Installation Guide

ON THIS PAGE

Installation Guide

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

This guide describes options for installing IG for customized or secure environments. For information about how to install and configure IG for evaluation, see the Getting Started.

Upgrade

Supported Upgrade Paths

The following table lists supported upgrade paths to IG 7.1:

Version	Upgrade supported?
IG 6.x	✔
IG 7.x	✔

For more information, see Checking your product versions are supported in the *ForgeRock Knowledge Base*.

For unsupported, legacy deployments, ForgeRock can assist you in the upgrade process.

Planning the Upgrade

Major, minor, maintenance, and patch product release levels are defined in ForgeRock Product Release Levels. How much you need to do to upgrade IG software depends on the magnitude of the differences between the version you currently use and the new version.

Minor, maintenance, and patch releases have a limited effect on current functionality but contain necessary bug and security fixes. Keep up-to-date with maintenance and patch releases because the fixes are important, and the risk of affecting service is minimal.

Do these planning tasks **before** you start an upgrade:

Planning task	Description
Find out what changed	Read the Release Notes for **all** releases between the version you currently use and the new version.
Understand the impact	Decide whether you need to change the configuration of your deployment for this release, and evaluate the work involved. Make sure you meet all of the requirements for the new version. In particular, make sure that you have a recent, supported Java version.
Plan for server downtime	At least one of your IG servers will be down during upgrade. Plan to route client applications to another server until the upgrade process is complete, and you have validated the result. Make sure client application owners are aware of the change, and let them know what to expect. If you have a single IG server, make sure the downtime happens in a low-usage window, and make sure you let client application owners plan accordingly.

Planning task	Description
Back up	The IG configuration is a set of files, including `admin.json`, `config.json`, `logback.xml`, routes, and scripts. Back up the IG configuration and store it in version control, so that you can roll back if something goes wrong. Also back up any tools scripts that you have edited for your deployment, and any trust stores used to connect securely.
Plan for rollback	Sometimes even a well-planned upgrade fails to go smoothly. In such cases, you need a plan to roll back smoothly to the pre-upgrade version. For IG servers, roll back by restoring a backed-up configuration.
Prepare a test environment	Before applying the upgrade in your production environment, always try to upgrade IG in a test environment. This will help you gauge the amount of work required, without affecting your production environment, and will help smooth out unforeseen problems. The test environment should resemble your production environment as closely as possible.

Upgrade the IG Configuration

Use the Release Notes for **all** releases between the version you currently use and the new version, and upgrade your configuration as follows:

- Review all Incompatible Changes, and adjust your configuration as necessary.
- Switch to the replacement settings in Deprecation. Although deprecated objects continue to work, they add to the notifications in the logs, and are eventually removed.
- Check the lists of Fixes, Limitations, and Known Issues, to see if they impact your deployment.
- Recompile your Java extensions. The method signature or imports for supported and evolving APIs can change in each version.
- Read the Documentation Updates for new examples and information that can help with your configuration.

Upgrade IG Instances

For information about the versions that are supported for upgrade, see Supported Upgrade Paths.

Upgrade a Single IG Instance

1. Read and act on Plan the Upgrade and Upgrade the IG Configuration.
2. Back up the IG configuration, and store it in version control so that you can roll back if something goes wrong.
3. Stop IG.
4. Make the new configuration available on the file system, and specify the `IG_INSTANCE_DIR` env variable or `ig.instance.dir` system property to point to them.
5. Restart IG.
6. In a test environment that simulates your production environment, validate that the upgraded service performs as expected with the new configuration. Check the logs for new or unexpected notifications or errors.
7. Allow client application traffic to flow to the upgraded site.

Upgrade a Site With Multiple IG Instances

The most straightforward option when upgrading sites with multiple IG instances is to upgrade in place. One by one, stop, upgrade, and then restart each server individually, leaving the service running during upgrade.

Migrate From Web Container Mode to Standalone Mode

IG is delivered as a standalone Java executable in a .zip file, as well as in a .war file. Consider these points to migrate from IG in web container mode to IG in standalone mode.

Session Replication Between IG Instances

High-availability of sessions is not supported in standalone mode.

Streaming Asynchronous Responses and Events

In ClientHandler and ReverseProxyHandler, use only the default mode of `asyncBehavior:non_streaming`; responses are processed when the entity content is entirely available.

If the property is set to `streaming`, the setting is ignored.

