

You configure the Federation component by modifying both the OpenIG `config.json` file and also by including Federation-specific XML files with the configuration.

The location of configuration information depends on the operating system where OpenIG runs, and on the user who runs the application server where OpenIG runs.

- On UNIX, Linux, and similar systems, where this user's home directory is referred to as `$HOME`, by default the Federation component looks in `$HOME/.openig/config` for `config.json` and in `$HOME/.openig/SAML` for the Federation XML configuration.
- On Windows, by default the Federation component looks in `%appdata%\OpenIG\config`, and in `%appdata%\OpenIG\SAML`. To locate the `%appdata%` folder for your version of Windows, open Windows Explorer, type `%appdata%` as the file path, and press Enter. You must create the `%appdata%\OpenIG\config` and `%appdata%\OpenIG\SAML` folders, and then copy the configuration files into the folders.

The following is a description of the files:

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

This is the core configuration file for OpenIG, where you configure a SamlFederationHandler in the *Reference*. If this file uses a Router in the *Reference*, you can configure the handler in a route file.

You must configure both the OpenIG core configuration, and also the XML files specific to the Federation component. The reason there are two sets of configuration files is that the Federation component includes a federation library from OpenAM.

In order to configure the Federation component you must tag swap the XML files. If you are familiar with the workflow in the OpenAM console you can instead generate a Fedlet and directly copy the configuration files into `$HOME/.openig/SAML`.

`$HOME/.openig/SAML/FederationConfig.properties`

Advanced features of the federation library from OpenAM. The defaults suffice in most deployments.

`$HOME/.openig/SAML/fedlet.cot`

Circle of trust for OpenIG and the Identity Provider.

`$HOME/.openig/SAML/idp.xml`

This metadata file is generated by the Identity Provider. You must copy the generated metadata file into the configuration directory.

`$HOME/.openig/SAML/idp-extended.xml`

Standard metadata extensions generated by the Identity Provider.

`$HOME/.openig/SAML/sp.xml`
`$HOME/.openig/SAML/sp-extended.xml`

These are the standard metadata and metadata extensions for the OpenIG Federation component.

6.4. Configuring the Federation Handler

The simplest way to configure the Federation component is to use the OpenAM task wizard to generate a Fedlet, and then copy the Fedlet configuration files to the correct locations. If you use the Fedlet configuration files, simply unpack `Fedlet.war` and copy all the files listed above into `$HOME/.openig/SAML`. You do not have to modify the files to do basic IDP and SP initiated SSO with OpenIG. When generating a Fedlet, know that the sample `config.json` templates uses `/saml` as the URI so your Fedlet end point should be specified as `*protocol*://*host.domain*:*port*/saml`.

If you do not use the Fedlet wizard, edit the configuration files for the unconfigured Fedlet, and then copy the Fedlet configuration files to the `$HOME/.openig/SAML` directory. You must still nevertheless get the metadata from the IDP, and then copy it to `idp.xml` in the same directory.

Once you have the Fedlet configuration files set up, add the SamlFederationHandler in the *Reference* object to the OpenIG configuration.

6.5. Example Settings

Application `myportal` requires a form with username and password for login. The username for `myportal` is the `mail` attribute at the user's Identity Provider. The password for `myportal` is the `mailPassword` attribute at the Identity Provider.

The incoming SAML2 assertion sent by the Identity Provider contains the `mail` and `mailPassword` attributes. The Federation component validates the incoming assertion, sets the session attributes `username` and `password` to the values of `mail` and `mailPassword` from the assertion attributes, and redirects the user to `/myportal/login`. A "LoginRequest" filter then retrieves the credentials and creates the form to log the user in to `myportal`.

The "SamlFederationHandler" configuration object looks like this:

```
{
    "name": "SamlFederationHandler",
    "type": "org.forgerock.openig.saml.SamlFederationHandler",
    "config": {
        "assertionMapping": {
            "username": "mail",
            "password": "mailPassword"
        },
        "redirectURI": "/myportal/login",
        "logoutURI": "/myportal/logout"
    }
}
```

The "LoginRequest" configuration object looks like this:

```
{
    "name": "LoginRequest",
    "type": "StaticRequestFilter",
    "config": {
        "method": "POST",
        "uri": "https://www.myportal.com/myportal/login",
        "form": {
            "username": [
                "${exchange.session.username}"
            ],
            "password": [
                "${exchange.session.password}"
            ]
        }
    }
}
```

6.6. Identity Provider Metadata

The Identity Provider metadata must be copied to the `$HOME/.openig/SAML/idp.xml` directory. See the documentation for your Identity Provider for instructions on how to get the metadata.

To export Identity Provider metadata from OpenAM, either save the response from the appropriate end point, such as `http://openam.example.com:8088/openam/saml2/jsp/exportmetadata.jsp`, or run an **ssoadm** command such as the following:

```
$ ssoadm \
 export-entity \
 --adminid amadmin \
 --password-file /tmp/pwd.txt \
 --entityid http://openam.example.com:8088/openam \
 --meta-data-file /tmp/idp.xml
```

6.7. Preparing to Try OpenIG as a SAML 2.0 Service Provider

The following sections in this chapter are a tutorial on setting up OpenAM to send a SAML 2.0 assertion to OpenIG containing user credentials, and OpenIG to validate the assertion and use the credentials to log the user in to the protected application.

Before you start this tutorial, prepare OpenIG and the minimal HTTP server as you did for the chapter on *Getting Started*.

OpenIG should be running in Jetty, configured to access the minimal HTTP server as described in that chapter.

The initial OpenIG configuration file should look like the one used to proxy requests through to the HTTP server and to capture request and response data, as you used in the chapter on *Getting Started*.

To test your setup, access the HTTP server home page through OpenIG at http://www.example.com:8080. Login as username `george`, password `costanza`. You should see a page showing the username and some information about the request.

In this tutorial, it is assumed that you are familiar with SAML 2.0 federation and with the components involved, including OpenAM. For OpenAM documentation, see https://backstage.forgerock.com/docs/am.

6.8. Configuring OpenAM

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

Login to the OpenAM console as administrator, and use the common task wizard to create a hosted Identity Provider. This tutorial does not address PKI configuration for validation and encryption, though OpenIG is capable of handling both when properly configured, just as any OpenAM Fedlet can handle both. Configure the Attribute Mapping to map the the `mail` attribute to `mail` and the `employeenumber` attribute to `employeenumber`. You can use the `test` certificate in the Identity Provider configuration for signing in this example.

Then use the common task wizard to create a Fedlet. Set the Name to `OpenIG`. Set the Destination URL to `http://www.example.com:8080/saml`. Also configure the Attribute Mapping for the Fedlet to map the the `mail` attribute to `mail` and the `employeenumber` attribute to `employeenumber`.

Why map these attributes? The SAML 2.0 attribute mapping indicates that the SP, OpenIG, wants the IDP, OpenAM in this case, to get the values of these attributes from the user profile and then send them to the SP, OpenIG. OpenIG can then use the values of the attributes, in this case `mail` and `employeenumber`, to log the user in to the application it protects.

This tutorial uses `mail` and `employeenumber` for the sake of simplicity. Both of those attributes are part of a user's profile out of the box with the default OpenAM configuration. Neither of the attributes are needed for anything else in this tutorial. So this tutorial uses `mail` to hold the username, and `employeenumber` to hold the password. In a real deployment, you would no doubt use other attributes that depend on how the real user profiles are configured.

Use the OpenAM console to create a user subject in the top level realm with Email Address `george` and Employee Number `costanza`.

6.9. Configuring OpenIG For Federation

Unpack the configuration files from the Fedlet you created in Section 6.8, "Configuring OpenAM". The Fedlet is packaged as a .zip file that contains a war file that in turn contains the configuration files to unpack. OpenAM displays the location of the .zip file upon successful creation of the Fedlet. If you followed the instructions above, the .zip is `$HOME/openam/myfedlets/OpenIG/Fedlet.zip` on the system where OpenAM runs.

```
$ cd $HOME/openam/myfedlets/OpenIG
$ unzip Fedlet.zip fedlet.war
$ unzip fedlet.war conf/*
$ mkdir $HOME/.openig/SAML
$ cp conf/* $HOME/.openig/SAML
$ ls -1 $HOME/.openig/SAML
FederationConfig.properties
fedlet.cot
idp-extended.xml
idp.xml
sp-extended.xml
sp.xml
```

On Windows, the SAML configuration files belong in `%appdata%\OpenIG\SAML`. To locate the `%appdata%` folder for your version of Windows, open Windows Explorer, type `%appdata%` as the file path, and press Enter.

Restart Jetty after preparing the SAML configuration files.

Add two new routes to the OpenIG configuration.

- Add a route that injects credentials into the exchange based on attribute values from the SAML assertion returned on successful authentication.

  The configuration file to add in this case is `$HOME/.openig/config/routes/05-saml.json`

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

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

  Notice the following features of the new route.

- The "SamlFederationHandler" extracts credentials from the attributes returned in the SAML 2.0 assertion. It then redirects to the `/federate` route.

- The route matches requests to `/saml`.

- The route uses the "JwtSession" session 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 the reference for JwtSession in the *Reference*.

• Add a route that handles requests to perform SAML federation.

The configuration file to add in this case is `$HOME/.openig/config/routes/05-federate.json`

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

```
                              ],
                              "handler": "ClientHandler"
                          }
                      },
                      "baseURI": "http://www.example.com:8081"
                  }
              ]
          }
      },
      "condition": "${matches(exchange.request.uri.path, '^/federate')}",
      "session": "JwtSession"
}
```

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

Notice the following features of the new route.

- The "DispatchHandler" dispatches requests to the "StaticResponseHandler" if the username has not yet been populated in the exchange, meaning the user has not yet authenticated with the IDP. Otherwise, if the credentials have been inserted into the exchange, the "DispatchHandler" dispatches requests to the "Chain" to log the user in to the protected application.
- The "StaticResponseHandler" redirects to the Service Provider initiated single sign-on end point to initiate SAML 2.0 web browser SSO. After authentication is successful and the "SamlFederationHandler" has injected credentials into the exchange, the user-agent ends up redirected to this same route.
- The "StaticRequestFilter" retrieves the username and password from the exchange and replaces your browser's original HTTP GET request with an HTTP POST login request that contains the credentials to authenticate.
- The route matches requests to `/federate`. This is the route you use to test the configuration.
- The route also uses the "JwtSession" session implementation.

6.10. Trying It Out

Log out of OpenAM console, and then test whether everything is properly configured.

- For IDP initiated SSO, click this IDP initiated SSO link, and then login to OpenAM with username `george`, password `costanza`.
- For SP initiated SSO, either browse to the URL for the new route, at http://www.example.com:8080/federate, or click this SP initiated SSO link, and then login to OpenAM with username `george`, password `costanza`.

However you initiate single sign-on, you should wind up viewing the page you normally see after logging in.

What is happening behind the scenes?

The initial incoming requests matches the `/federate` route. As the user is not yet authenticated, the "SPInitiatedSSORedirectHandler" sends a redirect to initiate SSO.

The user authenticates with the Identity Provider for SAML 2.0 single sign-on. After authentication, the Identity Provider redirects the user-agent back to the SAML URI on the Service Provider (OpenIG), which you configured for the Fedlet as `/saml`. The "SamlFederationHandler" gets the request to this route. The request holds the SAML 2.0 assertion whose attributes contain credentials.

The "SamlFederationHandler" processes an incoming SAML 2.0 assertion, injecting credentials values from the assertion into the exchange session. The "SamlFederationHandler" then redirects to the `/federate` route.

On the `/federate` route, once the attributes from the assertion are set in the session, OpenIG dispatches the exchange to the "Chain". The "StaticRequestFilter" in the "Chain" uses the attribute values to replace the request with an HTTP POST of login form data to log the user in to the protected application.

OpenIG returns the response page showing that the user has logged in.

Chapter 7

OpenIG as an OAuth 2.0 Resource Server

This chapter explains how OpenIG acts as an OAuth 2.0 Resource Server, and follows with a tutorial that shows you how to use OpenIG as a resource server.

7.1. About OpenIG as an OAuth 2.0 Resource Server

The OAuth 2.0 Authorization Framework describes a way of allowing a third-party application to access a user's resources without having the user's credentials. When resources are protected with OAuth 2.0, users can use their credentials with an OAuth 2.0-compliant identity provider, such as OpenAM, Facebook, Google and others to access the resources, rather than setting up an account with yet another third-party application.

In OAuth 2.0, there are four entities involved.

- The *resource owner* is the end user who owns protected resources on a resource server.

  For example, a resource owner has photos stored in a web service.

- The *resource server* provides the user's protected resources to authorized client applications.

  In OAuth 2.0, an authorization server grants the client application authorization based on the resource owner's consent.

  For example, a web service holds user's photos.

- The *client* is the application that needs access to the protected resources.

  For example, a photo printing service needs access to the user's photos.

- The *authorization server* is the service responsible for authenticating resource owners and obtaining their consent to allow client applications to access their resources.

  For example, OpenAM can act as the OAuth 2.0 authorization server to authenticate resource owners and obtain their consent. Other services can play this role as well. Google and Facebook for example provide OAuth 2.0 authorization services.

