Gateway Guide

OpenIG 3.0.0

Paul Bryan
Mark Craig
Jamie Nelson

ForgeRock AS
33 New Montgomery St., Suite 1500
San Francisco, CA 94105, USA
+1 415-523-0772
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
Solutions Overview

ForgeRock OpenIG provides the answer to three important challenges.

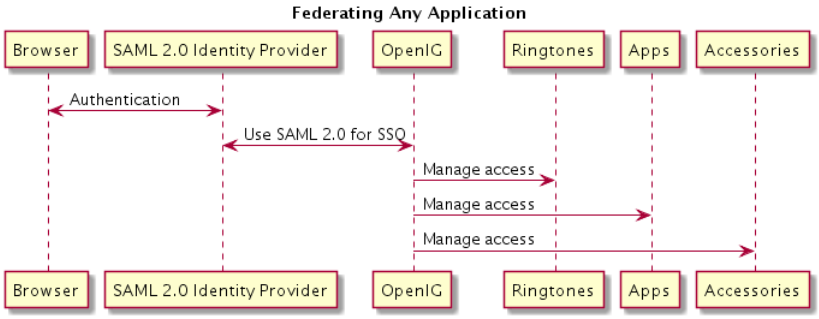
1.1. Extend SSO To Any Application

In today's enterprise, more than 30% of web applications are incompatible with web access management (WAM) software. That is, unlike OpenAM with OpenIG, most web access management products lack the agent to protect the web applications, or the application is a legacy solution that does not follow standard protocols for single sign-on. This limits the return on the enterprise WAM investment and constrains what types of web applications can be protected.

ForgeRock OpenIG addresses this problem by extending access management to encompass *all* web applications. With OpenIG, OpenAM deployments can now be extended to be inclusive of those applications that do not integrate with policy agents alone. In addition, ForgeRock OpenIG interoperates, out-of-the-box, with all management solutions. Most importantly, your organization can on-board any web application without ever modifying or touching the target application again, significantly reducing the development and quality assurance required to protect web applications.

1.2. Federate Enabling Applications

The expertise and cost required to add SAML 2.0 support to web applications is a problem for many businesses. Those businesses not moving to a standard for exposing their applications to their customers see increased cost and maintenance due to the complexity of one-off proprietary integrations. They can also see a loss of business to those customers requiring a Federation standard for authentication. Sometimes, deploying a full access management solution just to federate a few applications is is too complex and costly, and building out their own solution by modifying their applications is just not possible.



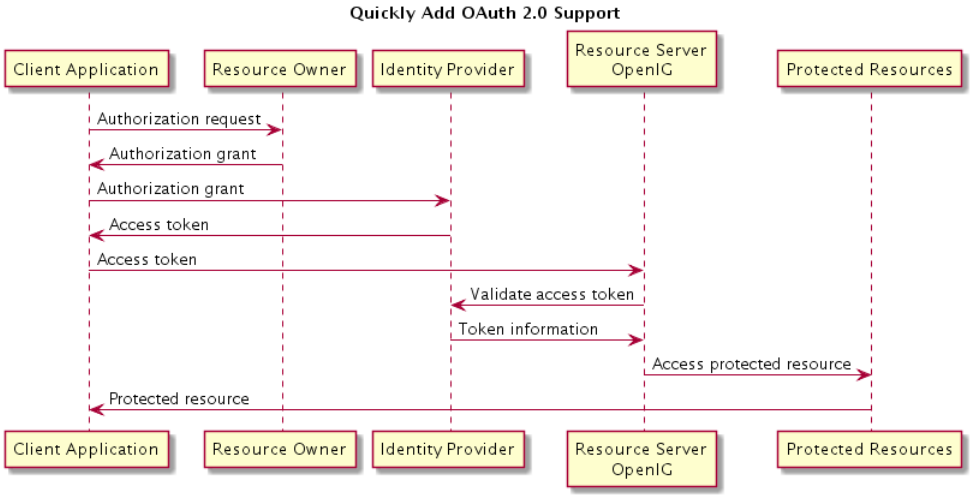
OpenIG Federation allows businesses to add SAML 2.0 support to their applications with little to no knowledge of the standard. In addition, there is no need to modify the application or to install any plugin or policy agent on the application container.

1.3. Quickly Add OAuth 2.0 & OpenID Connect 1.0 Support

OAuth 2.0 is modern federation alternative, aimed at making it easy for users to delegate third-party applications access to their protected resources without having to share credentials with the third-party applications. Like federation, OAuth 2.0 lets users benefit from new services without having to create new accounts. Instead they can use an existing account with an OAuth 2.0-compliant identity provider, such as Facebook, Google, or any provider using OpenAM. Many mobile and web applications are moving to use OAuth 2.0.

With OpenIG, you can add OAuth 2.0 support to existing protected resources, and use OpenIG capabilities to quickly develop new OAuth 2.0 applications. OpenIG can protect resources by using OAuth 2.0 (as resource server), interoperating with client applications and identity providers, so that your applications can rely on delegated authorization for access to users' protected resources.

The following figure shows OpenIG playing the role of resource server in an OAuth 2.0 authorization code grant flow.

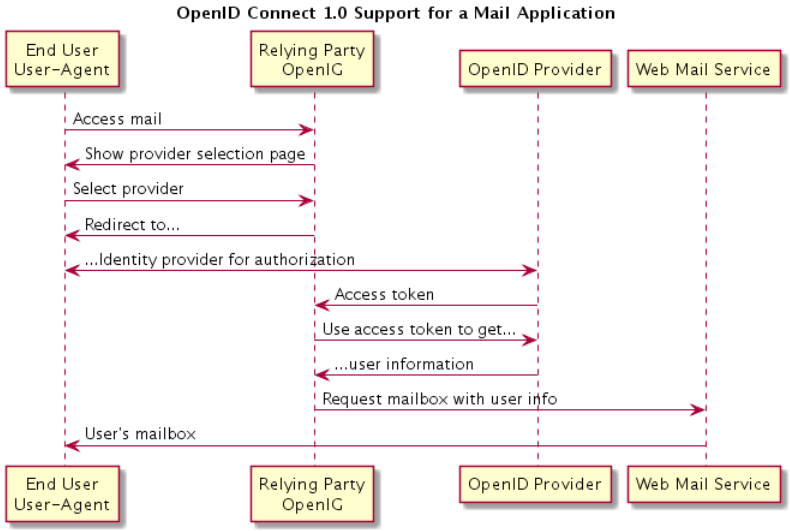


OpenIG can also play the role of OAuth 2.0 client, authenticating an end user using OAuth 2.0 delegated authorization with an existing OAuth 2.0-compliant identity provider. This means developers can quickly develop OAuth 2.0 client applications without having to learn the intricacies of OAuth 2.0 authorization grant flows. OpenIG takes care of obtaining the access token on behalf of your application and injects the access token for your use.

OpenID Connect 1.0 is an implementation of OAuth 2.0 whereby the protected resource is information about the user or about the user's account with the identity provider.

As OAuth 2.0 client, OpenIG can play the role of OpenID Connect 1.0 relying party, obtaining user information on behalf of your application.

The following figure shows OpenIG playing the role of relying party, where the mail server requires user information to show the right mailboxes.



In addition, there is generally no need to modify existing applications or servers housing protected resources or to install plugins or policy agents on the servers.

OpenIG can for example inject user information into the URL, adding a *username* as in `https://mail.example.com/mail/username`. For some applications, you might choose to provide additional capabilities, and so you might want to enhance the application to accept additional user information from the provider in other ways.

Chapter 2

How OpenIG Works

This chapter provides a detailed look at OpenIG components and how they work together.

2.1. How It Works In A Nutshell

The underlying core of ForgeRock OpenIG is based on a reverse proxy architecture. All HTTP traffic to each protected application is routed through OpenIG, enabling close inspection, transformation and filtering of each request. By inspecting the traffic, OpenIG is able to intercept requests that would normally require the user to authenticate, obtain the user's login credentials, and send the necessary HTTP request to the target application, thereby logging in the user without modifying or installing anything on the application. In its simplest form and basic configuration, OpenIG is a Java-based reverse proxy which runs as a web application. Enable the Form-Filter replay module and OpenIG automatically log users in when a timeout or authentication page is detected. Additionally, enable the SAML2 service and OpenIG becomes a SAML2 endpoint. In this mode of operation, OpenIG receives and verifies the SAML2 request and then logs the user directly into the target application.

2.2. How It Works In Detail

The following modules make up OpenIG.

2.2.1. OpenIG Core (Reverse Proxy)

OpenIG core is a standard Java EE servlet implementation of a reverse proxy. The main function of OpenIG core is to act as a reverse proxy to the target application. When deployed in its base configuration, OpenIG can be used as a pure reverse proxy. The power of the OpenIG core comes in its ability to search, transform, and process HTTP traffic to and from the target application. This enables OpenIG to recognize login pages, submit login forms, transform or filter content, and even function as a Federation endpoint for the application. All these features are possible without making any changes to the application's deployment container or the application itself.

2.2.2. Exchange

The Exchange in the *Reference* is a wrapper around the HTTP request and response objects that pass through OpenIG. Every request or response being processed in OpenIG can be accessed or modified

through the Exchange object. In addition, arbitrary data can be set in the Exchange to facilitate the passing of data and state between filters and handlers.

2.2.3. Router & Routes

In a multiple-file configuration, the Router in the *Reference* takes responsibility for managing Route in the *Reference* configurations.

Each Route accepts all Exchanges that match its (configurable) condition. The Route then optionally changes the request scheme, host, and port, and forwards the Exchange on to the configured chain of filters and handlers.

The Router can be configured either to load Route configurations only at startup or to reload Route configurations periodically, picking up any changes you make. You can thus reconfigure OpenIG without restarting the server.

2.2.4. Dispatcher

In a single-file configuration with no Router, the Dispatcher may be thought of as the internal router of OpenIG. Every request that comes into OpenIG is analyzed and forwarded on to the configured processing chain of filters and handlers. A request may be forwarded based on the target host, URL, URL parameters, headers, cookies, or any other component of the request.

2.2.5. Chain

A Chain in the *Reference* is a combination of one or more Filters and a handler that process an incoming request from the Dispatcher. For example, the Dispatcher can process an incoming request with a URL parameter of `action=login` and forward the request to the Login Chain. The Login Chain executes a list of Filters and then calls a Handler. The Handler sends the request on to the target application or to another Chain for further processing.

2.2.6. Handlers

The final processing of every Chain ends in a call to a Handler. A Handler can simply call another Chain or it can send the request on to the target application. The following Handlers are shipped with OpenIG:

- ClientHandler in the *Reference*: Sends the final request to the target application.

  You can specify HTTP client settings for requests to remote servers by adding an HttpClient in the *Reference* in the configuration for the `ClientHandler`.

- DispatchHandler in the *Reference*: Dispatches to one of a list of handlers.

- Route in the *Reference*: Allows you to configure a separate JSON configuration file that handles an Exchange when a specified condition is met.

- Router in the *Reference*: Routes Exchange processing to separate configuration files.
- SamlFederationHandler: Handles assertions from a SAML 2.0 IDP.
- ScriptableHandler in the *Reference*: Handles a request by using a script.
- SequenceHandler in the *Reference*: Links together multiple handlers or chains during request processing.
- StaticResponseHandler in the *Reference*: Used to send a response, such as a redirect, to a client during request processing.