Connection Reuse When Client Certificates Are Used for Authentication

In ClientHandler and ReverseProxyHandler, use only the default mode of `stateTrackingEnabled:true`; when a client certificate is used for authentication, connections cannot be reused.

If the property is set to `false`, the setting is ignored.

Tomcat Configuration

Feature	Standalone	Tomcat
Port number	Configure the `connectors` property of admin.json.	Configure in the `Connector` element of `/path/to/tomcat/conf/server.xml`: `<Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000" redirectPort="8443" />`
HTTPS server-side configuration	Create a keystore, set up secrets, and configure secrets stores, ports, and ServerTlsOptions in admin.json. For information, see Configure IG For HTTPS (Server-Side) in Standalone Mode.	Create a keystore, and set up the SSL port in the `Connector` element of `/path/to/tomcat/conf/server.xml`. For information, see Configure IG for HTTPS (Server-Side) in Tomcat.
Session cookie name	Configure the `session` property of admin.json.	Configure `WEB-INF/web.xml` when you unpack the IG .war file.
Access logs	Configure in the Audit framework. For information, see Auditing Your Deployment and Audit Framework.	Configure with `AccessLogValve`.

Feature	Standalone	Tomcat
JDBC datasource	Configure with the JdbcDataSource object. For information, see JdbcDataSource. For an example, see Log In With Credentials From a Database.	Configure in the `GlobalNamingResources` element of `/path/to/tomcat/conf/server.xml`.
Environment variables	Configure in `$HOME/.openig/bin/env.sh`, where `$HOME/.openig` is the instance directory.	Configure in `/path/to/tomcat/bin/setenv.sh`.
Jar files	Add to `$HOME/.openig/extra`, where `$HOME/.openig` is the instance directory.	Add to to web container classpath; for example `/path/to/tomcat/webapps/ROOT/WEB-INF/lib`.

Prepare the Network

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

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

Restart the browser after making this change.

Install IG in Standalone Mode

Download and Start IG in Standalone Mode

Consider the following best practices for installing and running IG:

- Create a service account with which to install and run IG, for example, `igadmin`. Do not use the root account to install and run IG.
- Allocate the following permissions to the account:
  - Read/write permissions on the installation directory, for example `/path/to/identity-gateway`.

- Execute permissions on the scripts in the installation `bin` directory, for example `/path/to/identity-gateway/bin`.

The following sections describe how to install and start IG in standalone mode, from a .zip file.

Download the IG .zip File

1. Create a local installation directory for IG. The examples in this section use `/path/to`.
2. Download `IG-7.1.2.zip` from the ForgeRock BackStage download site, and copy the .zip file to the installation directory:

```
$ cp IG-7.1.2.zip /path/to/IG-7.1.2.zip
```

3. Unzip the file:

```
$ unzip IG-7.1.2.zip
```

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

Start IG With Default Settings

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

1. Start IG:

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

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

By default, the base location for IG configuration files is in `$HOME/.openig`.

To read the configuration from a different location, specify the base location as an argument. The following example reads the configuration from the `config` directory under `/path/to/instance-dir`:

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

```
/path/to/instance-dir
```

2. Check that IG is running in one of the following ways:
   - Ping IG at `http://openig.example.com:8080/openig/ping`, and make sure an `HTTP 200` is returned.
   - Access the IG welcome page at `http://openig.example.com:8080`.
   - When IG is running in development mode, display the product version and build information at `http://openig.example.com:8080/openig/api/info`.

Start IG With Custom Settings

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

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

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

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

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

For more information, see AdminHttpApplication (admin.json).

Stop IG

1. In the terminal where IG is running, select CTRL+C to stop the service.

Configure IG For HTTPS (Server-Side) in Standalone Mode

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

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

Serve the Same Certificate for TLS Connections to All Server Names

During a TLS handshake, IG accesses secret key and certificate pairs sychronously; they are loaded in memory at IG startup, and **must** be present. You must restart IG to update a secret key and certificate pair. For information about secret stores provided in IG, see Secrets.

This example uses a PKCS12 keystore, but you can adapt it to use other certificates.

Before you start, install IG in standalone mode, as described in Download and Start IG in Standalone Mode.