In OAuth 2.0, there are different grant mechanisms whereby the client can obtain authorization. One grant mechanism involves the client redirecting the resource owner's browser to the authorization server to complete authentication and authorization. You might have experienced this grant mechanism yourself when logging in with your identity provider account to access a web service, rather than creating a new account directly with the web service. Whatever the grant mechanism, the client's aim is to get an OAuth 2.0 *access token* from the authorization server.

Access tokens are the credentials used to access protected resources. An access token is just a string that represents the authorization to access protected resources given by the authorization server. An access token, like cash, is a bearer token. This means that anyone who has the access token can use it to get the resources. Access tokens therefore must be protected, so requests involving them must go over HTTPS. 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.

When the client requests access to protected resources, it supplies the access token to the resource server housing the resources. The resource server must then validate the access token. If the access token is found to be valid, then the resource server can let the client have access to the resources.

When OpenIG acts therefore as an OAuth 2.0 resource server, its role is to validate access tokens. How an access token is validated is technically not covered in the specifications for OAuth 2.0. Typically the resource server validates an access token by submitting the token to a token information endpoint. The token information endpoint typically returns the access token, when it expires, and the OAuth 2.0 *scopes* associated with the token, potentially with other information. In OAuth 2.0, the token scopes are strings that can identify the scope of access authorized to the client, but can also be used for other purposes. For example, OpenAM maps them to user profile attribute values by default, and also allows custom scope handling plugins.

In the tutorial that follows, you configure OpenIG as a resource server, and use OpenAM as the OAuth 2.0 authorization server.

7.2. Preparing the Tutorial

In the chapter on *Getting Started*, you learned how to configure OpenIG to proxy traffic and capture request and response data. You also learned how to configure OpenIG to use a static request to log in with hard-coded credentials.

This tutorial shows you how OpenIG can act as an OAuth 2.0 resource server, validating OAuth 2.0 access tokens and including token info in the exchange.

This tutorial relies on OpenAM as an OAuth 2.0 authorization server. As an OAuth 2.0 client of OpenAM, you get an access token. You then submit the access token to OpenIG, and OpenIG acts as the resource server.

Before you start this tutorial, prepare OpenIG and the minimal HTTP server as you did for the chapter on *Getting Started*.

OpenIG should be running in Jetty, configured to access the minimal HTTP server as described in that chapter.

7.3. Setting Up OpenAM as an Authorization Server

Install and configure OpenAM on `http://openam.example.com:8088/openam` with the default configuration. If you use a different configuration, make sure you substitute in the tutorial accordingly. Although this tutorial does not use HTTPS, you must use HTTPS to protect access tokens in production environments.

Login to the OpenAM console as administrator, and use the common task wizard, Configure OAuth2, to configure an OAuth 2.0 authorization server in / (Top Level Realm).

Also create an OAuth 2.0 Client profile in / (Top Level Realm). This allows you to request an OAuth 2.0 access token on behalf of the client. In OpenAM console, browse to Access Control > / (Top Level Realm) > Agents > OAuth 2.0 Client, and then click New in the Agent table.

Create an OAuth 2.0 client profile with name `OpenIG` and password `password`. The name is the OAuth 2.0 client_id, and the password is the client_secret.

Edit the `OpenIG` client profile to add `mail` and `employeenumber` scopes to the Scope(s) list, and then save your work. In this tutorial, you overload these profile settings to pass credentials to OpenIG. This tutorial uses `mail` and `employeenumber` for the sake of simplicity. Both of those attributes are part of a user's profile out of the box with the default OpenAM configuration. Neither of the attributes are needed for anything else in this tutorial. So this tutorial uses `mail` to hold the username, and `employeenumber` to hold the password. In a real deployment, you would no doubt use other attributes that depend on how the real user profiles are configured.

Finally, create a user whose additional credentials you set in the Email Address and Employee Number fields if you have not already done so for another tutorial.

1. In OpenAM console, under Access Control > / (Top Level Realm) > Subjects > User, click New to create the user profile.

2. Set the ID to `george`, the password to `costanza`, and fill the other required fields as you like before clicking OK.

3. Click the user name to edit the profile again, setting Email Address to `george` and Employee Number to `costanza` before clicking Save.

4. When finished, log out of OpenAM console.

7.4. Configuring OpenIG as a Resource Server

To configure OpenIG as an OAuth 2.0 resource server, you use an OAuth2ResourceServerFilter in the *Reference*.

The filter expects an OAuth 2.0 access token in an incoming `Authorization` request header, such as the following.

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

The filter then uses the access token to validate the token and to retrieve token information from the authorization server. On successful validation, the filter injects the response from the authorization server into the location set by the "target" in the configuration.

If no access token is present in the request, or token validation does not complete successfully, the filter returns an HTTP error status to the user-agent, and OpenIG does not continue processing the exchange. This is done as specified in the RFC, OAuth 2.0 Bearer Token Usage.

You can therefore add additional filters and handlers to the chain directly after the `OAuth2ResourceServerFilter`, and expect to have the access token if the filter completes successfully.

To configure OpenIG as an OAuth 2.0 resource server, add a new route to the OpenIG configuration, by including the following route configuration file as `$HOME/.openig/config/routes/06-rs.json`.

```
{
    "handler": {
        "type": "Chain",
        "config": {
            "filters": [
                {
                    "type": "OAuth2ResourceServerFilter",
                    "config": {
                        "providerHandler": "ClientHandler",
                        "scopes": [
                            "mail",
                            "employeenumber"
                        ],
                        "tokenInfoEndpoint":
                            "http://openam.example.com:8088/openam/oauth2/tokeninfo",
                        "requireHttps": false,
                        "target": "${exchange.token}"
                    },
                    "timer": true
                },
                {
                    "type": "ScriptableFilter",
                    "config": {
                        "type": "application/x-groovy",
                        "source":
                            "import org.forgerock.json.fluent.JsonValue;
                             logger.info(exchange.token.asJsonValue() as String);
                             exchange.username = exchange.token.info.mail;
                             exchange.password = exchange.token.info.employeenumber;
                             next.handle(exchange)"
                    },
                    "timer": true
                },
                {
                    "type": "StaticRequestFilter",
                    "config": {
                        "method": "POST",
                        "uri": "http://www.example.com:8081",
                        "form": {
                            "username": [
                                "${exchange.username}"
                            ],
                            "password": [
                                "${exchange.password}"
                            ]
                        }
                    },
                    "timer": true
                }
            ],
            "handler": "ClientHandler"
        }
    },
    "condition": "${matches(exchange.request.uri.path, '^/rs')}",
    "timer": true
}
```

On Windows, the file name should be `%appdata%\OpenIG\config\routes\06-rs.json`.

Notice the following features of the new route.

- The "OAuth2ResourceServerFilter" includes a client handler to send access token validation requests, the list of required scopes that the filter expects to find in access tokens, the OpenAM token info endpoint used to validate access tokens, and `"requireHttps": false` to allow testing without having to set up keys and certificates. (In production environments, do use HTTPS to protect access tokens.)

  After successfully using the token info endpoint to validate an access token, the "OAuth2ResourceServerFilter" injects data from the response into `exchange.token`.

- After the "OAuth2ResourceServerFilter" has injected information for a valid access token into the exchange, the "ScriptableFilter" dumps the token information to the log. The "ScriptableFilter" also injects the credentials from the user profile in OpenAM into the exchange.

- The "StaticRequestFilter" retrieves the username and password from the exchange and replaces the original HTTP GET request with an HTTP POST login request that contains the credentials to authenticate.

- The route matches requests to `/rs`.

7.5. Trying It Out

To try your configuration, you need an access token. Get an access token from OpenAM and use it to access OpenIG as in the following example, which uses the OAuth 2.0 resource owner password credentials authorization grant.

```
$ curl \
 --user "OpenIG:password" \
 --data "grant_type=password&username=george&password=costanza&scope=mail%20employeenumber" \
 http://openam.example.com:8088/openam/oauth2/access_token
{
    "scope": "mail employeenumber",
    "expires_in": 60,
    "token_type": "Bearer",
    "refresh_token": "80963b0e-8283-434b-ba11-ce01ef0e93b6",
    "access_token": "ddf31dac-e23a-446c-bd21-db60cf19b9f3"
}

$ curl \
 --header "Authorization: Bearer ddf31dac-e23a-446c-bd21-db60cf19b9f3" \
 http://www.example.com:8080/rs
...
    <h1>User Information</h1>

    <dl>
        <dt>Username</dt>
        <dd>george</dd>
    </dl>

    <h1>Request Information</h1>

    <dl>
        <dt>Method</dt>
        <dd>POST</dd>

        <dt>URI</dt>
        <dd>/</dd>

        <dt>Headers</dt>
        <dd style="font-family: monospace; font-size: small;">...</dd>
    </dl>
```

Also look in the Jetty server output to see the access token information. The access token information looks something like the following.

```
TUE DEC 02 17:14:28 CET 2014 (INFO) {ScriptableFilter}/handler/config/filters/1
{
    "mail": "george",
    "employeenumber": "costanza",
    "scope": [
        "mail",
        "employeenumber"
    ],
    "grant_type": "password",
    "realm": "/",
    "token_type": "Bearer",
    "expires_in": 41,
    "access_token": "e362515f-ecf2-47b7-b1a7-c6480e705129"
}
```

What is happening behind the scenes?

After OpenIG gets the **curl** request, the resource server filter validates the access token with OpenAM, and injects the token information into the exchange. (If the access token was missing or invalid, then the resource server filter would have returned an error status to the user-agent. The OAuth 2.0 client would then have had to deal with the error.)

The "ScriptableFilter" logs the token information, and also extracts the credentials to inject them into the exchange. Finally the "StaticRequestFilter" uses the credentials to log the user in to the minimal HTTP server, which responds with the User Information page.

Chapter 8 OpenIG as an OAuth 2.0 Client

This chapter explains how OpenIG acts as an OAuth 2.0 client or OpenID Connect 1.0 relying party, and follows with a tutorial that shows you how to use OpenIG as an OpenID Connect 1.0 relying party.

8.1. About OpenIG as an OAuth 2.0 Client

As described in the chapter, *OpenIG as an OAuth 2.0 Resource Server*, an OAuth 2.0 client is the third-party application that needs access to a user's protected resources. The client application therefore has the user (the OAuth 2.0 resource owner) delegate authorization by authenticating with an identity provider (the OAuth 2.0 authorization server) using an existing account, and then consenting to authorize access to protected resources (on an OAuth 2.0 resource server).

OpenIG can act as an OAuth 2.0 client when you configure an OAuth2ClientFilter in the *Reference*. The filter handles the process of allowing the user to select a provider, and redirecting the user through the authentication and authorization steps of an OAuth 2.0 authorization code grant, which results in the authorization server returning an access token to the filter. At the outcome of a successful authorization grant, the filter injects the access token data into a configurable target field of the exchange so that subsequent filters and handlers have access to the access token. Subsequent requests can use the access token without re-authentication.

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

8.2. About OpenIG as an OpenID Connect 1.0 Relying Party

The specifications available through the OpenID Connect site describe an authentication layer built on OAuth 2.0, which is OpenID Connect 1.0.

OpenID Connect 1.0 is a specific implementation of OAuth 2.0 where the identity provider holds the protected resource that the third-party application aims to access. This resource is the *UserInfo*, information about the authenticated end-user expressed in a standard format.

In OpenID Connect 1.0, the key entities are the following.

- The *end user* (OAuth 2.0 resource owner) whose user information the application needs to access.

The end user wants to use an application through existing identity provider account without signing up to and creating credentials for yet another web service.

- The *Relying Party* (RP) (OAuth 2.0 client) 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.

- The *OpenID Provider* (OP) (OAuth 2.0 authorization server and also resource server) that holds the user information and grants access.

  The OP effectively has the end user consent to providing the RP with access to some of its user information. As OpenID Connect 1.0 defines unique identification for an account (subject identifier + issuer identifier), the RP can use this as a key to its own user profile.

  In the case of the online mail application, this key could be used to access the mailboxes and related account information. In the case of the online shopping site, this key could be used to access the offerings, account, shopping cart and so forth. The key makes it possible to serve users as if they had local accounts.

When OpenIG acts therefore as an OpenID Connect 1.0 relying party, its ultimate role is to retrieve user information from the OpenID provider, and then to inject that information into the exchange for use by subsequent filters and handlers.

In the tutorial that follows, you configure OpenIG as a relying party, and use OpenAM as the OpenID Provider.

8.3. Preparing the Tutorial

In the chapter on *Getting Started*, you learned how to configure OpenIG to proxy traffic and capture request and response data. You also learned how to configure OpenIG to use a static request to log in with hard-coded credentials.

This tutorial shows you how OpenIG can act as an OpenID Connect 1.0 relying party.

This tutorial relies on OpenAM as an OpenID Provider. As a relying party, OpenIG takes the end user to OpenAM for authorization and an access token. It then uses the access token to get end user information from OpenAM.

Before you start this tutorial, prepare OpenIG and the minimal HTTP server as you did for the chapter on *Getting Started*.

OpenIG should be running in Jetty, configured to access the minimal HTTP server as described in that chapter.

8.4. Setting Up OpenAM as an OpenID Provider

Install and configure OpenAM on `http://openam.example.com:8088/openam` with the default configuration. If you use a different configuration, make sure you substitute in the tutorial accordingly. Although this tutorial does not use HTTPS, you must use HTTPS to protect access tokens and user information in production environments.