2.2.7. Filters

Filters are responsible for processing HTTP requests in OpenIG. Filters can be chained together to act on the input stream coming from the browser, or the output stream returned back from the target application. A filter can do something as simple as logging the input and output stream or something more complex, such as processing login pages, fetching user attributes, or transforming content. There are multiple Filters shipped with OpenIG that can be combined in chains to provide very extensible request and response processing features. Custom filters can also be written using the Java SPIs. The following is a list of Filters shipped with OpenIG:

- AssignmentFilter in the *Reference*: Sets values in the HTTP request and response.
- CaptureFilter in the *Reference*: Captures the HTTP requests being processed by OpenIG. Capture can be used for audit purposes and may also be very useful when analyzing an application or troubleshooting a misbehaving OpenIG. Logs are written to a flat file on the OpenIG host.
- CookieFilter in the *Reference*: The default behavior of OpenIG is to accept and forward all cookies. Since this is not always the desirable behavior, the CookieFilter, when configured, allows you to suppress, manage, and relay cookies.
- CryptoHeaderFilter in the *Reference*: Encrypts or decrypts headers in a request or response.
- EntityExtractFilter in the *Reference*: Searches for specific content within the body of the requests. For example, it can be used to extract hidden form parameters in a login page, which are needed in the login request.
- ExceptionFilter in the *Reference*: Sends users to configured URLs when errors or exceptions occur during request processing or user interactions.
- FileAttributesFilter in the *Reference*: Looks up attributes in a flat file with the specified key. The attributes are added to the exchange to be used by subsequent filters or handlers.
- HeaderFilter in the *Reference*: The default behavior of OpenIG is to accept and forward all headers. The HeaderFilter can be configured to add additional headers or remove headers on both the HTTP request and the response. It can also be configured to parse and set header values in OpenIG context to allow filters access to the header attributes. This feature is used most commonly when OpenIG is integrated with OpenAM and being fronted by a policy agent.

- HttpBasicAuthFilter in the *Reference*: Performs HTTP basic authentication to the target application per RFC 2617.
- OAuth2ClientFilter in the *Reference*: acts as an OAuth 2.0 or OpenID Connect 1.0 client, authenticating an end user using OAuth 2.0 delegated authorization.
- OAuth2ResourceServerFilter in the *Reference*: acts as an OAuth 2.0 resource server, validating OAuth 2.0 access tokens, and injecting token information into the exchange.
- RedirectFilter in the *Reference*: Rewrites Location headers on responses that generate a redirect that would take the user directly to the application being proxied rather than taking the user through OpenIG.
- ScriptableFilter in the *Reference*: Processes the HTTP exchange by using a script.
- SqlAttributesFilter in the *Reference*: Executes an SQL prepared statement with configured parameters. The result is added to the exchange to be used by subsequent filters or handlers.
- StaticRequestFilter in the *Reference*: Creates and sends HTTP GET and POST requests. The request can be formed using parameters from previous processing or statically configured values.
- SwitchFilter in the *Reference*: Conditionally diverts the exchange to another handler.

2.2.8. Configuration

The configuration of OpenIG was designed to be very modular and self-contained. Each module within OpenIG stores its configuration in JSON representation, which is stored in flat files. The features of OpenIG can be configured by directly manipulating the JSON flat files.

2.2.9. Heaplets

Every OpenIG module which has JSON configuration also has a Heaplet associated with it. Each module's Heaplet is responsible for reading the JSON configuration and creating that module's configuration in the OpenIG runtime heap. Each module can then read its configuration from the heap as well as make shared configuration information available to other modules.

2.2.10. OpenIG SAML 2.0 Federation

When the Federation component is configured, OpenIG acts as the Service Provider in a Circle of Trust with a SAML 2.0-compliant Identity Provider. The Federation component supports both IDP and SP-initiated SAML 2.0 Web Single Sign-On. OpenIG Federation can serve as a a Service Provider in the classic Federation use case where the IDP and SP are different companies or domains.

Chapter 3
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 *Installing OpenIG*.

3.1. 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*.

3.2. 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-3.0.0.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.

3.3. 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-3.0.0-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-3.0.0-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.

3.4. 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.

   Copy the basic configuration file from the example, *Configuration for Proxy & Capture*, to `$HOME/.openig/config/config.json`. By default, OpenIG looks for `config.json` in the `$HOME/.openig/config` directory.

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

   Also on Windows, change the capture log file name in `config.json` from `/tmp/gateway.log` to a file system location that works for Windows systems, such as `C:\Temp\gateway.log`.

2. 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
```

3.5. 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`.

3.6. Try It Out

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

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

1. Edit the configuration file, `config.json`, to automatically log you in as username `demo`, password `password`.

   To do this, make the following two changes to the file.

   a. Between the `"OutgoingChain"` object and the "CaptureFilter" object, add a new `StaticRequestFilter` configuration object.

      The `StaticRequestFilter` performs an HTTP POST of the username and password as form data. OpenIG supports a variety of ways to get the credentials, but for now just hard code them into the configuration.

      The new filter configuration object should look like this:

      ```
      {
          "name": "LoginRequest",
          "type": "StaticRequestFilter",
          "config": {
              "method": "POST",
              "uri": "http://www.example.com:8081",
              "form": {
                  "username": [
                      "demo"
                  ],
                  "password": [
                      "changeit"
                  ]
              }
          }
      }
      ```

      Do not forget to add the comma after the object, so that your configuration file remains valid JSON.

   b. Edit the list of filters in the "OutgoingChain" object to include your new filter before the "CaptureFilter".

      The full "OutgoingChain" object should now look like this:

```
{
    "name": "OutgoingChain",
    "type": "Chain",
    "config": {
        "filters": [
            "LoginRequest",
            "CaptureFilter"
        ],
        "handler": "DefaultHandler"
    }
}
```

Make sure that the configuration is valid JSON, and then save your changes.

The resulting configuration file should be very close to the example, *Configuration for Hard-Coded Credentials*.

2. Restart the Jetty server where OpenIG is deployed.

3. Access the home page again, http://www.example.com:8080/.

   This time, OpenIG logs you in automatically.

Also view the file capturing information about requests and responses. In the default sample configuration, this file is `/tmp/gateway.log`.

What's happening behind the scenes?

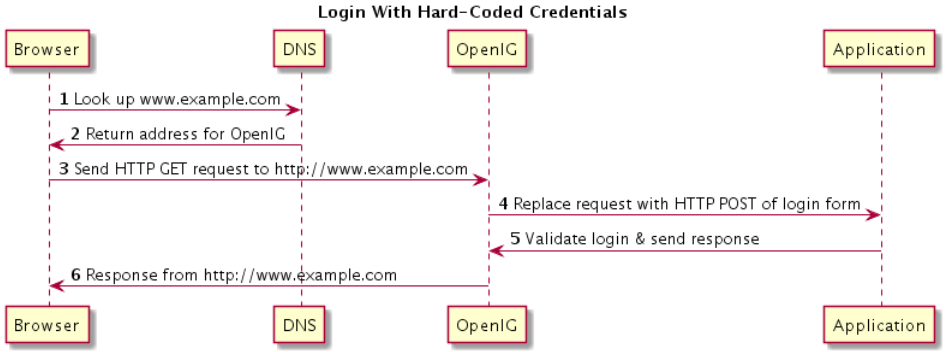
When your browser goes to `http://www.example.com:8080/`, it is actually connecting to OpenIG deployed in Jetty. OpenIG proxies all traffic it receives to the protected application at `http://www.example.com:8080/`, and returns responses from the application to your browser.

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. And OpenIG also 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 3.1.



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.

3.7. Where To Go From Here

In this chapter, you have scratched the surface of OpenIG. For more information, start with these chapters.

Installing OpenIG

This chapter covers everything you need to install OpenIG.

Tutorial On Looking Up Credentials

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

Tutorial For OpenIG Federation

This chapter shows how to configure OpenIG to get credentials from a SAML 2.0 Identity Provider.

Configuring 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.

Configuring 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.

Routing Tutorial

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 4

Installing OpenIG

This chapter covers everything you need to install OpenIG.

- 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 4.1, "Configuring Deployment Containers".

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

  For details, see Section 4.2, "Preparing the Network".

- Download, deploy, and configure OpenIG.

  For details, see Section 4.3, "Installing OpenIG".

4.1. 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.

4.1.1. 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.

The web application container where OpenIG runs is responsible for setting up TLS connections.

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.[1]

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 example, you can set the key store and password, the trust store and password (useful when OpenIG acts as a client), the cipher suites to enable for use, and other properties. For a list of

[1] When OpenIG acts as a client of a protected application or other server whose certificate is not recognized out of the box by the Java environment, then you must also install that certificate in the key store for the OpenIG web application container.

The following command installs a trusted signer's certificate, `ca.crt`, in a Java Key Store file.

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

The `-trustcacerts` option says to trust this as a signing certificate, and so works both with self-signed certificates and also with certificates used to sign server certificates.

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

4.1.2. 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.

4.1.2.1. 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.

4.1.2.2. Configuring Tomcat For HTTPS

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.

4.1.2.3. 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.

4.1.3. 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.

4.1.3.1. 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.

4.1.3.2. Configuring Jetty For HTTPS

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.

4.1.3.3. 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.

4.2. 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.

4.3. Installing OpenIG

Follow these steps to install OpenIG.

1. Download the OpenIG .war file.

   OpenIG can be downloaded from https://backstage.forgerock.com/downloads.

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, *Routing Tutorial*, 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, *Scripting Filters & Handlers*, for more information.

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

   OpenIG temporary files.

   This location can be used for capture files and 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 for Proxy & Capture*.

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.

On Windows, also edit the location of the capture log 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 capture log to see that traffic is going through OpenIG.

  The location for the capture log is set in `config.json`, by default `/tmp/gateway.log`.

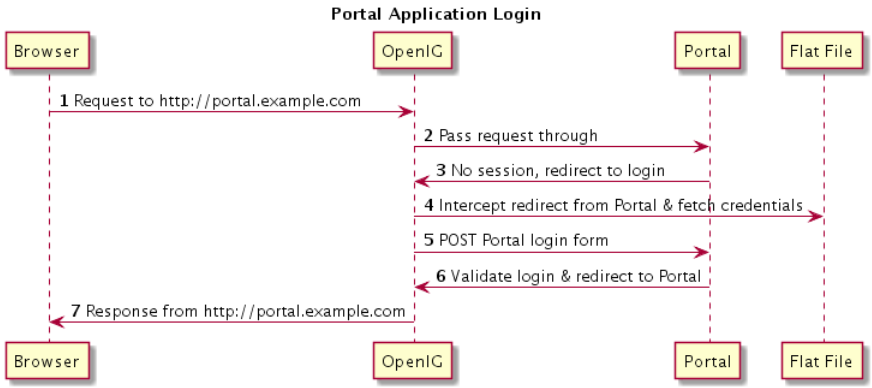
Chapter 5

Detailed Use Cases

This chapter diagrams and describes common use case request and response flows.

5.1. Portal Application Login

The figure below illustrates a sample flow with a description of each request from the browser to the back end application. This flow is based on the tutorial, *Login With Credentials From a File*. Try the tutorial yourself to learn how OpenIG works. The Flat-File attribute store contains only one set of credentials. OpenIG makes the assumption this user is logging into the sample application. In a real deployment OpenIG would look up the user credentials using its own session, a SAML 2.0 assertion, or a header from an OpenAM policy agent. Use cases that follow show examples of these types of deployments.

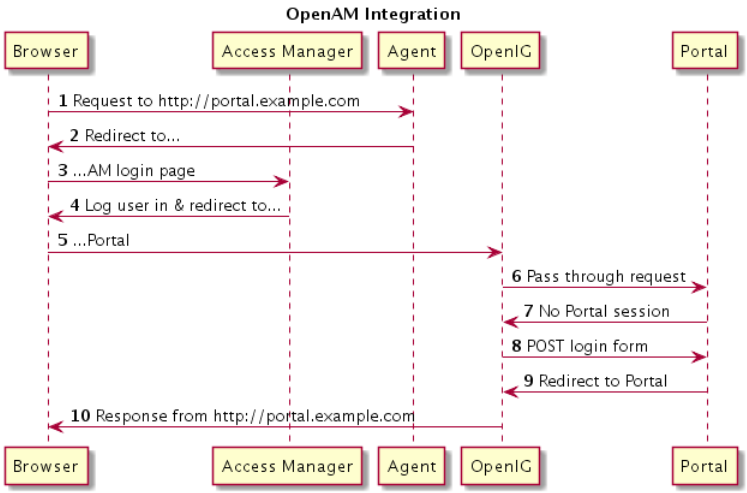


1. User accesses the Portal.
2. OpenIG intercepts request, finds no login page and passes it through.
3. Portal finds no local session and redirects to its login page for authentication.
4. OpenIG intercepts the redirect, finds a match for the login page, fetches the credentials from the flat file.
5. OpenIG creates the login form and POSTs it to the Portal login page.
6. Portal validates the credentials and redirects to the user's home page.

7. OpenIG passes the request through to the browser.

5.2. OpenAM Integration

The figure below illustrates OpenIG integrated into an OpenAM deployment. In this deployment OpenIG is running in a container that is protected by an OpenAM policy agent. The agent is configured to forward a header, with the subject (user) of the single sign-on session, to OpenIG. OpenIG then uses the subject as the login credentials, or uses the subject as a reference to look up the login credentials in a database or directory. The HR application is integrated into the SSO deployment without an agent or any modification to the application or its deployment configuration.



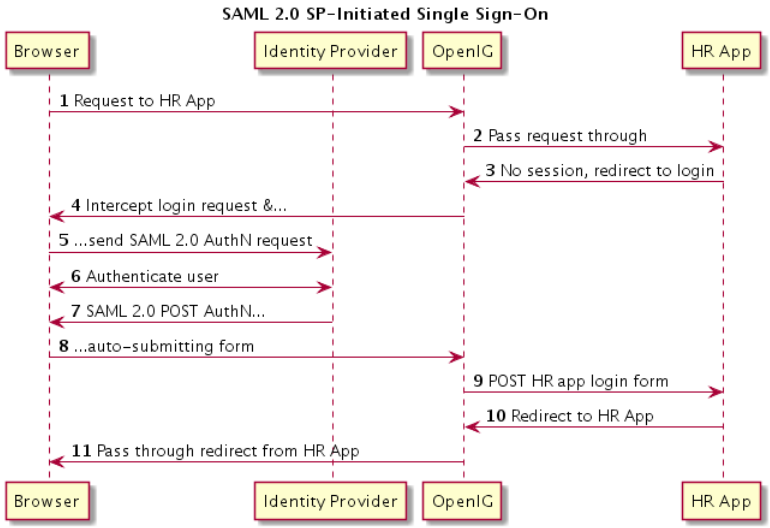
1. User browses to the Portal.
2. OpenAM policy agent intercepts the request, finds no valid OpenAM session, redirects the browser to...
3. ...an OpenAM login page.
4. OpenAM logs in the user and redirects...
5. ...back to the Portal.
6. OpenAM plugin finds a valid session, request goes through, OpenIG passes the request through to the Portal.
7. Portal finds no local session, redirects to the Portal login page.
8. OpenIG inspects the redirect, finds a match for the login page, creates the login form, and POSTs it to the Portal.

9. Portal validates the credentials and redirects to the Portal page.

10. OpenIG passes the request through to the browser.

5.3. OpenIG Federation SP Initiated SAML 2.0 SSO

The figure below illustrates OpenIG Federation providing SAML 2.0 features acting as Service Provider (SP) in an SP-initiated single sign-on configuration. In this sample, the HR application is an outsourced provider of HR services and has started seeing increased demand for SAML 2.0 support in their core application. The companies to which they outsource are refusing proprietary means of authentication and demanding the widely-accepted SAML 2.0 standard. The HR application cannot be modified to support SAML 2.0 nor do they have the time or money to integrate and deploy all of OpenAM. With OpenIG Federation, the HR application deploys OpenIG as a reverse proxy for their application, configures it as a SAML 2.0 Service Provider, and configures it to log users into the HR application upon successful verification of the SAML 2.0 authentication assertion from their customers' SAML 2.0 Identity Providers.



1. The user accesses the HR application through a bookmark in the browser.

2. OpenIG Federation inspects the request, no match is found for the HR application's login page so the request goes through.

3. HR application finds no HR session, sends a redirect to its login page.

4. OpenIG Federation intercepts the redirect, finds a match for the login page, and...

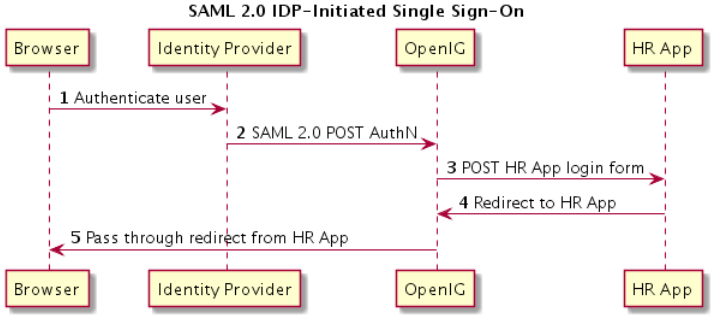
5. ...issues an SP-initiated SSO SAML 2.0 request to the organization's IDP.

6. The IDP receives the SAML 2.0 AuthN request and authenticates the user.
7. After authenticating the user the IDP sends a SAML 2.0 POST...
8. ...to OpenIG Federation using an auto-submitting form in the browser.

   OpenIG Federation validates the assertion and makes the assertion attributes available to the OpenIG login chain.
9. OpenIG login chain gets the user credentials and POSTs the login form to the application.
10. The HR application verifies the credentials and redirects to its home page.
11. OpenIG Federation passes the HR application's redirect to the browser.

5.4. OpenIG Federation IDP Initiated SAML 2.0 SSO

The figure below illustrates OpenIG Federation providing SAML 2.0 features acting as Service Provider in an IDP-initiated single sign-on configuration.



1. User clicks the HR link on the company portal and is redirected to the company Identity Provider (IDP) for authentication.
2. IDP sends an AuthN Response to the HR application.
3. OpenIG Federation receives the POST, validates the assertion, and makes the attributes available to the OpenIG login chain.
4. OpenIG login chain retrieves the user credentials and POSTs the login form to the HR application.
5. HR application validates the credentials and redirects to the main page of the application.

Chapter 6

Tutorial On Looking Up Credentials

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.

6.1. 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.

The initial OpenIG configuration file should look like the one used to login with a hard-coded username and password, see *Configuration for Hard-Coded Credentials*.

Tip

If you get stuck running the following samples, know that you can activate OpenIG debug logging with a "ConsoleLogSink" configuration object. Add it as the first object in the array of configuration objects.

```
{
    "name": "LogSink",
    "type": "ConsoleLogSink",
    "config": {
        "level": "DEBUG"
    }
}
```

6.2. 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. Edit the OpenIG configuration file, `config.json`, to use the `FileAttributesFilter` configuration object.

   To do this, make the following changes to the file.

   a. Before the "LoginRequest" object, add a new `FileAttributesFilter` configuration object.

      The `FileAttributesFilter` configuration specifies the file to access and the fields of records in the file, the key and value to look up to retrieve the user's record, and where exchange to store the search results.

      The new filter configuration object should look like this:

```
{
    "name": "CredentialsFromFile",
    "type": "FileAttributesFilter",
    "config": {
        "target": "${exchange.credentials}",
        "file": "/tmp/userfile",
        "key": "email",
        "value": "george@example.com"
    }
}
```

Do not forget to add the comma after the object, so that your configuration file remains valid JSON.

b. Edit the list of filters in the "OutgoingChain" object to include your new filter before the "LoginRequest".

The full "OutgoingChain" object should now look like this:

```
{
    "name": "OutgoingChain",
    "type": "Chain",
    "config": {
        "filters": [
            "CredentialsFromFile",
            "LoginRequest",
            "CaptureFilter"
        ],
        "handler": "DefaultHandler"
    }
}
```

c. Edit the "LoginRequest" configuration object so that it retrieves the username and password values from the exchange.

The edited configuration object should look like this:

```
{
    "name": "LoginRequest",
    "type": "StaticRequestFilter",
    "config": {
        "method": "POST",
        "uri": "http://www.example.com:8081",
        "form": {
            "username": [
                "${exchange.credentials.username}"
            ],
            "password": [
                "${exchange.credentials.password}"
            ]
        }
    }
}
```

You can find the entire configuration file in the example, *Configuration for Login With Credentials From a File*.

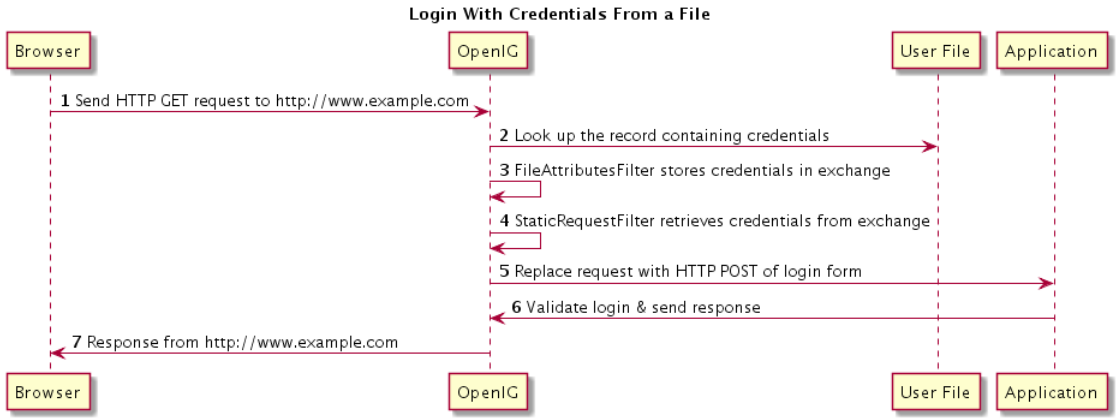
3. On Windows systems, edit the path names to the user file and the log file.
4. Verify that the configuration file is still valid JSON, and then save your work.
5. Restart the Jetty server where OpenIG is deployed.

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

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

What's happening behind the scenes?

Figure 6.1.



OpenIG intercepts your browser's HTTP GET request. 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.

6.3. 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 6.1. 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 6.2, "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 6.2. 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/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>
```

Procedure 6.3. Preparing the OpenIG Configuration

Follow these steps to prepare the OpenIG configuration to look up credentials in the database.