1. Locate a directory for secrets, for example, `/path/to/secrets`, and go to it.
2. Create a keystore holding a self-signed certificate:

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

NOTE

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

3. Add a file called `keystore.pass`, containing the keystore password `password`:

```
$ echo -n 'password' > keystore.pass
```

NOTE

NOTE

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

4. Set up TLS on IG:

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

      1. Linux
      2. Windows

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

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

```
{
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443,
      "tls": "ServerTlsOptions-1"
    }
  ],
  "properties": {
    "ig_keystore_directory": "/path/to/secrets"
  },
  "heap": [
    {
      "name": "ServerTlsOptions-1",
      "type": "ServerTlsOptions",
      "config": {
        "keyManager": {
          "type": "SecretsKeyManager",
          "config": {
            "signingSecretId": "key.manager.secret.id",
            "secretsProvider": "ServerIdentityStore"
          }
        }
      }
    },
    {
      "type": "FileSystemSecretStore",
      "name": "SecretsPasswords",
```

```
        "config": {
          "directory": "&{ig_keystore_directory}/",
          "format": "PLAIN"
        }
      },
      {
        "type": "KeyStoreSecretStore",
        "name": "ServerIdentityStore",
        "config": {
          "file": "&{ig_keystore_directory}/IG-keystore",
          "storePassword": "keystore.pass",
          "secretsProvider": "SecretsPasswords",
          "mappings": [
            {
              "secretId": "key.manager.secret.id",
              "aliases": ["https-connector-key"]
            }
          ]
        }
      }
    ]
}
```

Notice the following features of the file:

- IG starts on port `8080`, and on `8443` over TLS.
- IG's private keys for TLS are managed by the SecretsKeyManager, whose ServerIdentityStore references a KeyStoreSecretStore.
- The KeyStoreSecretStore maps the keystore alias to the secret ID for retrieving the server keys (private key + certificate).
- The password of the KeyStoreSecretStore is provided by the FileSystemSecretStore.

5. Start IG:

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

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

6. Access the IG welcome page on https://openig.example.com:8443.

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

Configure Environment Variables and System Properties for IG in Standalone Mode

Configure environment variables and system properties for IG in standalone mode, as follows:

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

Start IG With a Customized Router Scan Interval

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

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

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

Specify Environment Variables for Key and JVM Options

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

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

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

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

Add .jar Files for IG Extensions in Standalone Mode

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

API

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

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

Install IG in Apache Tomcat

IMPORTANT

If you use startup scripts to bootstrap the IG web container, the scripts can start the container process with a different user. To prevent errors, make sure that the location of the IG configuration is correct. Alternatively, adapt the startup scripts to specify the `IG_INSTANCE_DIR` env variable or `ig.instance.dir` system properties, taking care to set file permissions correctly.

If you start and stop the IG web container yourself, the default location of the IG configuration files is correct. By default, IG configuration files are located under `$HOME/.openig` on Linux, Mac, and UNIX systems, and under `%appdata%\OpenIG` on Windows.

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

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

Downloading and Starting IG in Tomcat

The commands in this guide assume that you install Tomcat to `/path/to/tomcat`, and after installation, you have a directory `/path/to/tomcat/webapps` in which you install IG. If you use another directory structure, substitute the commands.

1. Download a supported version of Tomcat server from its download page, and install it to `/path/to/tomcat`.
2. Remove the `ROOT` directory in Tomcat:

```
$ rm -rf /path/to/tomcat/webapps/ROOT
```

3. Download `IG-7.1.2.war` from the ForgeRock BackStage download site.

4. Copy the `IG-7.1.2.war` to the Tomcat `webapps` directory, as `ROOT.war`:

```
$ cp IG-7.1.2.war /path/to/tomcat/webapps/ROOT.war
```

   Tomcat automatically deploys IG in the root context on startup.

5. Start Tomcat:

```
$ /path/to/tomcat/bin/startup.sh
```

   If necessary, make the startup scripts executable.

6. Check that IG is running in one of the following ways:
   - Ping IG at http://openig.example.com:8080/openig/ping, and make sure an `HTTP 200` is returned.
   - Access the IG welcome page at http://openig.example.com:8080.
   - When IG is running in development mode, display the product version and build information at http://openig.example.com:8080/openig/api/info.

Configure Cookie Domains in Tomcat

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

- For stateful sessions, add a context element to `/path/to/conf/Catalina/server/root.xml`, as in the following example, and then restart Tomcat to read the configuration changes:

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

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

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

Configure IG for HTTPS (Server-Side) in Tomcat

This section describes how to set up IG to run as a server over HTTPS. For information about the set up for HTTPS (client-side), see Configure IG For HTTPS (Client-Side).