Login to the OpenAM console as administrator, and use the common task wizard, Configure OAuth2, to configure an OAuth 2.0 authorization server in / (Top Level Realm). This also configures OpenAM as an OpenID Provider.

Also create an OAuth 2.0 Client profile in / (Top Level Realm). This allows OpenIG to communicate with OpenAM as an OAuth 2.0 client. In OpenAM console, browse to Access Control > / (Top Level Realm) > Agents > OAuth 2.0 Client, and then click New in the Agent table.

Create an OAuth 2.0 client profile with name `OpenIG` and password `password`. The name is the "clientId" value, and the password is the "clientSecret" value that you use in the provider configuration in OpenIG.

Edit the `OpenIG` client profile to add the Redirection URI `http://www.example.com:8080/openid/callback`. Also add `openid` and `profile` scopes to the Scope(s) list, and then save your work.

In this tutorial, you overload the profile settings to pass credentials to OpenIG. This tutorial uses Full Name and Last Name for the sake of simplicity. Both of those attributes are part of a user's profile out of the box with the default OpenAM configuration. Neither of the attributes are needed for anything else in this tutorial. So this tutorial uses Last Name to hold the username, and Full Name to hold the password. In a real deployment, you would no doubt use other attributes, depending upon the user profiles and on your requirements.

To overload the profile, create a user whose additional credentials you set in the Full Name and Last Name fields, or edit the existing user `george` if you have already created the profile for another tutorial.

1. In OpenAM console, under Access Control > / (Top Level Realm) > Subjects > User, click New to create the user profile.

   If the profile already exists in the table, then click the link to open the profile for editing.

2. Set the ID to `george`, the password to `costanza`, the Last Name to `george`, and the Full Name to `costanza` before clicking OK (or Save).

3. When finished, log out of OpenAM console by clicking the log out button. It is not enough simply to close the browser tab, as the OpenAM session remains active until you log out or quit the browser.

8.5. Configuring OpenIG as a Relying Party

To configure OpenIG as an OpenID Connect 1.0 relying party, add a new route to the OpenIG configuration, by including the following route configuration file as `$HOME/.openig/config/routes/07-openid.json`.

```
{
  "handler": {
    "type": "Chain",
    "config": {
      "filters": [
        {
          "type": "OAuth2ClientFilter",
          "config": {
            "clientEndpoint": "/openid",
            "requireHttps": false,
            "requireLogin": true,
            "target": "${exchange.openid}",
            "scopes": [
              "openid",
              "profile"
            ],
            "failureHandler": {
              "type": "StaticResponseHandler",
              "config": {
                "comment": "Trivial failure handler for debugging only",
                "status": 500,
                "reason": "Error",
                "entity": "${exchange.openid}"
              }
            },
            "providerHandler": "ClientHandler",
            "providers": [
              {
                "name": "openam",
                "wellKnownConfiguration":
                    "http://openam.example.com:8088/openam/.well-known/openid-configuration",
                "clientId": "OpenIG",
                "clientSecret": "password"
              }
            ]
          }
        }
      ],
      "handler": {
        "type": "Chain",
        "config": {
          "filters": [
            {
              "type": "StaticRequestFilter",
              "config": {
                "method": "POST",
                "uri": "http://www.example.com:8081",
                "form": {
                  "username": [
                    "${exchange.openid.user_info.family_name}"
                  ],
                  "password": [
                    "${exchange.openid.user_info.name}"
                  ]
                }
              }
            }
          ],
          "handler": "ClientHandler"
        }
      }
    }
  },
  "condition": "${matches(exchange.request.uri.path, '^/openid')}",
  "baseURI": "http://www.example.com:8080"
}
```

On Windows, the file name should be `%appdata%\OpenIG\config\routes\07-openid.json`.

Notice the following features of the new route.

- At the global level the route changes the base URI for requests to ensure that the initial exchange happens between OpenIG and OpenAM, which is the OpenID Provider. This route sends only the final request to the protected application.

- The first filter in the outermost chain has the OAuth2ClientFilter in the *Reference* type. This is the filter that enables OpenIG to act as a relying party.

  The filter is configured to work only with a single provider, the OpenAM server you configured in Section 8.4, "Setting Up OpenAM as an OpenID Provider". If you had more than one provider configured, you would need a "loginHandler" as well to help end users pick a provider.

  The "OAuth2ClientFilter" has a base client endpoint of `/openid`. Incoming requests to `/openid/login` start the delegated authorization process. Incoming requests to `/openid/callback` are expected as redirects from the OP (as authorization server), so this is why you set the redirect URI in the client profile in OpenAM to `http://www.example.com:8080/openid/callback`.

  The "OAuth2ClientFilter" has `"requireHttps": false` as a convenience for testing. In production environments, require HTTPS.

  The filter has `"requireLogin": true` to ensure you see the delegated authorization process when you make your request.

  In the "OAuth2ClientFilter", the target for storing authorization state information is `${exchange.openid}`, so this is where subsequent filters and handlers can find access token and user information.

  The scopes are set to "openid" and "profile" as allowed for OpenID Connect 1.0.

Notice that on failure the filter dumps the current information in the exchange into a web page response to the end user. While this is helpful to you for debugging purposes, it is not helpful to an end user. In production environments, return a more user-friendly failure page.

Also in the "OAuth2ClientFilter", the typical "ClientHandler" configures the HTTP client that communicates with the OpenID Provider.

- After the filter injects the access token and user information into `exchange.openid`, OpenIG invokes a "Chain". The "Chain" uses the credentials to log the user in to the minimal HTTP server.

  With this configuration, all successful requests result in login attempts against the minimal HTTP server.

- The "StaticRequestFilter" retrieves the username and password from the exchange and replaces the original HTTP GET request with an HTTP POST login request that contains the credentials to authenticate.

- The route matches requests to `/openid`.

8.6. Trying It Out

To try your configuration, browse to OpenIG at http://www.example.com:8080/openid.

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

If successful, OpenIG logs you into the minimal HTTP server as George Costanza, and the minimal HTTP server returns George's page.

What is happening behind the scenes?

After OpenIG gets the browser request, the "OAuth2ClientFilter" redirects you to authenticate with OpenAM and consent to authorize access to user information. After you authorize access, OpenAM returns an access token to the filter.

The filter then uses that access token to get the user information. The filter injects the authorization state information into `exchange.openid`. The outermost chain then calls its handler, which as another "Chain".

This inner chain uses the credentials to log the user in to the minimal HTTP server, which responds with its User Information page.

Chapter 9 Configuring Routes

Other tutorials in this guide demonstrate how to use routes so that you can change the configuration without restarting OpenIG.

This tutorial takes a closer look at Router in the *Reference* and Route in the *Reference* configurations. This tutorial demonstrates the use of routes to handle multiple applications. It also shows how to lock down the configurations for deployment so that accidental changes to configuration files do not affect servers running in production.

9.1. Before You Start

Before you start this tutorial, prepare OpenIG and the minimal HTTP server as you did for the chapter on *Getting Started*.

OpenIG should be running in Jetty, configured to access the minimal HTTP server as described in that chapter.

You start therefore with a default route

9.2. Configuring the Router

When you set up the first tutorial, you configured a Router.

The Router is a handler that you can configure in the top-level `config.json` file for OpenIG, and in fact wherever you can configure a Handler. For the first tutorial, you added a Router as part of the base configuration, which is shown here again in the following listing.

```
{
    "handler": {
        "type": "Router",
        "audit": "global",
        "capture": "all"
    },
    "heap": [
        {
            "name": "LogSink",
            "type": "ConsoleLogSink",
            "config": {
                "level": "DEBUG"
            }
        },
        {
            "name": "JwtSession",
            "type": "JwtSession"
        },
        {
            "name": "ClientHandler",
            "type": "ClientHandler"
        },
        {
            "name": "capture",
            "type": "CaptureDecorator",
            "config": {
                "captureEntity": true,
                "_captureExchange": true
            }
        }
    ],
    "baseURI": "http://www.example.com:8081"
}
```

The Router's job is to pass the exchange to a route that matches a condition, and to periodically reload changed route configurations. As routes define the conditions on which they accept any given Exchange, 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 OpenIG is running, as you have done in other tutorials.

The configuration shown above passes all Exchanges to the Router using the default settings, meaning that the Router monitors `$HOME/.openig/config/routes` for Routes. When OpenIG receives a request, if more time has passed than the default scan interval of 10 seconds, then OpenIG rescans the Routes directory for changes and reloads any Routes changes it finds.

9.3. Configuring Additional Routes

Routes are configurations to handle an Exchange that meets a specified condition.

The condition is defined using a OpenIG expression in the *Reference*, so it can be based on almost any characteristic of the Exchange or even of the OpenIG runtime environment. Another way to think of the Route is like an independent DispatchHandler in the *Reference*.

Routes can also have their own names, used to order them lexicographically. If no name is specified, the Route file name is used. Route file names have the extension `.json`. In other words, a router only scans for files with the `.json` extension, and ignores files with other extensions.

Routes can have a base URI to change the scheme, host, and port of the request.

Routes wrap a heap of configuration objects, and hand off any Exchange they accept to a handler. In this way each Route is much like its own server-wide configuration file.

If no condition is specified for the Route, the Route accepts any Exchange. The following is a basic default route that accepts any Exchange and forwards it on without changes.

```
{
    "name": "default",
    "handler": {
        "type": "ClientHandler"
    }
}
```

The rest of this section indicates how to set up Route configurations to direct requests to ForgeRock.com and ForgeRock.org based on a query string parameter.

1. Add a ForgeRock.com Route file in the routes directory, `08-com.json`, that holds the following content.

   ```
   {
       "handler": "ClientHandler",
       "condition": "${matches(exchange.request.uri.query, 'site=com')}",
       "baseURI": "http://forgerock.com:80/",
       "audit": "ForgeRock.com route"
   }
   ```

   This Route accepts the Exchange when the query string parameter, `site` matches `com`. When this Route picks up an Exchange, it changes the request scheme, host, and port, and sends it to ForgeRock.com.

2. Add a ForgeRock.org community Route file in the routes directory, `08-org.json`, that holds the following content.

   ```
   {
       "handler": "ClientHandler",
       "condition": "${matches(exchange.request.uri.query, 'site=org')}",
       "baseURI": "https://forgerock.org:443/",
       "audit": "ForgeRock.org route"
   }
   ```

   This Route accepts the Exchange when the query string parameter, `site` matches `org`. When this Route picks up an Exchange, it changes the request scheme, host, and port, and sends it to ForgeRock.org.

9.4. Trying it Out

At this point you can try your new route configurations.

Browse to the .com URL: http://www.example.com:8080/?site=com.

You should see the ForgeRock.com page.

Browse to the .org URL: http://www.example.com:8080/?site=org.

You should see the ForgeRock.org Community page.

Now browse to the base URL to see that the default route still works: http://www.example.com:8080/.

What happened behind the scenes?

When you issued your first request with "?site=com", the request matched the condition defined in the ForgeRock.com route. OpenIG rebased the request and sent it along to `http://forgerock.com:80/`.

When you issued your second request with "?site=org", the request matched the condition defined in the ForgeRock.org Community route. OpenIG rebased the request and sent it along to `http://forgerock.org:80/`.

When the third request did not match any of the conditions defined, the Exchange was routed to the default Route (that accepts any Exchange). The static request filter returned the default page.

At this point, tinker with your route configurations without stopping OpenIG, and notice that changes are picked up every 10 seconds.

9.5. Locking Down Route Configurations

Having the Route configurations automatically reloaded is great in the lab, but is perhaps not what you want in production.

In that case, stop the server, edit the Router "scanInterval", and restart. When "scanInterval" is set to -1, the Router only loads routes at startup.

```
{
    "name": "Router",
    "type": "Router",
    "config": {
        "scanInterval": -1
    }
}
```

You can also change the file system location to look for routes.

```
{
    "name": "Router",
    "type": "Router",
    "config": {
        "directory": "/path/to/safe/routes",
        "scanInterval": -1
    }
}
```

Chapter 10
Configuration Templates

This chapter contains template routes for common configurations.

Before you use one of the templates here, install and configure OpenIG with a Router and default route as described in the chapter on *Getting Started*.

Next, take one of the templates and then modify it to suit your deployment. Read the summary of each template to find the right match for your application.

When you move to use OpenIG in production, be sure to turn off DEBUG level logging, and to deactivate "CaptureDecorator" use to avoid filling up disk space. Also consider locking down the Router configuration.

10.1. Proxy & Capture

If you installed and configured OpenIG with a Router and default route as described in the chapter on *Getting Started*, then you already proxy & capture both the application requests coming in and the server responses going out.