1. Edit the OpenIG configuration file, `config.json`, to use an `SqlAttributesFilter` configuration object.

   To do this, make the following changes to the file.

   a. Replace any content between the "OutgoingChain" and "LoginRequest" objects with a new `SqlAttributesFilter` configuration object.

      The `SqlAttributesFilter` configuration specifies the data source configured in the application server, the SQL statement and parameters to request the data, and the target where OpenIG stores the results.

      The new filter configuration object should look like this:

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

      Do not forget to add the comma after the object, so that your configuration file remains valid JSON.

b. Edit the list of filters in the "OutgoingChain" object to include your new filter before the "LoginRequest".

The full "OutgoingChain" object should now look like this:

```
{
    "name": "OutgoingChain",
    "type": "Chain",
    "config": {
        "filters": [
            "CredentialsFromSql",
            "LoginRequest",
            "CaptureFilter"
        ],
        "handler": "DefaultHandler"
    }
}
```

c. Edit the "LoginRequest" configuration object so that it retrieves the username and password values from the exchange.

The edited configuration object should look like this:

```
{
    "name": "LoginRequest",
    "type": "StaticRequestFilter",
    "config": {
        "method": "POST",
        "uri": "http://www.example.com:8081",
        "form": {
            "username": [
                "${exchange.credentials.USERNAME}"
            ],
            "password": [
                "${exchange.credentials.PASSWORD}"
            ]
        }
    }
}
```

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

2. Verify that the configuration file is still valid JSON, and then save your work.

You can find the entire configuration file in the example, *Configuration for Login With Credentials From a Database*.

Procedure 6.4. To Try Logging In With Credentials From a Database

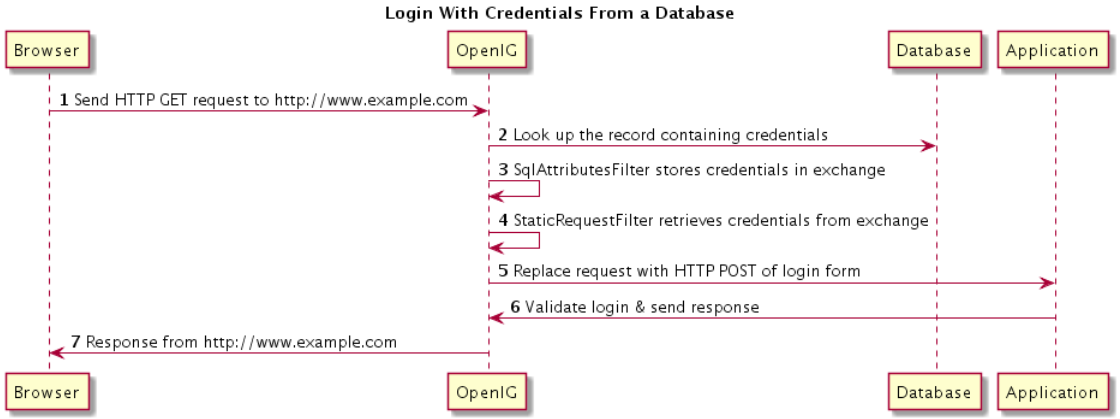
With H2, Jetty, and OpenIG correctly configured, you can try it out.

1. Restart Jetty.
2. Access the home page again, http://www.example.com:8080/.

   OpenIG logs you in automatically as George.

What's happening behind the scenes?

Figure 6.2.



OpenIG intercepts your browser's HTTP GET request. 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.

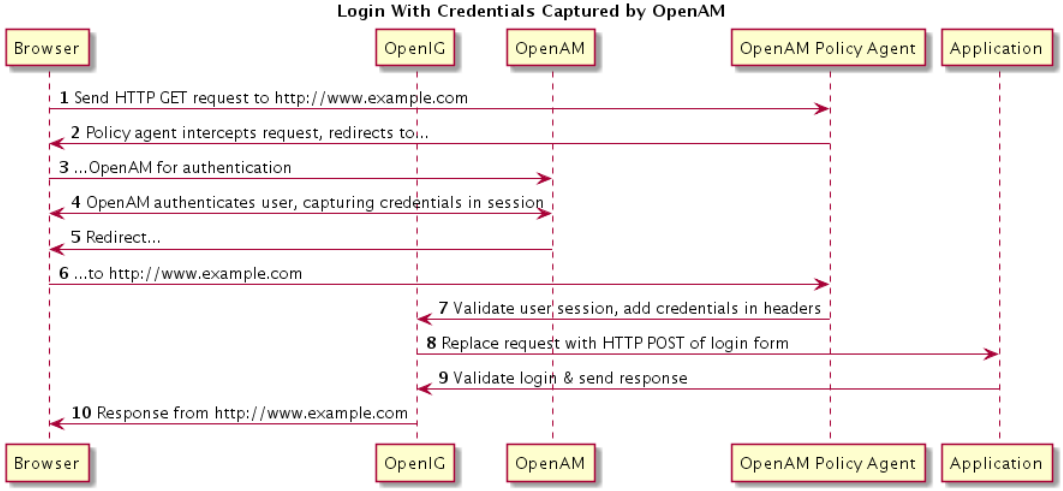
Chapter 7

Tutorial On OpenAM Password Capture & Replay

This tutorial 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.

7.1. 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.



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.

7.2. 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-3.0.0-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-3.0.0-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.

7.3. Setup Details

In this section, it is assumed that you are familiar with the components involved.

7.3.1. 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.

7.3.2. 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.

7.3.3. 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 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.

```
$ 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.

7.3.4. 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, see *Configuration for Proxy & Capture*.

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.

7.3.5. 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>/*</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/. You should be redirected to OpenAM for authentication.

Login as user `george`, password `costanza`. OpenAM redirects you back to Jetty.

7.3.6. Configuring OpenIG

Follow these steps to configure OpenIG.

- Replace the existing OpenIG configuration file with the configuration including a "CryptoHeaderFilter" object and a "StaticRequestFilter" object, see *Configuration for Password Capture & Replay*.

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

  The "StaticRequestFilter" retrieves the username and password values, and replaces the original request by an HTTP POST with login form data.

- Replace the `DESKEY` placeholder in the "CryptoHeaderFilter" configuration object with the key you generated in Section 7.3.3, "Configuring Password Capture".

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

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

- Restart Jetty so that OpenIG takes the configuration changes into account.
- Log out of OpenAM if you are logged in already.

7.4. Trying It Out

Access the application home page, http://www.example.com:8080/.

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 8

Using OpenIG 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.

See the *Tutorial For OpenIG Federation* in order to try this out with OpenAM playing the role of Identity Provider.

8.1. 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.

8.2. 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. You must configure both this file 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.

8.3. 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.

8.4. Federation Configuration Details

The following sample configuration is corresponds to a scenario where OpenIG receives a SAML assertion from OpenAM, and then logs the user in to the protected application using the username and password from the assertion. Descriptions of the fields follow the example.

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

"name" (required)

Name of the Federation component of OpenIG. Do not modify this value.

"type" (required)

Class name of the Federation handler. Do not modify this value.

"assertionMapping" (required)

The "assertionMapping" defines how to transform attributes from the incoming assertion to attribute value pairs in OpenIG. Each entry in the `assertionMapping` is of the form `localName: incomingName`, where *incomingName* is used to fetch the value from the incoming assertion, and *localName* is the name of the attribute set in the session.

The following statements correspond to the sample shown above.

If the incoming assertion contains the statement:

```
mail = george@example.com
```

```
mailPassword = costanza
```

Then the following values are set in the session:

```
username = george@example.com
```

```
password = costanza
```

For this to work, you must edit the <Attribute name="attributeMap"> element in the SP extended metadata file, `$HOME/.openig/SAML/sp-extended.xml`, so that it matches the assertion mapping configured in the IDP metadata.

"redirectURI" (required)

Set this to the page that the filter used to HTTP POST a login form recognizes as the login page for the protected application.

This is how OpenIG and the Federation component work together to provide single sign-on. When OpenIG detects the login page of the protected application, it redirects to the Federation component. Once the Federation handler validates the SAML exchanges with the IDP, and sets the required session attributes, it redirects back to the login page of the protected application. This allows the filter used to HTTP POST a login form to finish the job by creating a login form to post to the application based on the credentials retrieved from the session attributes.

"assertionConsumerEndpoint" (optional)

Default: "fedletapplication" (same as the Fedlet)

If you modify this attribute you must change the metadata to match.

"authnContext" (optional)

An authentication context to use when sending the request to the IDP, such as `PasswordProtectedTransport`.

"authnContextDelimiter" (optional)

The authentication context delimiter used when there are multiple authentication contexts in the assertion.

Default: `|`

"logoutURI" (optional)

Set this to the URI that logs the user out of the protected application.

You only need to set this if the application uses the single logout feature of the Identity Provider.

"sessionIndexMapping" (optional)

The IDP sends the `sessionIndex` for the user in the assertion. The value contained in the assertion is set as the value of the attribute `sessionIndex` in the session.

"singleLogoutEndpoint" (optional)

Default: "fedletSLORedirect" (same as the Fedlet)

If you modify this attribute you must change the metadata to match.

"singleLogoutEndpointSoap" (optional)

Default: "fedletSloSoap" (same as the Fedlet)

If you modify this attribute you must change the metadata to match.

"SPinitiatedSLOEndpoint" (optional)

Default: "SPInitiatedSLO"

If you modify this attribute you must change the metadata to match.

"SPinitiatedSSOEndpoint" (optional)

Default: "SPInitiatedSSO"

If you modify this attribute you must change the metadata to match.

"subjectMapping" (optional)

The value contained in the assertion subject is set as the value of the attribute `subjectName` in the session.

8.5. 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}"
            ]
        }
    }
}
```

8.6. 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
```

Chapter 9

Tutorial For OpenIG Federation

This tutorial demonstrates how to configure OpenIG as a SAML 2.0 federation service provider to log users in to a protected application.

When following this tutorial, you configure 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.

In this tutorial, it is assumed that you are familiar with SAML 2.0 federation and with the components involved. For an overview, read the chapter on *Using OpenIG Federation*.

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.

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, see *Configuration for Proxy & Capture*.

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.

9.2. 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`.

9.3. Configuring OpenIG For Federation

Replace the existing OpenIG configuration file with the new configuration from the example, *Configuration for the Federation Tutorial*.

Unpack the configuration files from the Fedlet you created in Section 9.2, "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.

9.4. 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 OpenIG protecting the application, at http://www.example.com:8080/, 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?

Consider the configuration, `$HOME/.openig/config/config.json`. In this configuration, the "DispatchHandler" is the entry point to OpenIG processing. The "DispatchHandler" provides an internal routing mechanism, based on the state of the exchange. If the incoming URI matches a SAML URI, then the "SamlFederationHandler" can process an incoming SAML 2.0 assertion, and also set the attributes from the assertion in the session. This condition is not met until the user has authenticated.

If no user name is set in the session of the exchange, the "DispatchHandler" considers that the user has not yet authenticated. This condition is met when the user has not yet authenticated. In this case the "DispatchHandler" dispatches to the static request handler, "SPInitiatedSSORedirectHandler", which redirects the user-agent to the Identity Provider for SAML 2.0 single sign-on. After authentication, the Identity Provider redirects the user-agent back to a SAML URI on the Service Provider (OpenIG), and so the request is dispatched to the "SamlFederationHandler" mentioned above.

Once the attributes from the assertion are set in the session, then the exchange is dispatched to the "LoginChain". The "LoginRequest" static request filter in the "LoginChain" 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.

Chapter 10

Configuring 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.

10.1. 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.

10.2. 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.

The initial OpenIG configuration file should look like the one used to login with a hard-coded username and password, see *Configuration for Hard-Coded Credentials*.

Tip

If you get stuck running the following samples, know that you can activate OpenIG debug logging with a "ConsoleLogSink" configuration object. Add it as the first object in the array of configuration objects.