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

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true">
  <SSLHostConfig sslProtocol="TLS" protocols="all"
certificateVerification="none">
    <Certificate
certificateKeystoreFile="/path/to/tomcat/conf/keystore"
                 certificateKeystorePassword="password"
                 certificateKeystoreType="PKCS12" />
  </SSLHostConfig>
</Connector>
```

Also create a keystore holding a self-signed certificate:

```
$ keytool \
-genkey \
-alias tomcat \
-keyalg RSA \
-keystore /path/to/tomcat/conf/keystore \
-storetype PKCS12 \
-storepass password \
-keypass password \
-dname "CN=openig.example.com,O=Example Corp,C=FR"
```

NOTE

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

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

Restart Tomcat to read the configuration changes.

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

Configure Access to MySQL Over JNDI in Tomcat

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

The following steps are for MySQL Connector/J:

1. Download the MySQL JDBC Driver Connector/J from http://dev.mysql.com/downloads/connector/j.
2. Copy the driver .jar to `/path/to/tomcat/lib/` so that it is on Tomcat's class path.
3. Add a JNDI data source for your MySQL server and database in `/path/to/tomcat/conf/context.xml`:

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

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

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

5. Restart Tomcat to read the configuration changes.

About Session Stickiness and Session Replication for Tomcat

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

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

  In the HowTo, you configure the `jvmRoute` attribute in the Tomcat server configuration, `/path/to/tomcat/conf/server.xml`, to identify the server. The connector can use this identifier to achieve session stickiness.
- A Tomcat cluster configuration can handle session replication. When setting up a cluster configuration, the ClusterManager defines the session replication implementation.

Install IG in Jetty

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

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

NOTE

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

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

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

Downloading and Starting IG in Jetty

The commands in this guide assume that you install Jetty to `/path/to/jetty`, and after installation, you have a directory `/path/to/jetty/webapps` in which you install IG. If you use another directory structure, substitute the commands.

1. Download a supported version of Jetty server from its download page, and install it to `/path/to/jetty`.
2. Download `IG-7.1.2.war` from the ForgeRock BackStage download site.
3. Copy the .war file:

   ```
   $ cp IG-7.1.2.war /path/to/jetty/webapps/IG-7.1.2.war
   ```

   Jetty automatically deploys IG in the root context on startup.
4. Start Jetty:
   - To start Jetty in the background, enter:

     ```
     $ /path/to/jetty/bin/jetty.sh start
     ```
   - To start Jetty in the foreground, enter:

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

5. Check that IG is running in one of the following ways:
   - Ping IG at http://openig.example.com:8080/openig/ping, and make sure an `HTTP 200` is returned.
   - Access the IG welcome page at http://openig.example.com:8080.
   - When IG is running in development mode, display the product version and build information at http://openig.example.com:8080/openig/api/info.

Configure Cookie Domains in Jetty

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

- For stateful sessions, add a session domain handler element that specifies the domain to `/path/to/jetty/etc/webdefault.xml`, as in the following example:

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

  Restart Jetty to read the configuration changes.

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

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

Configure IG for HTTPS (Server-Side) in Jetty

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

These instructions are for Jetty 9.4.21, and are not compatible with earlier versions of Jetty. For more information about Jetty and HTTPS, see http://www.eclipse.org/jetty/documentation/current/configuring-ssl.html#configuring-sslcontextfactory.

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

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

2. Set up the location for configurations and customizations to the Jetty distribution:
    - Create a directory `/path/to/jetty_base`.
    - Set the environment variable JETTY_BASE for `/path/to/jetty_base`:

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

3. Set up the keystore:
    - Remove the built-in keystore:

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

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

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

    NOTE

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

4. Create a directory to store local server customization and configurations in $JETTY_BASE:
    - Delete the global `start.ini`:

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

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

```
$ cd $JETTY_BASE
$ java -jar $JETTY_HOME/start.jar --create-startd

MKDIR : ${jetty.base}/start.d
INFO  : Base directory was modified
```

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

```
$ cd $JETTY_BASE
$ java  -jar $JETTY_HOME/start.jar \
--add-to-
start=server,webapp,deploy,ssl,jstl,ext,jsp,resources,cons
ole-capture,http,https