This template is a replacement for the default route, `$HOME/.openig/config/routes/99-default.json`, with a "DispatchHandler" to change the scheme to HTTPS on login. Simply change the "baseURI" to that of the target application, and login to the application. This template references a "ClientHandler" defined in `config.json`.

```
{
    "handler": {
        "type": "DispatchHandler",
        "config": {
            "bindings": [
                {
                    "condition": "${exchange.request.uri.scheme == 'http'}",
                    "handler": "ClientHandler",
                    "baseURI": "http://TARGETIP"
                },
                {
                    "condition": "${exchange.request.uri.path == '/login'}",
                    "handler": "ClientHandler",
                    "baseURI": "https://TARGETIP"
                },
                {
                    "handler": "ClientHandler",
                    "baseURI": "https://TARGETIP"
                }
            ]
        }
    },
    "capture": "all"
}
```

10.2. Simple Login Form

Logs the user into the target application with hard-coded user name and password. This template intercepts the login page request and replaces it with the login form.

```
{
    "handler": {
        "type": "DispatchHandler",
        "config": {
            "bindings": [
                {
                    "condition": "${exchange.request.uri.path == '/login'}",
                    "handler": {
                        "type": "Chain",
                        "config": {
                            "filters": [
                                {
                                    "type": "StaticRequestFilter",
                                    "config": {
                                        "method": "POST",
                                        "uri": "https://TARGETIP/login",
                                        "form": {
                                            "USER": [
                                                "MY_USERNAME"
                                            ],
                                            "PASSWORD": [
                                                "MY_PASSWORD"
                                            ]
                                        }
                                    }
                                }
                            ],
                            "handler": "ClientHandler"
                        }
                    },
                    "baseURI": "http://TARGETIP"
                },
                {
                    "handler": "ClientHandler",
                    "baseURI": "http://TARGETIP"
                }
            ]
        }
    }
}
```

This template is a replacement for the default route, `$HOME/.openig/config/routes/99-default.json`, Substitute `TARGETIP` with the IP address of your application. Also change `MY_USERNAME` and `MY_PASSWORD`, and adapt the "StaticRequestFilter" for your application. This template references a "ClientHandler" defined in `config.json`.

10.3. Login Form With Cookie From Login Page

For applications that expect a cookie from the login page to be sent in the login request form. This templates allows the login page request to go through to the target, intercepts the response, then creates the login form and adds the intercepted cookie to the POST.

```
{
    "heap": [
        {
            "name": "DispatchHandler",
            "type": "DispatchHandler",
            "config": {
                "bindings": [
                    {
                        "condition": "${exchange.request.uri.path == '/eum/login'}",
                        "handler": {
                            "type": "Chain",
                            "config": {
                                "filters": [
                                    {
                                        "type": "SwitchFilter",
                                        "config": {
                                            "onResponse": [
                                                {
                                                    "handler": "LoginRequestHandler"
                                                }
                                            ]
                                        }
                                    }
                                ],
                                "handler": "ClientHandler"
                            }
                        },
                        "baseURI": "http://TARGETIP"
                    },
                    {
                        "handler": "ClientHandler",
                        "baseURI": "http://TARGETIP"
                    }
                ]
            }
        },
        {
            "name": "LoginRequestHandler",
            "type": "Chain",
            "config": {
                "filters": [
                    {
                        "type": "StaticRequestFilter",
                        "config": {
                            "method": "POST",
                            "uri": "https://TARGETIP/login",
                            "form": {
                                "USER": [
                                    "MY_USERNAME"
                                ],
                                "PASSWORD": [
                                    "MY_PASSWORD"
                                ]
                            },
                            "headers": {
                                "cookie": [
                                    "${exchange.response.headers['Set-Cookie'][0]}"
                                ]
                            }
                        }
                    }
                ],
                "handler": "ClientHandler"
            }
        }
    ],
    "handler": "DispatchHandler"
}
```

This template is a replacement for the default route, `$HOME/.openig/config/routes/99-default.json`, Substitute `TARGETIP` with the IP address of your application. Also change `MY_USERNAME` and `MY_PASSWORD`, and adapt the "StaticRequestFilter" for your application. This template references a "ClientHandler" defined in `config.json`.

10.4. Login Form With Extract Filter & Cookie Filter

For applications that return the login page when the user tries to access a page without a valid session. This template shows how to use the "ExtractFilter" to find the login page on the response and use the "CookieFilter" to ensure the cookies from the application are replayed on each request. The sample application in this template is OpenAM.

Note

Without the "CookieFilter" in the "OutgoingChain" the cookie set in the login page response would not get set in the browser since that request is intercepted before it gets to the browser. The simplest way to deal with this situation is to let OpenIG manage all the cookies by enabling the "CookieFilter". The side effect of OpenIG managing cookies is none of the cookies are sent to the browser, but are managed locally by OpenIG.

```
{
    "heap": [
        {
            "name": "DispatchHandler",
            "type": "DispatchHandler",
            "config": {
                "bindings": [
                    {
                        "handler": {
                            "type": "Chain",
                            "config": {
                                "filters": [
                                    "IsLoginPage",
                                    {
                                        "type": "EntityExtractFilter",
                                        "config": {
                                            "messageType": "response",
                                            "target": "${exchange.isLoginPage}",
                                            "bindings": [
                                                {
                                                    "key": "found",
                                                    "pattern": "OpenAM\s\(Login\)",
                                                    "template": "true"
                                                }
                                            ]
                                        }
                                    }
                                ],
                                "handler": "OutgoingChain"
                            }
                        },
                        "baseURI": "http://TARGETIP:PORT"
                    }
                ]
            }
        },
        {
            "name": "IsLoginPage",
            "type": "SwitchFilter",
            "config": {
                "onResponse": [
                    {
                        "condition": "${exchange.isLoginPage.found == 'true'}",
                        "handler": {
                            "type": "Chain",
                            "config": {
                                "filters": [
                                    {
                                        "type": "StaticRequestFilter",
                                        "config": {
                                            "method": "POST",
                                            "uri":
                                                "http://TARGETIP:PORT/openam/UI/Login",
                                            "form": {
                                                "IDToken0": [
                                                    ""
                                                ],
                                                "IDToken1": [
                                                    "MY_USERNAME"
                                                ],
                                                "IDToken2": [
                                                    "MY_PASSWORD"
                                                ],
                                                "IDButton": [
                                                    "Log+In"
                                                ],
                                                "encoded": [
                                                    "false"
                                                ]
                                            },
                                            "headers": {
                                                "host": [
                                                    "TARGETFQDN:PORT"
                                                ]
                                            }
                                        }
                                    }
                                ],
                                "handler": "OutgoingChain"
                            }
                        }
                    }
                ]
            }
        },
        {
            "name": "OutgoingChain",
            "type": "Chain",
            "config": {
                "filters": [
                    {
                        "type": "CookieFilter"
                    }
                ],
                "handler": {
                    "type": "ClientHandler"
                }
            }
        }
    ],
    "handler": "DispatchHandler"
}
```

This template is a replacement for the default route, `$HOME/.openig/config/routes/99-default.json`, Substitute `TARGETIP` with the IP address of OpenAM, `TARGETFQDN` with the fully qualified domain name of OpenAM, `PORT` with the port on which OpenAM listens. Also change `MY_USERNAME` and `MY_PASSWORD` to match those of your OpenAM user. This template references a "ClientHandler" defined in `config.json`.

10.5. Login Which Requires a Hidden Value From the Login Page

Extracts a hidden value from the login page and includes it in the login form POSTed to the target application.

```
{
    "handler": {
        "type": "DispatchHandler",
        "config": {
            "bindings": [
                {
                    "condition": "${exchange.request.uri.path == '/login'}",
                    "handler": {
                        "name": "LoginChain",
                        "type": "Chain",
                        "config": {
                            "filters": [
                                {
                                    "type": "EntityExtractFilter",
                                    "config": {
                                        "messageType": "response",
                                        "target": "${exchange.hiddenValue}",
                                        "bindings": [
                                            {
                                                "key": "value",
                                                "pattern":
                                                    "wpLoginToken\"\\s.*value=\"(.*)\"",
                                                "template": "$1"
                                            }
                                        ]
                                    }
                                },
                                {
                                    "type": "StaticRequestFilter",
                                    "config": {
                                        "method": "POST",
                                        "uri": "https://TARGETIP/login",
                                        "form": {
                                            "USER": [
                                                "MY_USERNAME"
                                            ],
                                            "PASSWORD": [
                                                "MY_PASSWORD"
                                            ],
                                            "hiddenValue": [
                                                "${exchange.hiddenValue.value}"
                                            ]
                                        }
                                    }
                                }
                            ],
                            "handler": "ClientHandler"
                        }
                    },
                    "baseURI": "http://TARGETIP"
                },
                {
                    "handler": "ClientHandler",
                    "baseURI": "http://TARGETIP"
                }
            ]
        }
    }
}
```

This template is a replacement for the default route, `$HOME/.openig/config/routes/99-default.json`, Substitute `TARGETIP` with the IP address of your application. Also change `MY_USERNAME` and `MY_PASSWORD`, and adapt the "StaticRequestFilter" for your application. This template references a "ClientHandler" defined in `config.json`.

10.6. HTTP & HTTPS Application

Proxies traffic to an application listening on ports 80 and 443. The assumption is the application uses HTTPS for authentication and HTTP for the general application features. Assuming all login requests go to port 443, you must add the login filters and handlers to the "Chain".

```
{
    "handler": {
        "type": "DispatchHandler",
        "config": {
            "bindings": [
                {
                    "condition": "${exchange.request.uri.scheme == 'http'}",
                    "handler": "ClientHandler",
                    "baseURI": "http://TARGETIP"
                },
                {
                    "condition": "${exchange.request.uri.path == '/login'}",
                    "handler": {
                        "type": "Chain",
                        "config": {
                            "filters": [
                                "MY_FILTERS"
                            ],
                            "handler": "ClientHandler"
                        }
                    },
                    "baseURI": "https://TARGETIP"
                },
                {
                    "handler": "ClientHandler",
                    "baseURI": "https://TARGETIP"
                }
            ]
        }
    }
}
```

This template is a replacement for the default route, `$HOME/.openig/config/routes/99-default.json`, Substitute `TARGETIP` with the IP address of your application. Also add the necessary filter configurations that are required for login with your application, and then change `MY_FILTERS` to identify the added filters. This template references a "ClientHandler" defined in `config.json`.

10.7. OpenAM Integration With Headers

Logs the user into the target application using the headers passed down from an OpenAM policy agent. This template assumes the user name and password are passed down by the OpenAM policy agent as headers. If the header passed in 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.

```
{
    "handler": {
        "type": "DispatchHandler",
        "config": {
            "bindings": [
                {
                    "condition": "${exchange.request.uri.path == '/login'}",
                    "handler": {
                        "type": "Chain",
                        "config": {
                            "filters": [
                                {
                                    "type": "StaticRequestFilter",
                                    "config": {
                                        "method": "POST",
                                        "uri": "https://TARGETIP/login",
                                        "form": {
                                            "USER": [
                                                "${exchange.request.headers['username'][0]}"
                                            ],
                                            "PASSWORD": [
                                                "${exchange.request.headers['password'][0]}"
                                            ]
                                        }
                                    }
                                }
                            ],
                            "handler": "ClientHandler"
                        }
                    },
                    "baseURI": "http://TARGETIP"
                },
                {
                    "handler": "ClientHandler",
                    "baseURI": "http://TARGETIP"
                }
            ]
        }
    }
}
```

This template is a replacement for the default route, `$HOME/.openig/config/routes/99-default.json`, Substitute `TARGETIP` with the IP address of your application. Also adapt the "StaticRequestFilter" for your application. This template references a "ClientHandler" defined in `config.json`.

10.8. Microsoft Online Outlook Web Access

Logs the user into Microsoft Online Outlook Web Access (OWA). This template shows how you would use OpenIG and the OpenAM password capture feature to integrate with OWA. You can follow the chapter on *Getting Login Credentials From OpenAM*, and substitute this template as a replacement for the default route.

```
{
  "heap": [
    {
      "name": "DispatchHandler",
      "type": "DispatchHandler",
      "config": {
        "bindings": [
          {
            "condition": "${exchange.request.uri.path == '/owa/auth/logon.aspx'}",
            "handler": {
              "type": "Chain",
              "config": {
                "filters": [
                  {
                    "type": "CryptoHeaderFilter",
                    "config": {
                      "messageType": "REQUEST",
                      "operation": "DECRYPT",
                      "algorithm": "DES/ECB/NoPadding",
                      "key": "DESKEY",
                      "keyType": "DES",
                      "charSet": "utf-8",
                      "headers": [
                        "password"
                      ]
                    }
                  },
                  {
                    "type": "StaticRequestFilter",
                    "config": {
                      "method": "POST",
                      "uri": "https://65.55.171.158/owa/auth/owaauth.dll",
                      "headers": {
                        "Host": [
                          "red001.mail.microsoftonline.com"
                        ],
                        "Content-Type": [
                          "Content-Type:application/x-www-form-urlencoded"
                        ]
                      },
                      "form": {
                        "destination": [
                          "https://red001.mail.microsoftonline.com/owa/"
                        ],
                        "forcedownlevel": [
                          "0"
                        ],
                        "trusted": [
                          "0"
                        ],
                        "username": [
                          "${exchange.request.headers['username'][0]}"
                        ],
                        "password": [
                          "${exchange.request.headers['password'][0]}"
                        ],
                        "isUtf8": [
                          "1"
                        ]
                      }
                    }
                  }
                ],
                "handler": "OutgoingChain"
              }
            },
            "baseURI": "https://65.55.171.158"
          },
          {
            "handler": "OutgoingChain",
            "baseURI": "https://65.55.171.158"
          }
        ]
      }
    },
    {
      "name": "OutgoingChain",
      "type": "Chain",
      "config": {
        "filters": [
          {
            "type": "HeaderFilter",
            "config": {
              "messageType": "REQUEST",
              "remove": [
                "password",
                "username"
              ]
            }
          }
        ],
        "handler": {
          "type": "ClientHandler"
        }
      }
    }
  ],
  "handler": "DispatchHandler"
}
```

Change `DESKEY` to the actual key value that you generated when following the instructions in Configuring Password Capture. This template references a "ClientHandler" defined in `config.json`.