```
{
    "name": "LogSink",
    "type": "ConsoleLogSink",
    "config": {
        "level": "DEBUG"
    }
}
```

10.3. 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, edit the OpenAM demo user to set credentials in the Email Address and Employee Number fields of the demo user profile. In OpenAM console, browse to the user profile under Access Control > / (Top Level Realm) > Subjects > User > demo. Set Email Address to `george` and Employee Number to `costanza`.

10.4. 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 `exchange.oauth2AccessToken`.

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, replace the `config.json` file with the new configuration from the example, *Configuration for an OAuth 2.0 Resource Server*.

Notice "ResourceServer" configuration. This configuration 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 `"enforceHttps": false` to allow testing without having to set up keys and certificates. (In production environments, do use HTTPS to protect access tokens.)

When the "ResourceServer" filter has injected information for a valid access token into the exchange, the "CaptureTokenInfo" filter dumps the token information to the log. The "CaptureTokenInfo" filter also injects the credentials from the user profile in OpenAM into the exchange.

Finally, the "LoginRequestFilter" logs a user in to the minimal HTTP server by using the credentials injected from the token information. In this simple configuration, all requests result in login attempts against the minimal HTTP server.

After updating `config.json`, restart Jetty server so that OpenIG loads the new configuration.

10.5. 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=demo&password=changeit&scope=mail%20employeenumber" \
 http://openam.example.com:8088/openam/oauth2/access_token
{
    "scope": "mail employeenumber",
    "expires_in": 600,
    "token_type": "Bearer",
    "refresh_token": "7fe9b284-698e-4738-ad11-01b984920861",
    "access_token": "b8102957-486b-47f2-bd2b-54faa55ec363"
}

$ curl \
 --header "Authorization: Bearer b8102957-486b-47f2-bd2b-54faa55ec363" \
 http://www.example.com:8080
...
    <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 raw token information. The raw token information looks something like the following.

```
2014-08-04T16:43:38Z:CaptureTokenInfo.CaptureTokenInfo:INFO:
{
    "access_token": "b8102957-486b-47f2-bd2b-54faa55ec363",
    "mail": "george",
    "employeenumber": "costanza",
    "grant_type": "password",
    "scope": [
        "mail",
        "employeenumber"
    ],
    "realm": "/",
    "token_type": "Bearer",
    "expires_in": 35
}
```

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

Chapter 11

Configuring 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.

11.1. About OpenIG as an OAuth 2.0 Client

As described in the chapter on *Configuring 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.

11.2. 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.

11.3. 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.

Tip

If you get stuck running the following samples, know that you can activate OpenIG debug logging with a "ConsoleLogSink" configuration object. Add it as the first object in the array of configuration objects.

```
{
    "name": "LogSink",
    "type": "ConsoleLogSink",
    "config": {
        "level": "DEBUG"
    }
}
```

11.4. 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 these 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.

Finally, edit the OpenAM demo user to set credentials in the Last Name and Full Name fields of the demo user profile. In OpenAM console, browse to the user profile under Access Control > / (Top Level Realm) > Subjects > User > demo. Set Last Name to `george` and Full Name to `costanza`.

11.5. Configuring OpenIG as a Relying Party

To configure OpenIG as an OpenID Connect 1.0 relying party, replace the `config.json` file with the new configuration from the example, *Configuration for an OpenID Connect 1.0 Client*.

Also add the `DumpExchange.groovy` script from that section under `$HOME/.openig/scripts/groovy` (`%appdata%\OpenIG\scripts\groovy` on Windows). The script is called from the configuration on failure.

In the new `config.json` file, consider the "OpenIDConnectClient" filter. This configuration object 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 11.4, "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 "OpenIDConnectClient" filter 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 "OpenIDConnectClient" filter 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 "OpenIDConnectClient" filter, 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 "OpenIDConnectClient" filter, the typical "ClientHandler" configures the HTTP client that communicates with the OpenID Provider.

Returning to the top of the configuration, notice that the "OpenIDConnectChain" invokes an "OutgoingChain" handler after the filter injects the access token and user information into `exchange.openid`. The "OutgoingChain" in this case extracts credentials from the user information by script ("GetCredentials"), and then uses the credentials to log the user in to the minimal HTTP server ("LoginRequestFilter").

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

After updating `config.json`, restart Jetty server so that OpenIG loads the new configuration.

11.6. Trying It Out

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

When redirected to the OpenAM login page, login as user `demo`, password `changeit`, 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 "OpenIDConnectClient" filter 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. Before handling the next filter or handler in the exchange, which is the "OutgoingChain", the filter injects the authorization state information into `exchange.openid`.

The "OutgoingChain" extracts credentials to re-inject them into the exchange. Its login filter then uses the credentials to log the user in to the minimal HTTP server, which responds with its User Information page.

Chapter 12
Routing Tutorial

Other tutorials in this guide demonstrate use of a single configuration file for all of OpenIG. In those tutorials, you had to restart OpenIG to pick up configuration changes.

This tutorial instead demonstrates how you can use a Router in the *Reference* and Route in the *Reference* configurations to make changes at runtime. This tutorial also shows how to lock down the configurations for deployment so that accidental changes to configuration files do not affect servers running in production.

12.1. 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.

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, see *Configuration for Proxy & Capture*.

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.

12.2. Configuring Routes

Routes are configuration objects to handle a particular kind of Exchange.

The particular kind of an Exchange that a Route handles is an Exchange that fits the condition defined for the route. The condition is defined using a OpenIG expression in the *Reference*, so it can be based on almost any characteristic of the Exchange. 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.

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 one of the server-wide configuration files you have used in other tutorials.

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. This object explicitly shows you all the fields of the Route object. (You could omit "condition" and "baseURI" here as they have no effect.)

```
{
    "heap": {
        "objects": [
            {
                "name": "ClientHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "name": "default",
    "condition": null,
    "baseURI": null,
    "handler": "ClientHandler"
}
```

The rest of this section indicates how to set up Route configurations. Two of the Route configurations direct requests to ForgeRock.com and ForgeRock.org based on a parameter in form data. The third Route configuration directs request to the minimal HTTP server when the parameter is not set.

1. Create a file system directory where you store the Route configurations.

   By default, Route configurations are stored in `$HOME/.openig/config/routes` (`%appdata%\OpenIG\config\routes` on Windows). Create that file system directory now.

2. Add a ForgeRock.com Route file in the directory, `forgerock.json`, that holds the following content.

   ```
   {
       "heap": {
           "objects": [
               {
                   "name": "ForgeRockChain",
                   "type": "Chain",
                   "config": {
                       "filters": [],
                       "handler": "DefaultHandler"
                   }
               },
               {
                   "name": "DefaultHandler",
                   "type": "ClientHandler",
                   "config": {}
               }
           ]
   ```

```
    },
    "name": "ForgeRock",
    "handler": "ForgeRockChain",
    "condition": "${exchange.request.form.site[0] == 'forgerock'}",
    "baseURI": "http://forgerock.com:80/"
}
```

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

3. Add a ForgeRock.org community Route file in the directory, `community.json`, that holds the following content.

```
{
    "heap": {
        "objects": [
            {
                "name": "CommunityChain",
                "type": "Chain",
                "config": {
                    "filters": [],
                    "handler": "DefaultHandler"
                }
            },
            {
                "name": "DefaultHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "name": "Community",
    "handler": "CommunityChain",
    "condition": "${exchange.request.form.site[0] == 'community'}",
    "baseURI": "http://forgerock.org:80/"
}
```

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

4. Add a default Route file in the directory, `default.json`, that holds the following content.

```
{
    "heap": {
        "objects": [
            {
                "name": "LoginChain",
                "type": "Chain",
                "config": {
```

```
                    "filters": [
                        "LoginRequest"
                    ],
                    "handler": "DefaultHandler"
                }
            },
            {
                "name": "LoginRequest",
                "type": "StaticRequestFilter",
                "config": {
                    "method": "POST",
                    "uri": "http://www.example.com:8081",
                    "form": {
                        "username": [
                            "george"
                        ],
                        "password": [
                            "costanza"
                        ]
                    }
                }
            },
            {
                "name": "DefaultHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handler": "LoginChain",
    "name": "zDefault"
}
```

This Route has no condition set, and so it accepts any Exchange. When this Route picks up an Exchange, it uses a static request filter to login George Costanza with hard-coded username and password.

12.3. Configuring the Router

At this point you have configured the Routes, but OpenIG does not route any traffic to them. To use the routes, you must configure a Router.

The Router is a handler that you can configure in the top-level `config.json` file for OpenIG.[1] The Router's job is to pass Exchanges to configured Routes, 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 could configure the Router first and then add Routes while OpenIG is running.

Configure the Router as follows.

[1] In fact you can add a Router wherever you can add a Handler, not only in the top-level configuration.

1. Stop Jetty.
2. Replace the existing `config.json` file content with a simpler configuration that ends in a Router.

```
{
    "heap": {
        "objects": [
            {
                "name": "DispatchHandler",
                "type": "DispatchHandler",
                "config": {
                    "bindings": [
                        {
                            "handler": "Router",
                            "baseURI": "http://www.example.com:8081"
                        }
                    ]
                }
            },
            {
                "name": "Router",
                "type": "Router",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

   This configuration 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.

3. Restart Jetty.

12.4. Trying it Out

At this point you can try your new Router and Route configurations.

Make a request to hit the ForgeRock.com router.

```
$ curl --data "site=forgerock" http://www.example.com:8080
...HTML of ForgeRock.com home page...
```

Now make a request to hit the Community page.

```
$ curl --data "site=community" http://www.example.com:8080
...HTML of ForgeRock.org home page...
```

Now check that the default route still works.

```
$ curl http://www.example.com:8080 | grep george

   <title>Howdy, george</title>
        <dd>george</dd>
```

What happened behind the scenes?

When you issued your first request with HTTP POST form data "site=forgerock", 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 HTTP POST form data "site=community", the request matched the condition defined in the Community Route. OpenIG rebased the request and sent it along to `http://forgerock.org:80/`.

When the third request did not match either of the conditions defined, the Exchange was routed to the default Route (that accepts any Exchange). The static request filter in that route logged George in to the local server listening on `http://www.example.com:8081/`. The default Route has name "zDefault".

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

12.5. 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 13

Configuration Templates

This chapter contains templates of common configurations. Start with one of our templates and then modify to suit your deployment. Read the summary of each template to find the right match for your application. If you are not sure about the characteristics of your application, start with the basic Application Capture template. This template allows you to setup basic proxying and capture the traffic of the login sequence in a flat file, which then allows you to analyze the application and subsequently choose the right template or add your own configuration.

Note

- All templates have the CaptureFilter in the *Reference* enabled by default. Remove the capture filter from the outgoing chain before running the gateway in production. Capturing is typically used only for initial development or debugging and may rapidly fill up your available disk space if left enabled.
- Substitute the `TARGETIP` tag with the IP address of your application.
- Modify the `LoginRequest` filter to match the form required for login by your target application.