INFO  : webapp          initialized in
${jetty.base}/start.d/webapp.ini
INFO  : ext             initialized in
${jetty.base}/start.d/ext.ini
INFO  : server          initialized in
${jetty.base}/start.d/server.ini
INFO  : mail            transitively enabled
INFO  : servlet         transitively enabled
INFO  : jsp             initialized in
${jetty.base}/start.d/jsp.ini
INFO  : annotations     transitively enabled
INFO  : resources       initialized in
${jetty.base}/start.d/resources.ini
INFO  : transactions    transitively enabled
INFO  : threadpool      transitively enabled, ini template
available with --add-to-start=threadpool
INFO  : ssl             initialized in
${jetty.base}/start.d/ssl.ini
INFO  : plus            transitively enabled
INFO  : deploy          initialized in
${jetty.base}/start.d/deploy.ini
INFO  : jstl            initialized in
${jetty.base}/start.d/jstl.ini
INFO  : security        transitively enabled
INFO  : apache-jsp      transitively enabled
INFO  : jndi            transitively enabled
INFO  : console-capture initialized in
${jetty.base}/start.d/console-capture.ini
```

```
INFO  : apache-jstl     transitively enabled
INFO  : http            initialized in
${jetty.base}/start.d/http.ini
INFO  : client          transitively enabled
INFO  : https           initialized in
${jetty.base}/start.d/https.ini
INFO  : bytebufferpool  transitively enabled, ini template
available with --add-to-start=bytebufferpool
MKDIR : ${jetty.base}/lib
MKDIR : ${jetty.base}/lib/ext
MKDIR : ${jetty.base}/resources
MKDIR : ${jetty.base}/etc
COPY  : ${jetty.home}/modules/ssl/keystore to
${jetty.base}/etc/keystore
MKDIR : ${jetty.base}/webapps
MKDIR : ${jetty.base}/logs
INFO  : Base directory was modified
```

NOTE

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

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

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

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

```
$ cd $JETTY_HOME/lib
$ ls jetty-util-*.jar
```

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

password
OBF:1v2...v1v
MD5:5f4...f99
```

7. In `$JETTY_BASE/start.d/ssl.ini`, uncomment the following lines, and update the passwords with the OBF password returned in the previous step:

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

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

## Keystore password
jetty.sslContext.keyStorePassword=OBF:1v2j1uum1xtv1zej1zer
1xtn1uvk1v1v

## KeyManager password
jetty.sslContext.keyManagerPassword=OBF:1v2j1uum1xtv1zej1z
er1xtn1uvk1v1v
```

8. Copy the IG .war file to `$JETTY_BASE/webapps/IG-7.1.2.war`.
9. Go to $JETTY_BASE, and start Jetty:

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

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

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

Configure Access MySQL Over JNDI in Jetty

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

The following steps are for MySQL Connector/J:

1. Download the MySQL JDBC Driver Connector/J from http://dev.mysql.com/downloads/connector/j.
2. Copy the driver .jar to `/path/to/jetty/lib/jndi/` so that it is on Jetty's class path.
3. Add a JNDI data source for your MySQL server and database in `/path/to/jetty/etc/jetty.xml`:

```
<New id="jdbc/forgerock"
class="org.eclipse.jetty.plus.jndi.Resource">
  <Arg></Arg>
  <Arg>jdbc/forgerock</Arg>
```

```
  <Arg>
    <New
class="com.mysql.jdbc.jdbc2.optional.MysqlConnectionPoolDataSo
urce">
      <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.

About Session Stickiness and Session Replication for Jetty

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

- Jetty's persistent session mechanism appends a node ID to the session ID in the same way Tomcat appends the `jvmRoute` value to the session cookie. This can be useful for session stickiness if your load balancer examines the session ID.
- Session Clustering with a Database describes how to configure Jetty to persist sessions over JDBC, allowing session replication.

  Unless it is set up to be highly available, the database can be a single point of failure in this case.
- Session Clustering with MongoDB describes how to configure Jetty to persist sessions in MongoDB, allowing session replication.

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

Install IG in JBoss EAP

Download and Start IG in JBoss EAP

This section installs JBoss to `/path/to/jboss`. If you use another directory structure, substitute the commands.

1. Download a supported version of JBoss server from its download page, and install it to `/path/to/jboss`.
2. In the JBoss configuration file `/path/to/jboss/standalone/configuration/standalone.xml`, delete the line for the JBoss welcome-content handler:

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

3. Download `IG-7.1.2.war` from the ForgeRock BackStage download site.
4. Copy the `IG-7.1.2.war` to the JBoss deployment directory:

```
$ cp IG-7.1.2.war
/path/to/jboss/standalone/deployments/IG-7.1.2.war
```

5. Start JBoss as a standalone server:

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

   JBoss deploys IG in the root context.