Chapter 11 Extending OpenIG

This chapter looks at extending what OpenIG can do beyond what you get out of the box.

To extend what you can do with Filters and Handlers, OpenIG supports dynamic scripting languages like Groovy through the use of `ScriptableFilter` and `ScriptableHandler` objects.

If scripting is not enough, be aware that OpenIG includes a complete application programming interface, designed to allow you to customize OpenIG as required. Customizing OpenIG can be used to perform complex server interactions or intensive data transformations that you cannot achieve with scripts or existing handlers, filters and expressions in the *Reference*.

Interface Stability: Evolving in the *Reference*

11.1. About Scripting

You add these Filters and Handlers to your configuration in the same way as for other Filters and Handlers. Each takes as its configuration the script's Internet media "type" and either a "source" script included in the JSON configuration, or a "file" script that OpenIG reads from a file. The configuration can also optionally supply "args" in order to pass parameters to the script.

The following example defines a `ScriptableFilter`, written in the Groovy language, and stored in a file named `$HOME/.openig/scripts/groovy/SimpleFormLogin.groovy` (`%appdata%\OpenIG\scripts\groovy\SimpleFormLogin.groovy` on Windows).

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

Relative paths in the "file" field depend on how OpenIG is installed. If OpenIG is installed in an application server, then paths for Groovy scripts are relative to `$HOME/.openig/scripts/groovy`.

This base location `$HOME/.openig/scripts/groovy` is on the classpath when the scripts are executed. If therefore some Groovy scripts are not in the default package, but instead have their own package names, they belong in the directory corresponding to their package name. For example, a script in package `com.example.groovy` belongs under `$HOME/.openig/scripts/groovy/com/example/groovy/`.

OpenIG provides scripts with several global variables at run time, enabling them to access the Exchange, to store variables across executions, to write messages to the logs, and to make requests to a web service or to an LDAP directory service, in addition to Groovy's built-in functionality. For details, see the reference documentation for ScriptableFilter in the *Reference* and ScriptableHandler in the *Reference*.

This chapter demonstrates some of what you might do using scripts.

If you want to try these scripts, first install and configure OpenIG as described in the chapter on *Getting Started*.

When developing and debugging your scripts, consider configuring a *CaptureDecorator* in the *Reference* to log requests, responses, and the exchange data in JSON form. You can then turn off capturing when you move to production.

11.2. Scripting Dispatch

In order to route requests, especially when the conditions are complicated, you can use a `ScriptableHandler` instead of a DispatchHandler in the *Reference*.

The following script demonstrates a simple dispatch handler.

```
import org.forgerock.openig.http.Response

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

// Rather than get the response from an external source,
// this handler produces the response itself.
exchange.response = new Response();

switch (exchange.request.uri.path) {

    case "/login":

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

            exchange.response.status = 200
            exchange.response.entity = "<html><p>Welcome back, Babs!</p></html>"

        } else {

            exchange.response.status = 403
            exchange.response.entity = "<html><p>Authorization required</p></html>"

        }

        break
    default:

        exchange.response.status = 401
        exchange.response.entity = "<html><p>Please <a href='./login'>log in</a>.</p></html>"

        break

}
```

To try this handler, save the script as `$HOME/.openig/scripts/groovy/DispatchHandler.groovy` (`%appdata%\OpenIG\scripts\groovy\DispatchHandler.groovy` on Windows).

Next, add the following route to your configuration as `$HOME/.openig/config/routes/98-dispatch.json` (`%appdata%\OpenIG\config\routes\98-dispatch.json` on Windows).

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

The route sets up the headers required by the script when the user logs in.

To try it out, browse to http://www.example.com:8080.

The response from the script says `Please log in.` When you click the "log in" link, the "HeaderFilter" sets Username and Password headers in the request, and passes the request to the script.

The script then responds, `Welcome back, Babs!`

11.3. Scripting HTTP Basic Authentication

HTTP Basic authentication calls for the user agent such as a browser to send a user name and password to the server in an Authorization header. HTTP Basic authentication relies on an encrypted connection to protect the user name and password credentials, which are base64-encoded in the Authorization header, not encrypted.

The following script, for use in a `ScriptableFilter`, adds an Authorization header based on a username and password combination.

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

def userPass = username + ":" + password
def base64UserPass = userPass.getBytes().encodeBase64()
```

```
exchange.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.
 */
exchange.request.uri.scheme = "https"

// Call the next handler. This returns when the request has been handled.
next.handle(exchange)
```

To try this filter, save the script as `$HOME/.openig/scripts/groovy/BasicAuthFilter.groovy` (`%appdata%\OpenIG\scripts\groovy\BasicAuthFilter.groovy` on Windows).

Next, add the following route to your configuration as `$HOME/.openig/config/routes/09-basic.json` (`%appdata%\OpenIG\config\routes\09-basic.json` on Windows).

```
{
    "handler": {
        "type": "Chain",
        "config": {
            "filters": [
                {
                    "type": "ScriptableFilter",
                    "config": {
                        "type": "application/x-groovy",
                        "file": "BasicAuthFilter.groovy",
                        "args": {
                            "username": "bjensen",
                            "password": "hifalutin"
                        }
                    },
                    "capture": "filtered_request"
                }
            ],
            "handler": {
                "type": "StaticResponseHandler",
                "config": {
                    "status": 200,
                    "reason": "OK",
                    "entity": "Hello, Babs!"
                }
            }
        }
    },
    "condition": "${matches(exchange.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, OpenIG returns a static page.

To try it out, browse to http://www.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://www.example.com:8080/basic HTTP/1.1
Authorization: Basic YmplbnNlbjpoaWZhbHV0aW4=
```

11.4. Scripting LDAP Authentication

Many organizations use an LDAP directory service to store user profiles including authentication credentials. The LDAP directory service securely stores user passwords in a highly-available, central service capable of handling thousands of authentications per second.

The following script, for use in a `ScriptableFilter`, performs simple authentication against an LDAP server based on request form fields `username` and `password`.

```
import org.forgerock.opendj.ldap.*
import org.forgerock.openig.http.Response

/*
 * Perform LDAP authentication based on user credentials from a form.
 *
 * If LDAP authentication succeeds, then call the next handler.
 * If there is a failure, send a response back to the user.
 */

username = exchange.request.form?.username[0]
password = exchange.request.form?.password[0]

// For testing purposes, the LDAP host and port are provided in the exchange.
// Edit as needed to match your directory service.
host = exchange.ldapHost ?: "localhost"
port = exchange.ldapPort ?: 1389

client = ldap.connect(host, port as Integer)
try {

    // Assume the username is an exact match of either
    // the user ID, the email address, or the user's full name.
    filter = "(|(uid=%s)(mail=%s)(cn=%s))"

    user = client.searchSingleEntry(
            "ou=people,dc=example,dc=com",
            ldap.scope.sub,
            ldap.filter(filter, username, username, username))

    client.bind(user.name as String, password?.toCharArray())

    // Authentication succeeded.

    // Set a header (or whatever else you want to do here).
    exchange.request.headers.add("Ldap-User-Dn", user.name)

    // 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.
    exchange.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:
    exchange.description = user.description?.parse().asString()

    // Call the next handler. This returns when the request has been handled.
    next.handle(exchange)

} catch (AuthenticationException e) {

    // LDAP authentication failed, so fail the exchange with
    // HTTP status code 403 Forbidden.

    exchange.response = new Response()
    exchange.response.status = 403
    exchange.response.reason = e.message
    exchange.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 exchange with HTTP 500 Internal Server Error.

    exchange.response = new Response()
    exchange.response.status = 500
    exchange.response.reason = e.message
    exchange.response.entity = "<html><p>Server error: " + e.message + "</p></html>"

} finally {
    client.close()
}
```

For the list of methods to specify which type of objects to return, see the OpenDJ LDAP SDK Javadoc for `AttributeParser`.

To try this out, first install an LDAP directory server such as ForgeRock Directory Services or OpenDJ directory server. Also import some sample users who can authenticate over LDAP. With OpenDJ, you can generate sample users at installation time.

Next, save the script as `$HOME/.openig/scripts/groovy/LdapAuthFilter.groovy` (`%appdata%\OpenIG\scripts\groovy\LdapAuthFilter.groovy` on Windows). If the directory server installation does not match the assumptions made in the script, adjust the script to use the correct settings for your installation.

Finally, add the following route to your configuration as `$HOME/.openig/config/routes/10-ldap.json` (`%appdata%\OpenIG\config\routes\10-ldap.json` on Windows).

```
{
    "handler": {
        "type": "Chain",
        "config": {
            "filters": [
                {
                    "type": "ScriptableFilter",
                    "config": {
                        "type": "application/x-groovy",
                        "file": "LdapAuthFilter.groovy"
                    }
                }
            ],
            "handler": {
                "type": "ScriptableHandler",
                "config": {
                    "type": "application/x-groovy",
                    "source":
                        "import org.forgerock.openig.http.Response;
                         dn = exchange.request.headers['Ldap-User-Dn'][0];
                         entity = '<html><p>Ldap-User-Dn: ' + dn + '</p></html>';

                         exchange.response = new Response();
                         exchange.response.status = 200;
                         exchange.response.entity = entity;"
                }
            }
        }
    },
    "condition": "${matches(exchange.request.uri.path, '^/ldap')}"
}
```

The route calls the "LdapAuthFilter.groovy" script to authenticate the user over LDAP. On successful authentication, it responds with the the bind DN.

To try it out, browse to a URL where query string parameters specify a valid username and password, such as http://www.example.com:8080/ldap?username=user.0&password=password.

The response from the script shows the DN: `Ldap-User-Dn: uid=user.0,ou=People,dc=example,dc=com`.

11.5. Scripting SQL Queries

You can use a `ScriptableFilter` to look up information in a relational database and include the results in the Exchange.

The following filter looks up user credentials in a database given the user's email address, which is found in the form data of the request. The script then sets the credentials in headers, making sure the scheme is HTTPS to protect the request when it leaves OpenIG.

```
/*
 * 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 exchange headers for the next handler.
 */

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

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

// Call the next handler. This returns when the request has been handled.
next.handle(exchange)
```

The previous script demonstrates a `ScriptableFilter` that uses a `SqlClient` class defined in another script. The following code listing shows the `SqlClient` class.

```
import groovy.sql.Sql

import javax.naming.InitialContext
import javax.sql.DataSource

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

    // Get a DataSource from the container.
    InitialContext context = new InitialContext()
    DataSource dataSource = context.lookup("jdbc/forgerock") as DataSource
    def sql = new Sql(dataSource)

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

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

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

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

To try this out, first follow the tutorial on *Login With Credentials From a Database*. When everything in that tutorial works, you know that OpenIG can connect to the database, lookup users by email address, and successfully authenticate to the sample application.

Next, save the scripts as `$HOME/.openig/scripts/groovy/SqlAccessFilter.groovy` (`%appdata%\OpenIG\scripts\groovy\SqlAccessFilter.groovy` on Windows), and as `$HOME/.openig/scripts/groovy/SqlClient.groovy` (`%appdata%\OpenIG\scripts\groovy\SqlClient.groovy` on Windows).

Finally, add the following route to your configuration as `$HOME/.openig/config/routes/11-db.json` (`%appdata%\OpenIG\config\routes\11-db.json` on Windows).

```
{
    "handler": {
        "type": "Chain",
        "config": {
            "filters": [
                {
                    "type": "ScriptableFilter",
                    "config": {
                        "type": "application/x-groovy",
                        "file": "SqlAccessFilter.groovy"
                    }
                },
                {
                    "type": "StaticRequestFilter",
                    "config": {
                        "method": "POST",
                        "uri": "http://www.example.com:8081",
                        "form": {
                            "username": [
                                "${exchange.request.headers['Username'][0]}"
                            ],
                            "password": [
                                "${exchange.request.headers['Password'][0]}"
                            ]
                        }
                    }
                }
            ],
            "handler": "ClientHandler"
        }
    },
    "condition": "${matches(exchange.request.uri.path, '^/db')}"
}
```

The route calls the "ScriptableFilter" to look up credentials over SQL. It then uses calls a "StaticRequestFilter" to build a login request. Although the script sets the scheme to HTTPS, the "StaticRequestFilter" ignores that and resets the URI. This is to make it easier for you to try this out, without having to worry about setting up HTTPS.

To try it out, browse to a URL where a query string parameter specifies a valid email address, such as http://www.example.com:8080/db?mail=george@example.com.

If the lookup and authentication are successful, you see the profile page of the sample application.

11.6. About Developing Custom Extensions

If scripting is not enough, be aware that OpenIG includes a complete Java application programming interface, designed to allow you to customize OpenIG as required. Customizing OpenIG can be used to perform complex server interactions or intensive data transformations that you cannot achieve with scripts or existing handlers, filters and expressions in the *Reference*.

11.7. Key Extension Points

Primary extension points include these interfaces.

AuditEventListener

An AuditEventListener observes audit events. For an example, see MonitorEndpointHandler in the *Reference*.