13.1. Proxy & Capture

Proxies all requests and captures them in a flat file. Use this template if you need to analyze the traffic for your application. Simply change the `baseURI` to be that of the target application, restart OpenIG, and login to the application. The entire sequence is logged to the flat file.

```
{
"heap": {
    "objects": [
        {
            "name": "DispatchHandler",
            "type": "DispatchHandler",
            "config": {
                "bindings": [
                    {
                        "condition": "${exchange.request.uri.scheme == 'http'}",
                        "handler": "OutgoingChain",
                        "baseURI": "http://TARGETIP"
                    },
                    {
                        "condition": "${exchange.request.uri.path == '/login'}",
                        "handler": "LoginChain",
                        "baseURI": "https://TARGETIP"
                    },
```

```
                        {
                            "handler": "OutgoingChain",
                            "baseURI": "https://TARGETIP"
                        }
                    ]
                }
            },
            {
                "name": "LoginChain",
                "type": "Chain",
                "config": {
                    "filters": [],
                    "handler": "OutgoingChain"
                }
            },
            {
                "name": "OutgoingChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "CaptureFilter"
                    ],
                    "handler": "ClientHandler"
                }
            },
            {
                "name": "CaptureFilter",
                "type": "CaptureFilter",
                "config": {
                    "captureEntity": false,
                    "file": "/tmp/gateway.log"
                }
            },
            {
                "name": "ClientHandler",
                "comment": "Sends all requests to remote servers.",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

13.2. 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.

```
    {
    "heap": {
        "objects": [
            {
```

```
            "name": "DispatchHandler",
            "type": "DispatchHandler",
            "config": {
                "bindings": [
                    {
                        "condition": "${exchange.request.uri.path == '/login'}",
                        "handler": "LoginChain",
                        "baseURI": "http://TARGETIP"
                    },
                    {
                        "handler": "OutgoingChain",
                        "baseURI": "http://TARGETIP"
                    }
                ]
            }
        },
        {
            "name": "LoginChain",
            "type": "Chain",
            "config": {
                "filters": [
                    "LoginRequest"
                ],
                "handler": "OutgoingChain"
            }
        },
        {
            "name": "LoginRequest",
            "type": "StaticRequestFilter",
            "config": {
                "method": "POST",
                "uri": "https://TARGETIP/login",
                "form": {
                    "USER": [
                        "myusername"
                    ],
                    "PASSWORD": [
                        "mypassword"
                    ]
                }
            }
        },
        {
            "name": "OutgoingChain",
            "type": "Chain",
            "config": {
                "filters": [
                    "CaptureFilter"
                ],
                "handler": "ClientHandler"
            }
        },
        {
            "name": "CaptureFilter",
            "type": "CaptureFilter",
            "config": {
                "captureEntity": false,
                "file": "/tmp/gateway.log"
            }
```

```
            },
            {
                "name": "ClientHandler",
                "comment": "Responsible for sending all requests to remote servers.",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

13.3. 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": {
        "objects": [
            {
                "name": "DispatchHandler",
                "type": "DispatchHandler",
                "config": {
                    "bindings": [
                        {
                            "condition": "${exchange.request.uri.path == '/eum/login'}",
                            "handler": "LoginChain",
                            "baseURI": "http://TARGETIP"
                        },
                        {
                            "handler": "OutgoingChain",
                            "baseURI": "http://TARGETIP"
                        }
                    ]
                }
            },
            {
                "name": "LoginChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "SwitchFilter"
                    ],
                    "handler": "OutgoingChain"
                }
            },
            {
                "name": "SwitchFilter",
                "type": "SwitchFilter",
                "config": {
```

```
                "onResponse": [
                    {
                        "handler": "LoginRequestHandler"
                    }
                ]
            }
        },
        {
            "name": "LoginRequestHandler",
            "type": "Chain",
            "config": {
                "filters": [
                    "LoginRequest"
                ],
                "handler": "OutgoingChain"
            }
        },
        {
            "name": "LoginRequest",
            "type": "StaticRequestFilter",
            "config": {
                "method": "POST",
                "uri": "https://TARGETIP/login",
                "form": {
                    "USER": [
                        "myusername"
                    ],
                    "PASSWORD": [
                        "mypassword"
                    ]
                },
                "headers": {
                    "cookie": [
                        "${exchange.response.headers['Set-Cookie'][0]}"
                    ]
                }
            }
        },
        {
            "name": "OutgoingChain",
            "type": "Chain",
            "config": {
                "filters": [
                    "CaptureFilter"
                ],
                "handler": "ClientHandler"
            }
        },
        {
            "name": "CaptureFilter",
            "type": "CaptureFilter",
            "config": {
                "captureEntity": false,
                "file": "/tmp/gateway.log"
            }
        },
        {
            "name": "ClientHandler",
            "comment": "Responsible for sending all requests to remote servers.",
```

```
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

13.4. 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. If you change the `TARGETIP:PORT` to be the IP address of OpenAM, the `TARGETDN:PORT` to be the fully qualified name and port of OpenAM and modify `USERNAME` and `PASSWORD` in the `LoginRequest` you automatically log `USERNAME` into 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": {
    "objects": [
        {
            "name": "DispatchHandler",
            "type": "DispatchHandler",
            "config": {
                "bindings": [
                    {
                        "handler": "FindLoginPageChain",
                        "baseURI": "http://TARGETIP:PORT"
                    }
                ]
            }
        },
        {
            "name": "FindLoginPageChain",
            "type": "Chain",
            "config": {
                "filters": [
                    "IsLoginPage",
                    "FindLoginPage"
                ],
                "handler": "OutgoingChain"
            }
```

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

```
                        ]
                    },
                    "headers": {
                        "host": [
                            "TARGETFQDN:PORT"
                        ]
                    }
                }
            },
            {
                "name": "OutgoingChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "CookieFilter",
                        "CaptureFilter"
                    ],
                    "handler": "ClientHandler"
                }
            },
            {
                "name": "CookieFilter",
                "type": "CookieFilter",
                "config": {}
            },
            {
                "name": "CaptureFilter",
                "type": "CaptureFilter",
                "config": {
                    "captureEntity": true,
                    "file": "/tmp/gateway.log"
                }
            },
            {
                "name": "LogSink",
                "comment": "Default sink for logging information.",
                "type": "ConsoleLogSink",
                "config": {
                    "level": "DEBUG"
                }
            },
            {
                "name": "ClientHandler",
                "comment": "Responsible for sending all requests to remote servers.",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

13.5. 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.

```
{
"heap": {
    "objects": [
        {
            "name": "DispatchHandler",
            "type": "DispatchHandler",
            "config": {
                "bindings": [
                    {
                        "condition": "${exchange.request.uri.path == '/login'}",
                        "handler": "LoginChain",
                        "baseURI": "http://TARGETIP"
                    },
                    {
                        "handler": "OutgoingChain",
                        "baseURI": "http://TARGETIP"
                    }
                ]
            }
        },
        {
            "name": "LoginChain",
            "type": "Chain",
            "config": {
                "filters": [
                    "HiddenValueExtract",
                    "LoginRequest"
                ],
                "handler": "OutgoingChain"
            }
        },
        {
            "name": "HiddenValueExtract",
            "type": "EntityExtractFilter",
            "config": {
                "messageType": "response",
                "target": "${exchange.hiddenValue}",
                "bindings": [
                    {
                        "key": "value",
                        "pattern": "wpLoginToken\"\s.*value=\"(.*)\"",
                        "template": "$1"
                    }
                ]
            }
        },
        {
            "name": "LoginRequest",
            "type": "StaticRequestFilter",
```

```
                "config": {
                    "method": "POST",
                    "uri": "https://TARGETIP/login",
                    "form": {
                        "USER": [
                            "myusername"
                        ],
                        "PASSWORD": [
                            "mypassword"
                        ],
                        "hiddenValue": [
                            "${exchange.hiddenValue.value}"
                        ]
                    }
                }
            },
            {
                "name": "OutgoingChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "CaptureFilter"
                    ],
                    "handler": "ClientHandler"
                }
            },
            {
                "name": "CaptureFilter",
                "type": "CaptureFilter",
                "config": {
                    "captureEntity": false,
                    "file": "/tmp/gateway.log"
                }
            },
            {
                "name": "ClientHandler",
                "comment": "Responsible for sending all requests to remote servers.",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

13.6. 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 the login will all take place on port 443, you will need to add the login filters and handlers to the `LoginChain`. To get started quickly, modify the `baseURI` to be the `IPAddress` of your target application. This should allow you

to proxy all traffic to the application. Then add the logic for the `LoginChain` using the flow from one of the login templates.

```
{
"heap": {
    "objects": [
        {
            "name": "DispatchHandler",
            "type": "DispatchHandler",
            "config": {
                "bindings": [
                    {
                        "condition": "${exchange.request.uri.scheme == 'http'}",
                        "handler": "OutgoingChain",
                        "baseURI": "http://TARGETIP"
                    },
                    {
                        "condition": "${exchange.request.uri.path == '/login'}",
                        "handler": "LoginChain",
                        "baseURI": "https://TARGETIP"
                    },
                    {
                        "handler": "OutgoingChain",
                        "baseURI": "https://TARGETIP"
                    }
                ]
            }
        },
        {
            "name": "LoginChain",
            "type": "Chain",
            "config": {
                "filters": [],
                "handler": "OutgoingChain"
            }
        },
        {
            "name": "OutgoingChain",
            "type": "Chain",
            "config": {
                "filters": [
                    "CaptureFilter"
                ],
                "handler": "ClientHandler"
            }
        },
        {
            "name": "CaptureFilter",
            "type": "CaptureFilter",
            "config": {
                "captureEntity": false,
                "file": "/tmp/gateway.log"
            }
        },
        {
            "name": "ClientHandler",
            "comment": "Responsible for sending all requests to remote servers.",
```

```
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

13.7. 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.

```
    {
    "heap": {
        "objects": [
            {
                "name": "DispatchHandler",
                "type": "DispatchHandler",
                "config": {
                    "bindings": [
                        {
                            "condition": "${exchange.request.uri.path == '/login'}",
                            "handler": "LoginChain",
                            "baseURI": "http://TARGETIP"
                        },
                        {
                            "handler": "OutgoingChain",
                            "baseURI": "http://TARGETIP"
                        }
                    ]
                }
            },
            {
                "name": "LoginChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "LoginRequest"
                    ],
                    "handler": "OutgoingChain"
                }
            },
            {
                "name": "LoginRequest",
                "type": "StaticRequestFilter",
                "config": {
                    "method": "POST",
                    "uri": "https://TARGETIP/login",
                    "form": {
```

```
                        "USER": [
                            "${exchange.request.headers['username'][0]}"
                        ],
                        "PASSWORD": [
                            "${exchange.request.headers['password'][0]}"
                        ]
                    }
                }
            },
            {
                "name": "OutgoingChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "CaptureFilter"
                    ],
                    "handler": "ClientHandler"
                }
            },
            {
                "name": "CaptureFilter",
                "type": "CaptureFilter",
                "config": {
                    "captureEntity": false,
                    "file": "/tmp/gateway.log"
                }
            },
            {
                "name": "ClientHandler",
                "comment": "Responsible for sending all requests to remote servers.",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

13.8. Microsoft Online Outlook Web Access

A sample template used to log a user into Microsoft Online Outlook Web Access. This template shows how you would use OpenIG and the OpenAM password capture feature to integrate with OWA. You can follow the Tutorial On Password Capture & Replay tutorial and substitute this template.