6. Check that IG is running in one of the following ways:
   - Ping IG at http://openig.example.com:8080/openig/ping, and make sure an `HTTP 200` is returned.
   - Access the IG welcome page at http://openig.example.com:8080.
   - When IG is running in development mode, display the product version and build information at http://openig.example.com:8080/openig/api/info.

Configure Cookie Domains in JBoss EAP

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

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

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

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

Configure IG for HTTPS (Server-Side) in JBoss EAP

This section describes how to set up JBoss to run IG over HTTPS. These instructions are for JBoss EAP 7.3, and are not compatible with earlier versions. For information about the set up for HTTPS (client-side), see Configure IG For HTTPS (Client-Side).

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

Before you start, install IG in JBoss as described in Download and Start IG in JBoss EAP. JBoss is installed in `/path/to/jboss` .

1. Set the environment variable JBOSS_HOME in two terminals:

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

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

   ```
   $ $JBOSS_HOME/bin/add-user.sh myadmin myadmin-password

   Added user 'myadmin' to file
   '$JBOSS_HOME/standalone/configuration/mgmt-
   users.properties'
   Added user 'myadmin' to file
   '$JBOSS_HOME/domain/configuration/mgmt-users.properties'
   ```

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

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

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

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

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

```
$ keytool \
-genkey \
-alias jboss \
-storetype PKCS12 \
-keyalg RSA \
-keystore $JBOSS_HOME/tmp/keystore \
-storepass password \
-keypass password \
-dname "CN=openig.example.com,O=Example Corp,C=FR"
```

NOTE

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

6. Start JBoss as a standalone server:

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

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

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

8. Make sure IG is deployed on port `8080`:

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

9. Enable SSL:
   - Enable the SSL server:

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

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

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

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

Change the Default Location of the Configuration Folders

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

1. Linux
2. Windows

```
$HOME/.openig
```

```
%appdata%\OpenIG
```

Change the default base location in the following ways:

- Set the `IG_INSTANCE_DIR` environment variable to the full path to the base location:
  1. Linux
  2. Windows

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

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

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

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

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

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

Configure IG For HTTPS (Client-Side)

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

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

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

- KeyStore, to hold the server certificates or the CA's signing certificate. See KeyStore.

- SecretsTrustManager, to let IG handle the certificates in the KeyStore when deciding whether to trust a server certificate. See SecretsTrustManager.
- (Optional) KeyManager, to let IG present its certificate from the keystore when the server must authenticate IG as client. See KeyManager.
- ClientHandler and ReverseProxyHandler reference to ClientTlsOptions, for connecting to TLS-protected endpoints. See ClientTlsOptions.

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

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

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

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

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

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

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

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

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

Certificate was added to keystore
```

NOTE

NOTE

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

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

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

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

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

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

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

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

1. Linux
2. Windows

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

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

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

6. Add the following route to IG:

    1. Linux
    2. Windows

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

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

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

```
      "config": {
        "verificationSecretId": "trust.manager.secret.id",
        "secretsProvider":"KeyStoreSecretStore-1"
      }
    },
    {
      "name": "ReverseProxyHandler-1",
      "type": "ReverseProxyHandler",
      "config": {
        "tls": {
          "type": "ClientTlsOptions",
          "config": {
            "trustManager": "SecretsTrustManager-1"
          }
        },
        "hostnameVerifier": "ALLOW_ALL"
      },
      "capture": "all"
    }
  ],
  "handler": "ReverseProxyHandler-1"
}
```

Notice the following features of the route:

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

7. Test the setup:

   - Start the sample application

     ```
     $ java -jar sampleapp_install_dir/IG-sample-
     application-7.1.2.jar
     ```

     a. Go to http://openig.example.com:8080/home/client-side-https.

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

Encrypt and Share JWT Sessions

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

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

Encrypt JWT Sessions

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

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

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

NOTE

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

2. Add the following route to IG:

   1. Linux
   2. Windows

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

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

```
{
  "name": "jwt-session-encrypt",
  "heap":  [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file":
"/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePassword": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{
        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
      }]
    }
  },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    }
  ],
  "session": {
    "type": "JwtSession",
    "config": {
      "authenticatedEncryptionSecretId":
"jwtsession.symmetric.secret.id",
      "encryptionMethod": "A256CBC-HS512",
      "secretsProvider": ["KeyStoreSecretStore-1"],
      "cookie": {
        "name": "IG",
        "domain": ".example.com"
      }
    }
  },
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "reason": "OK",
      "headers": {
        "Content-Type": [ "text/plain" ]
```