You must implement the interface for your audit agent and its heaplet must extend ConditionalAuditEventListener.ConditionalListenerHeaplet.

Decorator

A Decorator adds new behavior to another object, without changing the base type of the object.

When suggesting custom Decorator names, know that OpenIG reserves all field names that use only alphanumeric characters. To avoid clashes, use dots or dashes in your field names, such as "my-decorator".

Filter

A Filter serves to process the request and/or the response.

Handler

A Handler generates a response given a request.

These Filter and Handler interfaces are similar to Java Enterprise Edition `Filter` and `Servlet` interfaces, with some differences in the semantics of messages. While you can simply implement these interfaces, OpenIG also provides convenience classes: GenericFilter and GenericHandler.

11.8. Implementing a Filter

The `Filter` interface exposes a filter() method, which takes an Exchange object and the Chain of remaining filters and handler to dispatch to. Initially, exchange.request contains the request to be filtered. To pass the request to the next filter or handler in the chain, the filter calls next.handle(exchange). After this call, exchange.response contains the response that can be filtered.

A filter might 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(exchange)` and creating its own response object in the exchange. 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 exchange to the rest of the chain.

Note

If an existing response exists in the exchange object and the filter intends to replace it with its own, it must call the response.close() method in order to signal that the processing of the response from a remote server is complete.

11.9. Implementing a Handler

The `Handler` interface exposes a handle() method, which takes an Exchange object. It processes the request in exchange.request and produces a response in exchange.response. A handler can elect to dispatch the exchange to another handler or chain.

Note

If an existing response exists in the exchange object and the filter intends to replace it with its own, it must first check to see if the it must call the response.close() method in order to signal that the processing of the response from a remote server is complete.

11.10. Heap Object Configuration

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.

11.11. Sample Filter

The following sample filter sets an arbitrary header in the incoming request and outgoing response.

```
package org.forgerock.openig.doc;

import org.forgerock.openig.filter.GenericFilter;
import org.forgerock.openig.handler.Handler;
import org.forgerock.openig.handler.HandlerException;
import org.forgerock.openig.heap.GenericHeaplet;
import org.forgerock.openig.heap.HeapException;
import org.forgerock.openig.http.Exchange;

import java.io.IOException;

/**
 * Filter to set a header in the incoming request and in the outgoing response.
 */
public class SampleFilter extends GenericFilter {

    /** 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 exchange          Wraps request and response.
     * @param next              Next filter or handler in the chain.
     * @throws HandlerException Failure when handling the exchange.
     * @throws IOException      I/O exception when handling the exchange.
     */
    @Override
    public void filter(Exchange exchange, Handler next)
            throws HandlerException, IOException {

        // Set header in the request.
        exchange.request.getHeaders().putSingle(name, value);

        // Pass to the next filter or handler in the chain.
        next.handle(exchange);

        // Set header in the response.
        exchange.response.getHeaders().putSingle(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").required().asString();
            filter.value = config.get("value").required().asString();

            return filter;
        }
    }
}
```

When you set the sample filter "type" in the configuration, you need to provide the fully qualified class name, as in `"type": "org.forgerock.openig.doc.SampleFilter"`. You can however implement a class alias resolver to make it possible to use a short name instead, as in `"type": "SampleFilter"`.

```
package org.forgerock.openig.doc;

import org.forgerock.openig.alias.ClassAliasResolver;

import java.util.HashMap;
import java.util.Map;

/**
 * Allow use of short name aliases in configuration object types.
 *
 * This allows a configuration with {@code "type": "SampleFilter"}
 * instead of {@code "type": "org.forgerock.openig.doc.SampleFilter"}.
 */
public class SampleClassAliasResolver implements ClassAliasResolver {

    private static final Map<String, Class<?>> ALIASES =
            new HashMap<String, Class<?>>();

    static {
        ALIASES.put("SampleFilter", SampleFilter.class);
    }

    /**
     * Get the class for a short name alias.
     *
     * @param alias Short name alias.
     * @return      The class, or null if the alias is not defined.
     */
    @Override
    public Class<?> resolve(String alias) {
        return ALIASES.get(alias);
    }
}
```

When you add your own resolver, you must make it discoverable within your custom library. You do this by adding a services file named after the class resolver interface, where the file contains the fully qualified class name of your resolver, under `META-INF/services/org.forgerock.openig.alias.ClassAliasResolver` in the jar file for your customizations. When you have more than one resolver, add one fully qualified class name per line. If you build your project using Maven, then you can add this under the `src/main/resources` directory. The content of the file in this example is one line:

```
org.forgerock.openig.doc.SampleClassAliasResolver
```

The corresponding heap object configuration then looks as follows.

```
{
    "name": "SampleFilter",
    "type": "SampleFilter",
    "config": {
        "name": "X-Greeting",
        "value": "Hello world"
    }
}
```

11.12. Building Customizations

You can use Apache Maven to manage dependencies on OpenIG. The dependencies are found in the ForgeRock Maven repository.

The following listing shows the Maven POM configuration for the ForgeRock Maven repository and the dependency to build the sample Filter.

```
<repositories>
  <repository>
    <id>forgerock-staging-repository</id>
    <name>ForgeRock Release Repository</name>
    <url>http://maven.forgerock.org/repo/releases</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
  <repository>
    <id>forgerock-snapshots-repository</id>
    <name>ForgeRock Snapshot Repository</name>
    <url>http://maven.forgerock.org/repo/snapshots</url>
    <releases>
      <enabled>false</enabled>
    </releases>
  </repository>
</repositories>

<dependencies>
  <dependency>
    <groupId>org.forgerock.openig</groupId>
    <artifactId>openig-core</artifactId>
    <version>3.1.0</version>
  </dependency>
</dependencies>
```

You can then build your customizations into a jar file and install them in your local Maven repository by using the **mvn install** command.

```
$ mvn install
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ sample-filter ---
[INFO] Building jar: .../sample-filter/target/sample-filter-1.0.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.478s
[INFO] Finished at: Fri Nov 07 16:57:18 CET 2014
[INFO] Final Memory: 18M/309M
[INFO] ------------------------------------------------------------------------
```

11.13. Embedding Customizations in OpenIG

After building your customizations into a jar file, you can include them in the OpenIG war file for deployment. You do this by unpacking `OpenIG-3.1.0.war`, including your jar library in `WEB-INF/lib`, and then creating a new war file.

For example, if your jar file is in a project named `sample-filter`, and the development version is `1.0.0-SNAPSHOT`, you might include the file as in the following example.

```
$ mkdir root && cd root
$ jar -xf ~/Downloads/OpenIG-3.1.0.war
$ cp ~/Documents/sample-filter/target/sample-filter-1.0.0-SNAPSHOT.jar WEB-INF/lib
$ jar -cf ../custom.war *
```

In this example, the resulting `custom.war` contains the custom sample filter. You can deploy the custom war file as you would deploy `OpenIG-3.1.0.war`.

Chapter 12
OpenIG Audit Framework

OpenIG provides a publish-and-subscribe audit framework. Filters and Handlers publish audit events. Agents in OpenIG that are registered with the audit system can subscribe to audit events.

Agents take responsibility for disseminating audit data to clients and to other systems. The MonitorEndpointHandler in the *Reference* is an example of an audit agent.

To audit a Filter, Handler, or a route, you add an audit decoration. Audit decoration values are tags, which are strings useful to audit agents. Agents can filter audit events of interest based on tags and other conditions. The following example route has an audit decoration.

```
{
    "handler": "ClientHandler",
    "condition": "${matches(exchange.request.uri.query, 'site=com')}",
    "baseURI": "http://forgerock.com:80/",
    "audit": "ForgeRock.com route"
}
```

OpenIG creates an "audit" decorator, so you do not need to do so. For details on audit decorations, see the *Reference* for AuditDecorator in the *Reference*.

The "MonitorEndpointHandler" is a simple audit agent that collates basic statistics about the number of messages "in progress", "completed", and "internal errors" for each Filter or Handler that you have tagged, and returns the data in JSON format. To try auditing with the "MonitorEndpointHandler" agent, first set up the following routes.

- A route for the "MonitorEndpointHandler", `00-monitor.json`:

```
{
  "handler": {
    "type": "MonitorEndpointHandler"
  },
  "condition": "${exchange.request.method == 'GET'
                  and exchange.request.uri.path == '/openig/monitor'}"
}
```

- A route to ForgeRock.com, `08-com.json`:

```
{
    "handler": "ClientHandler",
    "condition": "${matches(exchange.request.uri.query, 'site=com')}",
    "baseURI": "http://forgerock.com:80/",
    "audit": "ForgeRock.com route"
}
```

- A route to ForgeRock.org, `08-org.json`:

```
{
    "handler": "ClientHandler",
    "condition": "${matches(exchange.request.uri.query, 'site=org')}",
    "baseURI": "https://forgerock.org:443/",
    "audit": "ForgeRock.org route"
}
```

- A default route with a static handler, `99-default.json`:

```
{
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "reason": "OK",
      "entity": "Hello from a static default route"
    }
  },
  "audit": "Static default route"
}
```

With the routes in place and OpenIG running, access the following route endpoints a few times to trigger audit events by using the following URLs.

http://www.example.com:8080/
http://www.example.com:8080/?site=com
http://www.example.com:8080/?site=org

After triggering a few audit events, access the monitor endpoint, http://www.example.com:8080/openig/monitor. You should see counts of the audit events in JSON format.

```
{
    "Static default route": {
        "in progress": 0,
        "completed": 14,
        "internal errors": 0
    },
    "ForgeRock.com route": {
        "in progress": 0,
        "completed": 16,
        "internal errors": 0
    },
    "ForgeRock.org route": {
        "in progress": 0,
        "completed": 15,
        "internal errors": 0
    },
    "global": {
        "in progress": 0,
        "completed": 45,
        "internal errors": 0
    }
}
```

According to the example output shown above, OpenIG successfully handled 16 requests on the "ForgeRock.com route", 15 requests on the "ForgeRock.org route", 14 default route requests for a total of 45 request completed on the top-level "global" route.

You can build your own audit agents that implement AuditEventListener for the audit agent logic and extend ConditionalAuditEventListener.ConditionalListenerHeaplet for the heaplet as described in the section on *Key Extension Points*. For instructions on bundling your custom audit agents, see *Building Customizations*.

Chapter 13 Troubleshooting

This chapter covers common problems and their solutions.

13.1. Object not found in heap

```
org.forgerock.json.fluent.JsonValueException: /handler:
   object Router2 not found in heap
    at org.forgerock.openig.heap.HeapImpl.resolve(HeapImpl.java:351)
    at org.forgerock.openig.heap.HeapImpl.resolve(HeapImpl.java:334)
    at org.forgerock.openig.heap.HeapImpl.getHandler(HeapImpl.java:538)
```

You have specified "handler": "Router2" in `config.json`, but no handler configuration object named "Router2" exists. Make sure you have added an entry for the handler and that you have correctly spelled its name.

13.2. Extra or missing character / invalid JSON

```
HTTP ERROR 500

Problem accessing /wp/wp-login.php. Reason:

    Server Error

Caused by:

    org.forgerock.openig.handler.HandlerException: no handler to dispatch to
```

A better description of the error appears as an error in the console log:

```
TUE DEC 02 17:35:57 CET 2014 (ERROR) {Router}/handler
The route defined in file '/Users/me/.openig/config/routes/route1.json'
 cannot be added
------------------------------
TUE DEC 02 17:35:57 CET 2014 (ERROR) {Router}/handler
Cannot read/parse content of /Users/me/.openig/config/routes/route1.json
[             HeapException] > Cannot read/parse content of
                              /Users/me/.openig/config/routes/route1.json
[       JsonParseException] > Unexpected character (',' (code 44)):
                              was expecting double-quote to start field name
 at [Source: java.io.InputStreamReader@4c8d1091; line: 4, column: 28]
```

In this case, extra comma is spotted at line 4, column 28.

Use a JSON editor or JSON validation tool such as JSONLint to make sure your JSON is valid.

13.3. The values in the flat file are incorrect

Ensure the flat file is readable by the user running the container for OpenIG. Values are all characters, including space and tabs, between the separator, so make sure the values are not padded with spaces.

13.4. Problem accessing URL

```
HTTP ERROR 500

Problem accessing /myURL . Reason:

java.lang.String cannot be cast to java.util.List
Caused by:
java.lang.ClassCastException: java.lang.String cannot be cast to java.util.List
```

This error is typically encountered when using the AssignmentFilter in the *Reference* and setting a string value for one of the Headers. All headers are stored in Lists so the header must be addressed with a subscript.

For example, if you try to set `exchange.request.headers['Location']` for a redirect in the response object, you should instead set `exchange.request.headers['Location'][0]`. A header without a subscript leads to the error above.

13.5. StaticResponseHandler results in a blank page

You must define an entity for the response as in the following example.

```
{
    "name": "AccessDeniedHandler",
    "type": "StaticResponseHandler",
    "config": {
        "status": 403,
        "reason": "Forbidden",
        "entity": "<html><p>User does not have permission</p></html>"
    }
}
```

13.6. OpenIG is not logging users in

If you are proxying to more than one application in multiple DNS domains, you must make sure your container is enabled for domain cookies. For details on your specific container, see the section on *Configuring Deployment Containers*.