```
    {
    "heap": {
        "objects": [
            {
                "name": "LogSink",
                "comment": "Default sink for logging information.",
                "type": "ConsoleLogSink",
                "config": {
```

```
                "level": "DEBUG"
            }
        },
        {
            "name": "DispatchHandler",
            "type": "DispatchHandler",
            "config": {
                "bindings": [
                    {
                        "condition": "${exchange.request.uri.path == '/owa/auth/logon.aspx'}",
                        "handler": "LoginChain",
                        "baseURI": "https://65.55.171.158"
                    },
                    {
                        "handler": "OutgoingChain",
                        "baseURI": "https://65.55.171.158"
                    }
                ]
            }
        },
        {
            "name": "LoginChain",
            "type": "Chain",
            "config": {
                "filters": [
                    "CryptoHeaderFilter",
                    "LoginRequest"
                ],
                "handler": "OutgoingChain"
            }
        },
        {
            "name": "CryptoHeaderFilter",
            "type": "CryptoHeaderFilter",
            "config": {
                "messageType": "REQUEST",
                "operation": "DECRYPT",
                "algorithm": "DES/ECB/NoPadding",
                "key": "DESKEY",
                "keyType": "DES",
                "charSet": "utf-8",
                "headers": [
                    "password"
                ]
            }
        },
        {
            "name": "LoginRequest",
            "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"
                    ]
                }
            }
        },
        {
            "name": "OutgoingChain",
            "type": "Chain",
            "config": {
                "filters": [
                    "HeaderFilter",
                    "CaptureFilter"
                ],
                "handler": "ClientHandler"
            }
        },
        {
            "name": "HeaderFilter",
            "type": "HeaderFilter",
            "config": {
                "messageType": "REQUEST",
                "remove": [
                    "password",
                    "username"
                ]
            }
        },
        {
            "name": "CaptureFilter",
            "type": "CaptureFilter",
            "config": {
                "captureEntity": false,
                "file": "/tmp/gateway.log"
            }
        },
        {
            "name": "ClientHandler",
            "type": "ClientHandler",
            "config": {}
        }
    ]
```

```
    },
    "handlerObject": "DispatchHandler"
}
```

Chapter 14

Scripting Filters & Handlers

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.

Interface Stability: Evolving in the *Reference*

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 following example defines a `ScriptableFilter`, written in the Groovy language, and stored in a file named `SimpleFormLogin.groovy`.

```
{
    "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.

14.1. 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
import org.forgerock.openig.io.ByteArrayBranchingStream

/*
 * 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

}
```

14.2. 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 hard-coded username and password.

```
/*
 * Perform basic authentication with a hard-coded user name and password.
 */

def credentials = "bjensen:hifalutin".getBytes().encodeBase64().toString()
exchange.request.headers.add("Authorization", "Basic ${credentials}" 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)
```

14.3. 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.session.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.session.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.

14.4. 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 the gateway.

```
/*
 * 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
    }
}
```

Chapter 15

Customizing OpenIG

With ScriptableFilter in the *Reference* and also ScriptableHandler in the *Reference* objects, you can often avoid customizing OpenIG Java code by writing scripts instead. For examples, see Scripting Filters & Handlers.

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*

15.1. Key Extension Points

The two primary extension points are the interfaces: Filter (for processing a request and/or response en route) and Handler (for generating responses from requests). These interfaces are similar to the Java Enterprise Edition `Filter` and `Servlet` interfaces, with some differences in the semantics of messages. While you can simply implement these interfaces, there are also included convenience classes: GenericFilter and GenericHandler that you can use if you intend to make your extensions configurable through the OpenIG configuration resource.

15.2. 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.

15.3. 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.

15.4. 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 NestedHeaplet class in a nested class in the class you want to create and initialize and implementing its create method.

Within the `create` method, you can access the object's configuration through the config field.

15.5. Sample Filter

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

```
package com.example.filter;

// Java Standard Edition
import java.io.IOException;

// OpenIG Core Library
import org.forgerock.openig.filter.GenericFilter;
import org.forgerock.openig.handler.HandlerException;
import org.forgerock.openig.heap.HeapException;
import org.forgerock.openig.heap.NestedHeaplet;
import org.forgerock.openig.http.Exchange;

public class HelloWorldFilter extends GenericFilter {
```

```
    public String name;
    public String value;

    @Override
    public void filter(Exchange exchange, Handler next)
            throws HandlerException, IOException {
        exchange.request.getHeaders().putSingle(name, value); // set header in request
        next.handle(exchange); // pass to remaining filters & handler in chain
        exchange.response.getHeaders().putSingle(name, value); // set header in response
    }

    public static class Heaplet extends NestedHeaplet {

        @Override
        public Object create() throws HeapException {
            HelloWorldFilter filter = new HelloWorldFilter();
            filter.name = config.get("name").required().asString(); // required
            filter.value = config.get("value").required().asString(); // req'd
            return filter;
        }
    }
}
```

The corresponding heap object configuration is as follows.

```
{
     "name": "HelloWorldFilter",
     "type": "com.example.filter.HelloWorldFilter",
     "config": {
         "name": "X-Hello",
         "value": "World"
     }
}
```

Chapter 16
Troubleshooting

This chapter covers common problems and their solutions.

16.1. Object not found in heap

```
org.forgerock.json.fluent.JsonValueException: /handlerObject:
   object Router2 not found in heap
    at org.forgerock.openig.heap.HeapUtil.getRequiredObject(HeapUtil.java:69)
    at org.forgerock.openig.servlet.GatewayServlet.init(GatewayServlet.java:188)
    at org.eclipse.jetty.servlet.ServletHolder.initServlet(ServletHolder.java:595)
```

You have specified "handlerObject": "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.

16.2. Unexpected character (x) at position 1103

```
HTTP ERROR 500
Problem accessing /. Reason:

Unexpected character (x) at position 1103
```

This error usually means a missing double quote or a missing bracket in the configuration file. Use a JSON editor or JSON validation tool such as JSONLint to make sure your JSON is valid.

16.3. 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.

16.4. 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
at org.forgerock.openig.filter.LoggingFilter.writeHeaders(LoggingFilter.java:132
at org.forgerock.openig.filter.LoggingFilter.logResponse(LoggingFilter.java:119)
at org.forgerock.openig.filter.LoggingFilter.filter(LoggingFilter.java:86)
at org.forgerock.openig.filter.Chain.handle(Chain.java:54)
```

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.

16.5. StaticResponseHandler results in a blank page

You must define an entity for the response. For example:

```
{
    "name": "AccessDeniedHandler",
    "type": "org.forgerock.openig.handler.StaticResponseHandler",
    "config": {
        "status": 403,
        "reason": "Forbidden",
        "entity": "<html><h2>User does not have permission</h2></html>"
    }
}
```

16.6. 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*.

16.7. 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.

16.8. 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`.

Appendix A. Tutorial Configuration Files

This appendix holds the following configuration files for tutorials in this guide.

Example A.1. Configuration for Proxy & Capture

```
{
    "heap": {
        "objects": [
            {
                "name": "DispatchHandler",
                "type": "DispatchHandler",
                "config": {
                    "bindings": [
                        {
                            "handler": "OutgoingChain",
```

```
                            "baseURI": "http://www.example.com:8081"
                        }
                    ]
                }
            },
            {
                "name": "OutgoingChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "CaptureFilter"
                    ],
                    "handler": "DefaultHandler"
                }
            },
            {
                "name": "CaptureFilter",
                "type": "CaptureFilter",
                "config": {
                    "captureEntity": false,
                    "file": "/tmp/gateway.log"
                }
            },
            {
                "name": "DefaultHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

Example A.2. Configuration for Hard-Coded Credentials

```
{
    "heap": {
        "objects": [
            {
                "name": "DispatchHandler",
                "type": "DispatchHandler",
                "config": {
                    "bindings": [
                        {
                            "handler": "OutgoingChain",
                            "baseURI": "http://www.example.com:8081"
                        }
                    ]
                }
            },
            {
                "name": "OutgoingChain",
                "type": "Chain",
                "config": {
```

```
                    "filters": [
                        "LoginRequest",
                        "CaptureFilter"
                    ],
                    "handler": "DefaultHandler"
                }
            },
            {
                "name": "LoginRequest",
                "type": "StaticRequestFilter",
                "config": {
                    "method": "POST",
                    "uri": "http://www.example.com:8081",
                    "form": {
                        "username": [
                            "demo"
                        ],
                        "password": [
                            "changeit"
                        ]
                    }
                }
            },
            {
                "name": "CaptureFilter",
                "type": "CaptureFilter",
                "config": {
                    "captureEntity": false,
                    "file": "/tmp/gateway.log"
                }
            },
            {
                "name": "DefaultHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

Example A.3. Configuration for Login With Credentials From a File

```
{
    "heap": {
        "objects": [
            {
                "name": "DispatchHandler",
                "type": "DispatchHandler",
                "config": {
                    "bindings": [
                        {
                            "handler": "OutgoingChain",
                            "baseURI": "http://www.example.com:8081"
```

```
                        }
                    ]
                }
            },
            {
                "name": "OutgoingChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "CredentialsFromFile",
                        "LoginRequest",
                        "CaptureFilter"
                    ],
                    "handler": "DefaultHandler"
                }
            },
            {
                "name": "CredentialsFromFile",
                "type": "FileAttributesFilter",
                "config": {
                    "target": "${exchange.credentials}",
                    "file": "/tmp/userfile",
                    "key": "email",
                    "value": "george@example.com"
                }
            },
            {
                "name": "LoginRequest",
                "type": "StaticRequestFilter",
                "config": {
                    "method": "POST",
                    "uri": "http://www.example.com:8081",
                    "form": {
                        "username": [
                            "${exchange.credentials.username}"
                        ],
                        "password": [
                            "${exchange.credentials.password}"
                        ]
                    }
                }
            },
            {
                "name": "CaptureFilter",
                "type": "CaptureFilter",
                "config": {
                    "captureEntity": false,
                    "file": "/tmp/gateway.log"
                }
            },
            {
                "name": "DefaultHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

Example A.4. Configuration for Login With Credentials From a Database

```
{
    "heap": {
        "objects": [
            {
                "name": "DispatchHandler",
                "type": "DispatchHandler",
                "config": {
                    "bindings": [
                        {
                            "handler": "OutgoingChain",
                            "baseURI": "http://www.example.com:8081"
                        }
                    ]
                }
            },
            {
                "name": "OutgoingChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "CredentialsFromSql",
                        "LoginRequest",
                        "CaptureFilter"
                    ],
                    "handler": "DefaultHandler"
                }
            },
            {
                "name": "CredentialsFromSql",
                "type": "SqlAttributesFilter",
                "config": {
                    "target": "${exchange.credentials}",
                    "dataSource": "java:comp/env/jdbc/forgerock",
                    "preparedStatement": "SELECT username, password FROM users WHERE email = ?;",
                    "parameters": [
                        "george@example.com"
                    ]
                }
            },
            {
                "name": "LoginRequest",
                "type": "StaticRequestFilter",
                "config": {
                    "method": "POST",
                    "uri": "http://www.example.com:8081",
                    "form": {
                        "username": [
                            "${exchange.credentials.USERNAME}"
                        ],
                        "password": [
                            "${exchange.credentials.PASSWORD}"
```