```
        },
        "entity": "Hello world!"
      }
    },
    "condition": "${request.uri.path == '/jwt-session-
encrypt'}"
}
```

Notice the following features of the route:

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

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

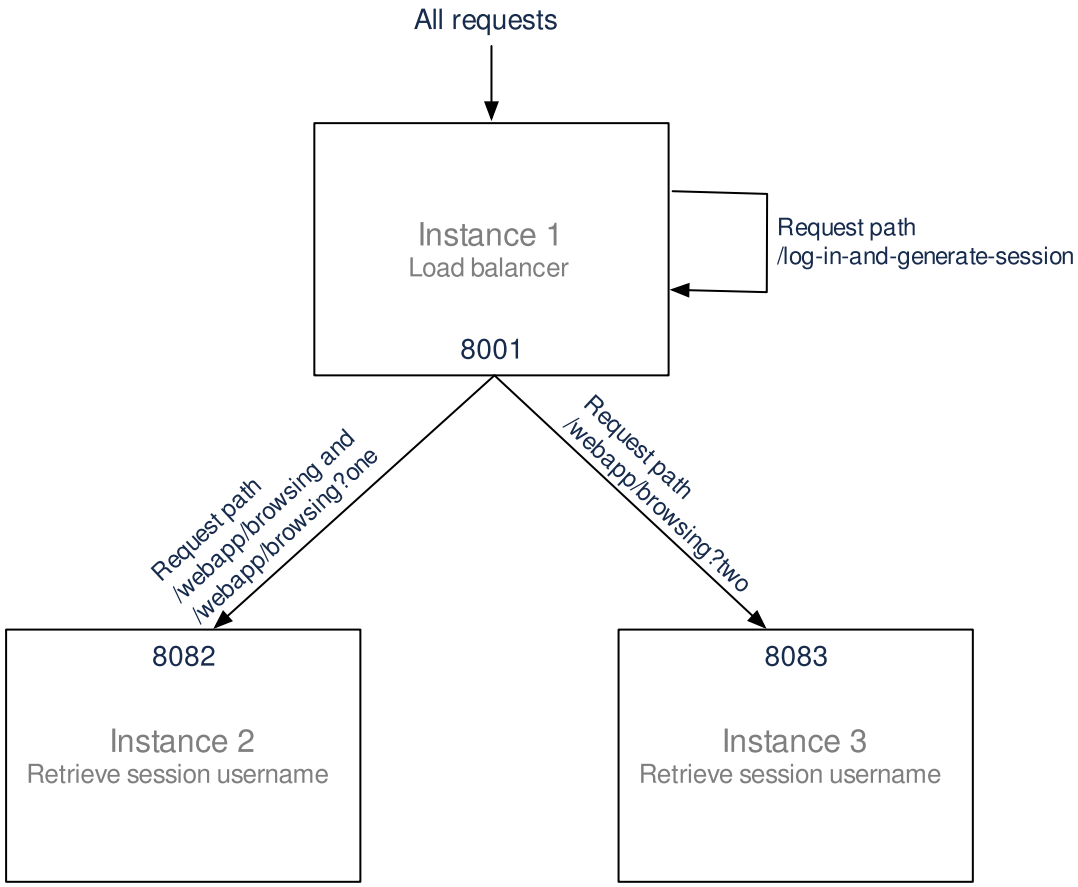
```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='
```

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

Share JWT Session Between Multiple Instances of IG

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

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



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

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

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

NOTE

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

2. Set up and start the first instance of IG, which acts as the load balancer:
   - Download and install the instance to `/path/to/instance1`.
   - Create a configuration directory for the instance:

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

a. Add the following route to IG:

1. Linux
2. Windows

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

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

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

```
      "secretsProvider": ["KeyStoreSecretStore-1"],
      "cookie": {
        "name": "IG",
        "domain": ".example.com"
      }
    }
  },
  "handler": {
    "type": "DispatchHandler",
    "config": {
      "bindings": [{
        "condition": "${find(request.uri.path,
'/webapp/browsing') and
(contains(request.uri.query, 'one') or
empty(request.uri.query))}",
        "baseURI":
"http://openig.example.com:8082",
        "handler": "ReverseProxyHandler"
      }, {
        "condition": "${find(request.uri.path,
'/webapp/browsing') and contains(request.uri.query,
'two')}",
        "baseURI":
"http://openig.example.com:8083",
        "handler": "ReverseProxyHandler"
      }, {
        "condition": "${find(request.uri.path,
'/log-in-and-generate-session')}",
        "handler": {
          "type": "Chain",
          "config": {
            "filters": [{
              "type": "AssignmentFilter",
              "config": {
                "onRequest": [{
                  "target":
"${session.authUsername}",
                  "value": "Sam Carter"
                }]
              }
            }],
            "handler": {
              "type": "StaticResponseHandler",
              "config": {
                "status": 200,
```