13.7. Read timed out error when sending a request

If a "baseURI" configuration setting causes a request to come back to OpenIG, OpenIG never produces a response to the request. You then observe the following behavior.

You send a request and OpenIG seems to hang. Then you see a failure message, `HTTP Status 500 - Read timed out`, accompanied by OpenIG throwing an exception, `java.net.SocketTimeoutException: Read timed out`.

To fix this issue, make sure that "baseURI" configuration settings do not cause requests to come back to OpenIG.

13.8. OpenIG does not use new route configuration

OpenIG loads all configuration at startup. By default, it then periodically reloads changed route configurations.

If you make changes to a route that result in an invalid configuration, OpenIG logs errors, but it keeps the previous, correct configuration, and continues to use the old route.

OpenIG only uses the new configuration after you save a valid version or when you restart OpenIG.

Of course, if you restart OpenIG with an invalid route configuration, then OpenIG tries to load the invalid route at startup and logs an error. In that case, if there is no default handler to accept any incoming exchange for the invalid route, then you see an error, `No handler to dispatch to`.

13.9. Make OpenIG skip a route

If you have copied routes from another OpenIG server, those routes might depend on environment or container configuration that you have not yet configured locally.

You can work around this problem by changing the route file extension. A router ignores route files that do not have the `.json` extension.

For example, suppose you copy route all sample route configurations from the documentation, and then start OpenIG without first configuring your container. This can result in an error such as the following.

```
org.forgerock.json.fluent.JsonValueException:
 /handler/config/filters/0/config/dataSource:
 javax.naming.NameNotFoundException;
 remaining name 'jdbc/forgerock'
  at org.forgerock.openig.servlet.GatewayServlet.init(GatewayServlet.java:222)
  at org.eclipse.jetty.servlet.ServletHolder.initServlet(ServletHolder.java:595)
  at org.eclipse.jetty.servlet.ServletHolder.getServlet(ServletHolder.java:458)
  at org.eclipse.jetty.servlet.ServletHolder.handle(ServletHolder.java:724)
```

This arises from the route in `03-sql.json`, which defines an "SqlAttributesFilter" that depends on a JNDI data source configured in the container.

```
{
    "type": "SqlAttributesFilter",
    "config": {
        "dataSource": "java:comp/env/jdbc/forgerock",
        "preparedStatement":
          "SELECT username, password FROM users WHERE email = ?;",
        "parameters": [
            "george@example.com"
        ],
        "target": "${exchange.credentials}"
    }
}
```

To prevent OpenIG from loading the route configuration until you have had time to configure the container, change the file extension to render the route inactive.

```
$ mv ~/.openig/config/routes/03-sql.json ~/.openig/config/routes/03-sql.inactive
```

If necessary, restart the container to force OpenIG to reload the configuration.

When you have configured the data source in the container, change the file extension back to `.json` to render the route active again.

```
$ mv ~/.openig/config/routes/03-sql.inactive ~/.openig/config/routes/03-sql.json
```

Appendix A. SAML 2.0 & Multiple Applications

You can use a single OpenIG server as SAML 2.0 Service Provider for multiple protected applications.

A.1. Before You Start

Before you try the samples described here, familiarize yourself with OpenIG SAML 2.0 support by reading the chapter, *OpenIG as a SAML 2.0 Service Provider*, and working through the tutorial in that chapter.

Also make sure you understand the principles for configuring SAML 2.0 entities in OpenAM. The preparation for handling multiple applications involves editing the SAML 2.0 Service Provider configurations based on the original Fedlet configuration, and then importing the new configurations as SAML 2.0 entities in OpenAM.

At this point, you should have OpenIG protecting the sample application as SAML 2.0 Service Provider, with OpenAM working as Identity Provider configured as described in the tutorial.

A.2. Preparing the Network

You must configure the network so that browser traffic to the application hosts is proxied through OpenIG.

Modify DNS or host file settings so that the hosts name of the protected applications resolve to the IP address of OpenIG on the system where the browser runs. Restart the browser as necessary to take the changes into account.

The examples that follow use host names `sp.one.example` and `sp.two.example`. To try the examples on your computer, you can edit the host file settings to add these to the loopback address.

```
127.0.0.1    localhost www.example.com sp.one.example sp.two.example
```

A.3. Preparing the SAML 2.0 Service Provider Configurations

Based on the original Fedlet configuration, you add configuration for each new protected application.

In the following examples, the first application runs on host `sp.one.example`. The examples assign the entity ID `One` to this application, and use the metaAlias `/sp1` in the SAML configuration. The second application runs on `sp.two.example` with entity ID `Two` and metaAlias `/sp2`.

Edit the `SAML/fedlet.cot` file to include the entity IDs as in the following example.

```
cot-name=Circle of Trust
sun-fm-cot-status=Active
sun-fm-trusted-providers=http://openam.example.com:8088/openam,OpenIG,One,Two
sun-fm-saml2-readerservice-url=
sun-fm-saml2-writerservice-url=
```

For each application, make copies of the SAML configuration files `sp.xml` and `sp-extended.xml`. Edit the copy of `sp.xml` for the application so that the entity ID matches the application, the Location and ResponseLocation attributes reflect those of the application, and the AssertionConsumerService Location attributes include the metaAlias.

Example A.1. Service Provider Configuration for Application One

```
<!--
  Set the entityID and edit *Location attributes to match the service provider.
  Note that AssertionConsumerService Location attributes include the metaAlias.
-->
<EntityDescriptor
  entityID="One"
  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.one.example:8080/saml/fedletSloRedirect"
      ResponseLocation="http://sp.one.example:8080/saml/fedletSloRedirect"/>
    <SingleLogoutService
      Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
      Location="http://sp.one.example:8080/saml/fedletSloPOST"
      ResponseLocation="http://sp.one.example:8080/saml/fedletSloPOST"/>
    <SingleLogoutService
      Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
      Location="http://sp.one.example: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.one.example:8080/saml/fedletapplication/metaAlias/sp1"/>
    <AssertionConsumerService
      index="1"
      Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Artifact"
      Location="http://sp.one.example:8080/saml/fedletapplication/metaAlias/sp1"/>
  </SPSSODescriptor>
  <RoleDescriptor
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:query="urn:oasis:names:tc:SAML:metadata:ext:query"
    xsi:type="query:AttributeQueryDescriptorType"
    protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
  </RoleDescriptor>
  <XACMLAuthzDecisionQueryDescriptor
    WantAssertionsSigned="false"
    protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
  </XACMLAuthzDecisionQueryDescriptor>
</EntityDescriptor>
```

Example A.2. Service Provider Configuration for Application Two

```
<!--
  Set the entityID and edit *Location attributes to match the service provider.
  Note that AssertionConsumerService Location attributes include the metaAlias.
-->
<EntityDescriptor
  entityID="Two"
  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.two.example:8080/saml/fedletSloRedirect"
      ResponseLocation="http://sp.two.example:8080/saml/fedletSloRedirect"/>
    <SingleLogoutService
      Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-POST"
      Location="http://sp.two.example:8080/saml/fedletSloPOST"
      ResponseLocation="http://sp.two.example:8080/saml/fedletSloPOST"/>
    <SingleLogoutService
      Binding="urn:oasis:names:tc:SAML:2.0:bindings:SOAP"
      Location="http://sp.two.example: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.two.example:8080/saml/fedletapplication/metaAlias/sp2"/>
    <AssertionConsumerService
      index="1"
      Binding="urn:oasis:names:tc:SAML:2.0:bindings:HTTP-Artifact"
      Location="http://sp.two.example:8080/saml/fedletapplication/metaAlias/sp2"/>
  </SPSSODescriptor>
  <RoleDescriptor
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:query="urn:oasis:names:tc:SAML:metadata:ext:query"
    xsi:type="query:AttributeQueryDescriptorType"
    protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
  </RoleDescriptor>
  <XACMLAuthzDecisionQueryDescriptor
    WantAssertionsSigned="false"
    protocolSupportEnumeration="urn:oasis:names:tc:SAML:2.0:protocol">
  </XACMLAuthzDecisionQueryDescriptor>
</EntityDescriptor>
```

Edit the copy of `sp-extended.xml` for the application so that the entity ID matches the application, and the metaAlias and appLogoutUrl are correctly set.

Example A.3. Service Provider Extended Configuration for Application One

```
<!--
  Set the entityID and edit the SPSSOConfig metaAlias attribute.
  Also set the value of appLogoutUrl.
-->
<EntityConfig xmlns="urn:sun:fm:SAML:2.0:entityconfig"
    xmlns:fm="urn:sun:fm:SAML:2.0:entityconfig"
    hosted="1"
    entityID="One">

    <SPSSOConfig metaAlias="/sp1">
        <Attribute name="description">
            <Value></Value>
        </Attribute>
        <Attribute name="signingCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthOn">
            <Value>false</Value>
        </Attribute>
        <Attribute name="basicAuthUser">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthPassword">
            <Value></Value>
        </Attribute>
        <Attribute name="autofedEnabled">
            <Value>false</Value>
        </Attribute>
        <Attribute name="autofedAttribute">
            <Value></Value>
        </Attribute>
        <Attribute name="transientUser">
            <Value>anonymous</Value>
        </Attribute>
        <Attribute name="spAdapter">
            <Value></Value>
        </Attribute>
        <Attribute name="spAdapterEnv">
            <Value></Value>
        </Attribute>
        <Attribute name="fedletAdapter">
            <Value>com.sun.identity.saml2.plugins.DefaultFedletAdapter</Value>
        </Attribute>
        <Attribute name="fedletAdapterEnv">
            <Value></Value>
        </Attribute>
        <Attribute name="spAccountMapper">
            <Value>com.sun.identity.saml2.plugins.DefaultLibrarySPAccountMapper</Value>
        </Attribute>
        <Attribute name="useNameIDAsSPUserID">
            <Value>false</Value>
        </Attribute>
        <Attribute name="spAttributeMapper">
            <Value>com.sun.identity.saml2.plugins.DefaultSPAttributeMapper</Value>
        </Attribute>
        <Attribute name="spAuthncontextMapper">
            <Value>com.sun.identity.saml2.plugins.DefaultSPAuthnContextMapper</Value>
        </Attribute>
        <Attribute name="spAuthncontextClassrefMapping">
            <Value>
            urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport|0|default
            </Value>
        </Attribute>
        <Attribute name="spAuthncontextComparisonType">
           <Value>exact</Value>
        </Attribute>
        <Attribute name="attributeMap">
           <Value>employeenumber=employeenumber</Value>
           <Value>mail=mail</Value>
        </Attribute>
        <Attribute name="saml2AuthModuleName">
           <Value></Value>
       </Attribute>
       <Attribute name="localAuthURL">
           <Value></Value>
       </Attribute>
       <Attribute name="intermediateUrl">
           <Value></Value>
       </Attribute>
       <Attribute name="defaultRelayState">
           <Value></Value>
       </Attribute>
       <Attribute name="appLogoutUrl">
           <Value>http://sp.one.example:8080/saml/logout</Value>
       </Attribute>
       <Attribute name="assertionTimeSkew">
           <Value>300</Value>
       </Attribute>
       <Attribute name="wantAttributeEncrypted">
           <Value></Value>
       </Attribute>
       <Attribute name="wantAssertionEncrypted">
           <Value></Value>
       </Attribute>
       <Attribute name="wantNameIDEncrypted">
           <Value></Value>
       </Attribute>
       <Attribute name="wantPOSTResponseSigned">
           <Value></Value>
       </Attribute>
       <Attribute name="wantArtifactResponseSigned">
           <Value></Value>
       </Attribute>
       <Attribute name="wantLogoutRequestSigned">
           <Value></Value>
       </Attribute>
       <Attribute name="wantLogoutResponseSigned">
           <Value></Value>
       </Attribute>
       <Attribute name="wantMNIRequestSigned">
```

```
            <Value></Value>
        </Attribute>
        <Attribute name="wantMNIResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="responseArtifactMessageEncoding">
            <Value>URI</Value>
        </Attribute>
        <Attribute name="cotlist">
        <Value>Circle of Trust</Value></Attribute>
        <Attribute name="saeAppSecretList">
        </Attribute>
        <Attribute name="saeSPUrl">
            <Value></Value>
        </Attribute>
        <Attribute name="saeSPLogoutUrl">
        </Attribute>
        <Attribute name="ECPRequestIDPListFinderImpl">
            <Value>com.sun.identity.saml2.plugins.ECPIDPFinder</Value>
        </Attribute>
        <Attribute name="ECPRequestIDPList">
            <Value></Value>
        </Attribute>
        <Attribute name="ECPRequestIDPListGetComplete">
            <Value></Value>
        </Attribute>
        <Attribute name="enableIDPProxy">
            <Value>false</Value>
        </Attribute>
        <Attribute name="idpProxyList">
            <Value></Value>
        </Attribute>
        <Attribute name="idpProxyCount">
            <Value>0</Value>
        </Attribute>
        <Attribute name="useIntroductionForIDPProxy">
            <Value>false</Value>
        </Attribute>
        <Attribute name="spSessionSyncEnabled">
            <Value>false</Value>
        </Attribute>
         <Attribute name="relayStateUrlList">
         </Attribute>
     </SPSSOConfig>
     <AttributeQueryConfig metaAlias="/attrQuery">
         <Attribute name="signingCertAlias">
             <Value></Value>
         </Attribute>
         <Attribute name="encryptionCertAlias">
             <Value></Value>
         </Attribute>
         <Attribute name="wantNameIDEncrypted">
             <Value></Value>
         </Attribute>
         <Attribute name="cotlist">
             <Value>Circle of Trust</Value>
         </Attribute>
     </AttributeQueryConfig>
     <XACMLAuthzDecisionQueryConfig metaAlias="/pep">
```

```
        <Attribute name="signingCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthOn">
            <Value>false</Value>
        </Attribute>
        <Attribute name="basicAuthUser">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthPassword">
            <Value></Value>
        </Attribute>
        <Attribute name="wantXACMLAuthzDecisionResponseSigned">
            <Value>false</Value>
        </Attribute>
        <Attribute name="wantAssertionEncrypted">
            <Value>false</Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </XACMLAuthzDecisionQueryConfig>
</EntityConfig>
```