```
                        ]
                    }
                }
            },
            {
                "name": "CaptureFilter",
                "type": "CaptureFilter",
                "config": {
                    "captureEntity": false,
                    "file": "/tmp/gateway.log"
                }
            },
            {
                "name": "DefaultHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

Example A.5. Configuration for Password Capture & Replay

```
{
    "heap": {
        "objects": [
            {
                "name": "DispatchHandler",
                "type": "DispatchHandler",
                "config": {
                    "bindings": [
                        {
                            "handler": "OutgoingChain",
                            "baseURI": "http://www.example.com:8081"
                        }
                    ]
                }
            },
            {
                "name": "OutgoingChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "CryptoHeaderFilter",
                        "LoginRequest",
                        "HeaderFilter",
                        "CaptureFilter"
                    ],
                    "handler": "DefaultHandler"
                }
            },
            {
                "name": "CryptoHeaderFilter",
```

```
                "type": "CryptoHeaderFilter",
                "config": {
                    "messageType": "REQUEST",
                    "operation": "DECRYPT",
                    "algorithm": "DES/ECB/NoPadding",
                    "key": "DESKEY",
                    "keyType": "DES",
                    "charSet": "utf-8",
                    "headers": [
                        "password"
                    ]
                }
            },
            {
                "name": "LoginRequest",
                "type": "StaticRequestFilter",
                "config": {
                    "method": "POST",
                    "uri": "http://www.example.com:8081",
                    "form": {
                        "username": [
                            "${exchange.request.headers['username'][0]}"
                        ],
                        "password": [
                            "${exchange.request.headers['password'][0]}"
                        ]
                    }
                }
            },
            {
                "name": "HeaderFilter",
                "type": "HeaderFilter",
                "config": {
                    "messageType": "REQUEST",
                    "remove": [
                        "password",
                        "username"
                    ]
                }
            },
            {
                "name": "CaptureFilter",
                "type": "CaptureFilter",
                "config": {
                    "captureEntity": true,
                    "file": "/tmp/gateway.log"
                }
            },
            {
                "name": "DefaultHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

Example A.6. Configuration for the Federation Tutorial

```
{
    "heap": {
        "objects": [
            {
                "name": "LogSink",
                "comment": "Default sink for logging information.",
                "type": "ConsoleLogSink",
                "config": {
                    "level": "DEBUG"
                }
            },
            {
                "name": "DispatchHandler",
                "type": "DispatchHandler",
                "config": {
                    "bindings": [
                        {
                            "condition": "${matches(exchange.request.uri.path, '^/saml')}",
                            "handler": "SamlFederationHandler"
                        },
                        {
                            "condition": "${empty exchange.session.username}",
                            "handler": "SPInitiatedSSORedirectHandler",
                            "baseURI": "http://www.example.com:8081"
                        },
                        {
                            "handler": "LoginChain",
                            "baseURI": "http://www.example.com:8081"
                        }
                    ]
                }
            },
            {
                "name": "SamlFederationHandler",
                "type": "org.forgerock.openig.handler.saml.SamlFederationHandler",
                "config": {
                    "assertionMapping": {
                        "username": "mail",
                        "password": "employeenumber"
                    },
                    "subjectMapping": "subjectName",
                    "redirectURI": "/"
                }
            },
            {
                "name": "SPInitiatedSSORedirectHandler",
                "type": "StaticResponseHandler",
                "config": {
                    "status": 302,
                    "reason": "Found",
                    "headers": {
                        "Location": [
                            "http://www.example.com:8080/saml/SPInitiatedSSO"
                        ]
```

```
                    }
                }
            },
            {
                "name": "LoginChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "LoginRequest"
                    ],
                    "handler": "ClientHandler"
                }
            },
            {
                "name": "LoginRequest",
                "type": "StaticRequestFilter",
                "config": {
                    "method": "POST",
                    "uri": "http://www.example.com:8081",
                    "form": {
                        "username": [
                            "${exchange.session.username}"
                        ],
                        "password": [
                            "${exchange.session.password}"
                        ]
                    }
                }
            },
            {
                "name": "ClientHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

Example A.7. Configuration for an OAuth 2.0 Resource Server

```
{
    "heap": {
        "objects": [
            {
                "name": "ResourceServerChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "CaptureFilter",
                        "ResourceServer",
                        "CaptureTokenInfo",
                        "LoginRequestFilter"
                    ],
                    "handler": "ClientHandler"
```

```
                }
            },
            {
                "name": "CaptureFilter",
                "type": "CaptureFilter",
                "config": {
                    "captureEntity": false,
                    "file": "/tmp/gateway.log"
                }
            },
            {
                "name": "ResourceServer",
                "type": "OAuth2ResourceServerFilter",
                "config": {
                    "httpHandler": "ClientHandler",
                    "requiredScopes": [
                        "mail",
                        "employeenumber"
                    ],
                    "tokenInfoEndpoint": "http://openam.example.com:8088/openam/oauth2/tokeninfo",
                    "enforceHttps": false
                }
            },
            {
                "name": "CaptureTokenInfo",
                "type": "ScriptableFilter",
                "config": {
                    "type": "application/x-groovy",
                    "source"
                          : "logger.info(exchange.oauth2AccessToken.rawInfo.toString());
                            exchange.username = exchange.oauth2AccessToken.rawInfo.get('mail').asString();
                            exchange.password =
exchange.oauth2AccessToken.rawInfo.get('employeenumber').asString();
                            next.handle(exchange)"
                }
            },
            {
                "name": "LoginRequestFilter",
                "type": "StaticRequestFilter",
                "config": {
                    "method": "POST",
                    "uri": "http://www.example.com:8081",
                    "form": {
                        "username": [
                            "${exchange.username}"
                        ],
                        "password": [
                            "${exchange.password}"
                        ]
                    }
                }
            },
            {
                "name": "ClientHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
```

```
    "handlerObject": "ResourceServerChain",
    "baseURI": "http://www.example.com:8081"
}
```

Example A.8. Configuration for an OpenID Connect 1.0 Client

The following listing shows `config.json`

```
{
    "heap": {
        "objects": [
            {
                "name": "OpenIDConnectChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "CaptureFilter",
                        "OpenIDConnectClient"
                    ],
                    "handler": "OutgoingChain"
                }
            },
            {
                "name": "OpenIDConnectClient",
                "type": "OAuth2ClientFilter",
                "config": {
                    "clientEndpoint": "/openid",
                    "requireHttps": false,
                    "requireLogin": true,
                    "target": "${exchange.openid}",
                    "scopes": [
                        "openid",
                        "profile"
                    ],
                    "failureHandler": "Dump",
                    "providerHandler": "ClientHandler",
                    "providers": [
                        {
                            "name": "openam",
                            "wellKnownConfiguration":
                              "http://openam.example.com:8088/openam/.well-known/openid-configuration",
                            "clientId": "OpenIG",
                            "clientSecret": "password"
                        }
                    ]
                }
            },
            {
                "name": "Dump",
                "type": "Chain",
                "config": {
                    "filters": [
                        "CaptureFilter"
                    ],
                    "handler": "DumpExchange"
```

```
            }
        },
        {
            "name": "DumpExchange",
            "type": "ScriptableHandler",
            "config": {
                "type": "application/x-groovy",
                "file": "DumpExchange.groovy"
            }
        },
        {
            "name": "OutgoingChain",
            "type": "Chain",
            "config": {
                "filters": [
                    "GetCredentials",
                    "LoginRequestFilter",
                    "CaptureFilter"
                ],
                "handler": "ClientHandler"
            }
        },
        {
            "name": "GetCredentials",
            "type": "ScriptableFilter",
            "config": {
                "type": "application/x-groovy",
                "source": "exchange.username = exchange.openid.user_info.family_name;
                           exchange.password = exchange.openid.user_info.name;
                           next.handle(exchange)"
            }
        },
        {
            "name": "LoginRequestFilter",
            "type": "StaticRequestFilter",
            "config": {
                "method": "POST",
                "uri": "http://www.example.com:8081",
                "form": {
                    "username": [
                        "${exchange.username}"
                    ],
                    "password": [
                        "${exchange.password}"
                    ]
                }
            }
        },
        {
            "name": "CaptureFilter",
            "type": "CaptureFilter",
            "config": {
                "captureEntity": true,
                "file": "/tmp/gateway.log"
            }
        },
        {
            "name": "ClientHandler",
            "type": "ClientHandler",
```

```
                "config": {}
            }
        ]
    },
    "handlerObject": "OpenIDConnectChain"
}
```

The following listing shows `DumpExchange.groovy`

```
import org.forgerock.openig.http.Response
import groovy.json.JsonOutput

map = new LinkedHashMap(exchange)
map.remove("exchange")
map.remove("javax.servlet.http.HttpServletRequest")
map.remove("javax.servlet.http.HttpServletResponse")

json = JsonOutput.prettyPrint(JsonOutput.toJson(map))

exchange.response = new Response()
exchange.response.status = 200
exchange.response.entity = "<html><pre>" + json + "</pre></html>"
```

Example A.9. Configuration for the Routing Tutorial

The following listing shows `config.json`

```
{
    "heap": {
        "objects": [
            {
                "name": "DispatchHandler",
                "type": "DispatchHandler",
                "config": {
                    "bindings": [
                        {
                            "handler": "Router",
                            "baseURI": "http://www.example.com:8081"
                        }
                    ]
                }
            },
            {
                "name": "Router",
                "type": "Router",
                "config": {}
            }
        ]
    },
    "handlerObject": "DispatchHandler"
}
```

The following listing shows `community.json`

```
{
    "heap": {
        "objects": [
            {
                "name": "CommunityChain",
                "type": "Chain",
                "config": {
                    "filters": [],
                    "handler": "DefaultHandler"
                }
            },
            {
                "name": "DefaultHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "name": "Community",
    "handler": "CommunityChain",
    "condition": "${exchange.request.form.site[0] == 'community'}",
    "baseURI": "http://forgerock.org:80/"
}
```

The following listing shows `default.json`

```
{
    "heap": {
        "objects": [
            {
                "name": "LoginChain",
                "type": "Chain",
                "config": {
                    "filters": [
                        "LoginRequest"
                    ],
                    "handler": "DefaultHandler"
                }
            },
            {
                "name": "LoginRequest",
                "type": "StaticRequestFilter",
                "config": {
                    "method": "POST",
                    "uri": "http://www.example.com:8081",
                    "form": {
                        "username": [
                            "george"
                        ],
                        "password": [
```

```
                            "costanza"
                        ]
                    }
                }
            },
            {
                "name": "DefaultHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "handler": "LoginChain",
    "name": "zDefault"
}
```

The following listing shows `forgerock.json`

```
{
    "heap": {
        "objects": [
            {
                "name": "ForgeRockChain",
                "type": "Chain",
                "config": {
                    "filters": [],
                    "handler": "DefaultHandler"
                }
            },
            {
                "name": "DefaultHandler",
                "type": "ClientHandler",
                "config": {}
            }
        ]
    },
    "name": "ForgeRock",
    "handler": "ForgeRockChain",
    "condition": "${exchange.request.form.site[0] == 'forgerock'}",
    "baseURI": "http://forgerock.com:80/"
}
```

Index