```
                "headers": {
                  "Content-Type": [ "text/html" ]
                },
                "entity": "<html><body>Sam Carter
logged IN. (JWT session generated)</body></html>"
              }
            }
          }
        }
      }]
    }
  },
  "capture": "all"
}
```

Notice the following features of the route:

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

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

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

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

   - Start the instance on port `8001`:

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

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

   - Download and install the instance to `/path/to/instance2`
   - Create a configuration directory for the instance:

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

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

```
{
  "name": "instance2-retrieve-session-username",
  "heap":  [{
    "name": "KeyStoreSecretStore-1",
    "type": "KeyStoreSecretStore",
    "config": {
      "file":
"/path/to/secrets/jwtsessionkeystore.pkcs12",
      "storeType": "PKCS12",
      "storePassword": "keystore.secret.id",
      "secretsProvider": ["SystemAndEnvSecretStore-1"],
      "mappings": [{
        "secretId": "jwtsession.symmetric.secret.id",
        "aliases": ["symmetric-key"]
      }]
    }
  },
    {
      "name": "SystemAndEnvSecretStore-1",
      "type": "SystemAndEnvSecretStore"
    }
  ],
  "session": {
    "type": "JwtSession",
    "config": {
      "authenticatedEncryptionSecretId":
"jwtsession.symmetric.secret.id",
      "encryptionMethod": "A256CBC-HS512",
      "secretsProvider": ["KeyStoreSecretStore-1"],
      "cookie": {
        "name": "IG",
        "domain": ".example.com"
      }
    }
  },
  "handler": {
    "type": "StaticResponseHandler",
    "config": {
      "status": 200,
      "headers": {
        "Content-Type": [ "text/html" ]
      },
```

```
      "entity": "<html><body>${session.authUsername!=
null?'Hello, '.concat(session.authUsername).concat('
!'):'Session.authUsername is not defined'}! (instance2)
</body></html>"
    }
  },
  "condition": "${find(request.uri.path,
'/webapp/browsing')}",
  "capture": "all"
}
```

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

- The route matches the condition `/webapp/browsing`. When a request matches `/webapp/browsing`, the DispatchHandler dispatches it to instance 2.
- The StaticResponseHandler displays information from the session context.

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

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

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

- Start the instance on port `8082`:

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

4. Set up and start the third instance of IG:
   - Download and install the instance to `/path/to/instance3`
   - Create the configuration directory:

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

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

```
{
  "name": "instance3-retrieve-session-username",
  "heap":  [{
```

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

```
  "condition": "${find(request.uri.path,
'/webapp/browsing')}",
  "capture": "all"
}
```

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

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

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

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

- Start the instance on port `8083`:

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

6. Test the setup:

- Access instance 1, to generate a session:

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

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

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

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

```
$ curl -v
http://openig.example.com:8001/webapp/browsing?one --
header "cookie:IG=<JWT cookie>"

GET /webapp/browsing?one HTTP/1.1
...
```

```
cookie: IG=eyJ...QHyI
...
HTTP/1.1 200 OK
...
Hello, Sam Carter !! (instance2)
```

Note that instance 2 can access the session info.

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

```
$ curl -v
http://openig.example.com:8001/webapp/browsing?two --
header "cookie:IG=<JWT cookie>"

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

Note that instance 3 can access the session info.

Prepare For Load Balancing and Failover

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

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

IG saves state information in the following ways:

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

StaticRequestFilters, that can store information in the context. Most filters depend on information in the request, response, or context, some of which is first stored by IG.

IG retrieves information locally in the following ways:

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

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

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

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

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

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

Secure Connections

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

When IG is running in web container mode, and acting as a server, the TLS connection is configured in the container. When IG is running in standalone mode, and acting as a server, the TLS connection is configured in `admin.json`.

When IG is acting as a client, the TLS connection is configured in the ReverseProxyHandler. For details, see Configure IG For HTTPS (Client-Side) and ReverseProxyHandler.

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

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

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

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

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

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

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

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

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