Example A.4. Service Provider Extended Configuration for Application Two

```
<!--
  Set the entityID and edit the SPSSOConfig metaAlias attribute.
  Also set the value of appLogoutUrl.
-->
<EntityConfig xmlns="urn:sun:fm:SAML:2.0:entityconfig"
    xmlns:fm="urn:sun:fm:SAML:2.0:entityconfig"
    hosted="1"
    entityID="Two">

    <SPSSOConfig metaAlias="/sp2">
        <Attribute name="description">
            <Value></Value>
        </Attribute>
        <Attribute name="signingCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthOn">
            <Value>false</Value>
        </Attribute>
        <Attribute name="basicAuthUser">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthPassword">
            <Value></Value>
        </Attribute>
        <Attribute name="autofedEnabled">
            <Value>false</Value>
        </Attribute>
        <Attribute name="autofedAttribute">
            <Value></Value>
        </Attribute>
        <Attribute name="transientUser">
            <Value>anonymous</Value>
        </Attribute>
        <Attribute name="spAdapter">
            <Value></Value>
        </Attribute>
        <Attribute name="spAdapterEnv">
            <Value></Value>
        </Attribute>
        <Attribute name="fedletAdapter">
            <Value>com.sun.identity.saml2.plugins.DefaultFedletAdapter</Value>
        </Attribute>
        <Attribute name="fedletAdapterEnv">
            <Value></Value>
        </Attribute>
        <Attribute name="spAccountMapper">
            <Value>com.sun.identity.saml2.plugins.DefaultLibrarySPAccountMapper</Value>
        </Attribute>
        <Attribute name="useNameIDAsSPUserID">
            <Value>false</Value>
```

```
         </Attribute>
         <Attribute name="spAttributeMapper">
             <Value>com.sun.identity.saml2.plugins.DefaultSPAttributeMapper</Value>
         </Attribute>
         <Attribute name="spAuthncontextMapper">
             <Value>com.sun.identity.saml2.plugins.DefaultSPAuthnContextMapper</Value>
         </Attribute>
         <Attribute name="spAuthncontextClassrefMapping">
             <Value>
             urn:oasis:names:tc:SAML:2.0:ac:classes:PasswordProtectedTransport|0|default
             </Value>
         </Attribute>
         <Attribute name="spAuthncontextComparisonType">
            <Value>exact</Value>
         </Attribute>
         <Attribute name="attributeMap">
            <Value>employeenumber=employeenumber</Value>
            <Value>mail=mail</Value>
         </Attribute>
         <Attribute name="saml2AuthModuleName">
            <Value></Value>
        </Attribute>
        <Attribute name="localAuthURL">
            <Value></Value>
        </Attribute>
        <Attribute name="intermediateUrl">
            <Value></Value>
        </Attribute>
        <Attribute name="defaultRelayState">
            <Value></Value>
        </Attribute>
        <Attribute name="appLogoutUrl">
            <Value>http://sp.two.example:8080/saml/logout</Value>
        </Attribute>
        <Attribute name="assertionTimeSkew">
            <Value>300</Value>
        </Attribute>
        <Attribute name="wantAttributeEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="wantAssertionEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="wantNameIDEncrypted">
            <Value></Value>
        </Attribute>
        <Attribute name="wantPOSTResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantArtifactResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantLogoutRequestSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantLogoutResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="wantMNIRequestSigned">
```

```
            <Value></Value>
        </Attribute>
        <Attribute name="wantMNIResponseSigned">
            <Value></Value>
        </Attribute>
        <Attribute name="responseArtifactMessageEncoding">
            <Value>URI</Value>
        </Attribute>
        <Attribute name="cotlist">
        <Value>Circle of Trust</Value></Attribute>
        <Attribute name="saeAppSecretList">
        </Attribute>
        <Attribute name="saeSPUrl">
            <Value></Value>
        </Attribute>
        <Attribute name="saeSPLogoutUrl">
        </Attribute>
        <Attribute name="ECPRequestIDPListFinderImpl">
            <Value>com.sun.identity.saml2.plugins.ECPIDPFinder</Value>
        </Attribute>
        <Attribute name="ECPRequestIDPList">
            <Value></Value>
        </Attribute>
        <Attribute name="ECPRequestIDPListGetComplete">
            <Value></Value>
        </Attribute>
        <Attribute name="enableIDPProxy">
            <Value>false</Value>
        </Attribute>
        <Attribute name="idpProxyList">
            <Value></Value>
        </Attribute>
        <Attribute name="idpProxyCount">
            <Value>0</Value>
        </Attribute>
        <Attribute name="useIntroductionForIDPProxy">
            <Value>false</Value>
        </Attribute>
        <Attribute name="spSessionSyncEnabled">
            <Value>false</Value>
        </Attribute>
         <Attribute name="relayStateUrlList">
         </Attribute>
    </SPSSOConfig>
    <AttributeQueryConfig metaAlias="/attrQuery">
         <Attribute name="signingCertAlias">
             <Value></Value>
         </Attribute>
         <Attribute name="encryptionCertAlias">
             <Value></Value>
         </Attribute>
         <Attribute name="wantNameIDEncrypted">
             <Value></Value>
         </Attribute>
         <Attribute name="cotlist">
             <Value>Circle of Trust</Value>
         </Attribute>
    </AttributeQueryConfig>
    <XACMLAuthzDecisionQueryConfig metaAlias="/pep">
```

```
        <Attribute name="signingCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="encryptionCertAlias">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthOn">
            <Value>false</Value>
        </Attribute>
        <Attribute name="basicAuthUser">
            <Value></Value>
        </Attribute>
        <Attribute name="basicAuthPassword">
            <Value></Value>
        </Attribute>
        <Attribute name="wantXACMLAuthzDecisionResponseSigned">
            <Value>false</Value>
        </Attribute>
        <Attribute name="wantAssertionEncrypted">
            <Value>false</Value>
        </Attribute>
        <Attribute name="cotlist">
            <Value>Circle of Trust</Value>
        </Attribute>
    </XACMLAuthzDecisionQueryConfig>
</EntityConfig>
```

For each of the service provider extended configuration files, prepare a copy for use when importing the configuration into OpenAM. The only change to make in each copy is to set `hosted="0"`, so that when you import the configuration into OpenAM, OpenAM considers it that of a *remote* service provider.

A.4. Importing Service Provider Configurations Into OpenAM

For each new protected application, import a SAML 2.0 entity into OpenAM.

1. Login to OpenAM console as global administrator (`amadmin`).
2. On the Federation tab > Entity Providers table, click Import Entity.
3. Import the entity using the metadata from the edited copies of `sp.xml` and `sp-extended.xml`, where the copy of `sp-extended.xml` has `hosted="0"`.

   The service provider configurations should have Location `Remote` in the Entity Providers table.
4. Log out of OpenAM Console.

A.5. Preparing Configurations in OpenIG

For each new protected application, prepare a OpenIG configuration. The configurations in this section follow the example in the chapter, *OpenIG as a SAML 2.0 Service Provider*.

Before editing route configurations for the protected applications, configure a top-level router that does not rebase the incoming URLs, such as the following `config.json`. This differs from the example used in earlier tutorials.

```
{
    "handler": {
        "type": "Router"
    },
    "heap": [
        {
            "name": "LogSink",
            "type": "ConsoleLogSink",
            "config": {
                "level": "DEBUG"
            }
        },
        {
            "name": "ClientHandler",
            "type": "ClientHandler"
        },
        {
            "name": "capture",
            "type": "CaptureDecorator",
            "config": {
                "captureEntity": true,
                "captureExchange": true
            }
        }
    ]
}
```

Also restart OpenIG to put all configuration changes into effect.

For each application set up a pair of routes, one to handle redirection for SAML authentication and login to the application, the other to act as the SAML 2.0 assertion consumer that maps attributes from the SAML assertion into the exchange and redirects back to the first route.

The following examples show the routes for application One.

Example A.5. Route for SAML Authentication & Login: Application One

```
{
    "handler": {
        "type": "DispatchHandler",
        "config": {
            "bindings": [
                {
                    "condition": "${empty exchange.session.sp1Username}",
                    "handler": {
                        "type": "StaticResponseHandler",
                        "config": {
                            "status": 302,
                            "reason": "Found",
                            "headers": {
                                "Location": [
                                    "http://sp.one.example:8080/saml/SPInitiatedSSO?metaAlias=/sp1"
                                ]
                            }
                        }
                    },
                    "baseURI": "http://sp.one.example:8081"
                },
                {
                    "handler": {
                        "type": "Chain",
                        "config": {
                            "filters": [
                                {
                                    "type": "StaticRequestFilter",
                                    "config": {
                                        "method": "POST",
                                        "uri": "http://sp.one.example:8081",
                                        "form": {
                                            "username": [
                                                "${exchange.session.sp1Username}"
                                            ],
                                            "password": [
                                                "${exchange.session.sp1Password}"
                                            ]
                                        }
                                    }
                                }
                            ],
                            "handler": "ClientHandler"
                        }
                    },
                    "baseURI": "http://sp.one.example:8081"
                }
            ]
        }
    },
    "condition": "${matches(exchange.request.uri.host, 'sp.one.example')}"
}
```

Example A.6. SAML Assertion Consumer: Application One

```
{
    "handler": {
        "type": "SamlFederationHandler",
        "config": {
            "comment": "Use unique session properties for this SP.",
            "assertionMapping": {
                "sp1Username": "mail",
                "sp1Password": "employeenumber"
            },
            "authnContext": "sp1AuthnContext",
            "sessionIndexMapping": "sp1SessionIndex",
            "subjectMapping": "sp1SubjectName",
            "redirectURI": "/sp1"
        }
    },
    "condition": "${matches(exchange.request.uri.host, 'sp.one.example')
                    and matches(exchange.request.uri.path, '^/saml')}"
}
```

The following examples show the routes for application Two.

Example A.7. Route for SAML Authentication & Login: Application Two

```
{
    "handler": {
        "type": "DispatchHandler",
        "config": {
            "bindings": [
                {
                    "condition": "${empty exchange.session.sp2Username}",
                    "handler": {
                        "type": "StaticResponseHandler",
                        "config": {
                            "status": 302,
                            "reason": "Found",
                            "headers": {
                                "Location": [
                                    "http://sp.two.example:8080/saml/SPInitiatedSSO?metaAlias=/sp2"
                                ]
                            }
                        }
                    },
                    "baseURI": "http://sp.two.example:8081"
                },
                {
                    "handler": {
                        "type": "Chain",
                        "config": {
                            "filters": [
                                {
                                    "type": "StaticRequestFilter",
                                    "config": {
                                        "method": "POST",
                                        "uri": "http://sp.two.example:8081",
                                        "form": {
                                            "username": [
                                                "${exchange.session.sp2Username}"
                                            ],
                                            "password": [
                                                "${exchange.session.sp2Password}"
                                            ]
                                        }
                                    }
                                }
                            ],
                            "handler": "ClientHandler"
                        }
                    },
                    "baseURI": "http://sp.two.example:8081"
                }
            ]
        }
    },
    "condition": "${matches(exchange.request.uri.host, 'sp.two.example')}"
}
```

Example A.8. SAML Assertion Consumer: Application Two

```
{
    "handler": {
        "type": "SamlFederationHandler",
        "config": {
            "comment": "Use unique session properties for this SP.",
            "assertionMapping": {
                "sp2Username": "mail",
                "sp2Password": "employeenumber"
            },
            "authnContext": "sp2AuthnContext",
            "sessionIndexMapping": "sp2SessionIndex",
            "subjectMapping": "sp2SubjectName",
            "redirectURI": "/sp2"
        }
    },
    "condition": "${matches(exchange.request.uri.host, 'sp.two.example')
                    and matches(exchange.request.uri.path, '^/saml')}"
}
```

A.6. Trying It Out

Try the configuration for multiple protected applications, logging in to OpenAM as for the single SP federation example with username `george`, password `costanza`.

If you use the example configurations described here with all services running on your computer protecting the sample application, then you can try the SAML 2.0 web single sign-on profile with application One by using either of the following links.

- The link for SP initiated SSO.
- The link for IDP initiated SSO.

Similarly you can try the SAML 2.0 web single sign-on profile with application Two by using either of the following links.

- The link for SP initiated SSO.
- The link for IDP initiated SSO.

If you have not configured the examples exactly as shown in this guide, then adapt the SSO links accordingly.

Index