

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

Learn about upgrade between supported versions of IG in [Product Support Lifecycle Policy | PingGateway and Agents](#) .

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.

Planning task	Description
Understand the impact	<p>Decide whether you need to change the configuration of your deployment for this release, and evaluate the work involved.</p> <p>Make sure you meet all of the <u>requirements</u> for the new version. In particular, make sure that you have a recent, supported Java version.</p>
Plan for server downtime	<p>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.</p> <p>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.</p>
Back up	<p>The IG configuration is a set of files, including <code>admin.json</code>, <code>config.json</code>, <code>logback.xml</code>, <code>routes</code>, and <code>scripts</code>. Back up the IG configuration and store it in version control, so that you can roll back if something goes wrong.</p> <p>Also back up any tools scripts that you have edited for your deployment, and any trust stores used to connect securely.</p>
Plan for rollback	<p>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.</p> <p>For IG servers, roll back by restoring a backed-up configuration.</p>
Prepare a test environment	<p>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.</p> <p>The test environment should resemble your production environment as closely as possible.</p>

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 [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 <code>connectors</code> property of <u>admin.json</u> .	Configure in the <code>Connector</code> element of <code>/path/to/tomcat/conf/server.xml</code> : <pre><Connector port="8080" protocol="HTTP/1.1" connectionTimeout="20000" " redirectPort="8443" /></pre>

Feature	Standalone	Tomcat
HTTPS server-side configuration	<p>Create a keystore, set up secrets, and configure secrets stores, ports, and ServerTlsOptions in admin.json.</p> <p>For information, see Configure IG for HTTPS (server-side) in standalone mode.</p>	<p>Create a keystore, and set up the SSL port in the Connector element of <code>/path/to/tomcat/conf/server.xml</code>.</p> <p>For information, see Configure IG for HTTPS (server-side) in Tomcat.</p>
Session cookie name	Configure the session property of admin.json .	Configure <code>WEB-INF/web.xml</code> when you unpack the IG .war file.
Access logs	<p>Configure in the Audit framework.</p> <p>For information, see Auditing your deployment and Audit framework.</p>	Configure with <code>AccessLogValve</code> .
JDBC datasource	<p>Configure with the <code>JdbcDataSource</code> object.</p> <p>For information, see JdbcDataSource.</p> <p>For an example, see Log in with credentials from a database.</p>	Configure in the <code>GlobalNamingResources</code> element of <code>/path/to/tomcat/conf/server.xml</code> .
Environment variables	Configure in <code>\$HOME/.openig/bin/env.sh</code> , where <code>\$HOME/.openig</code> is the instance directory.	Configure in <code>/path/to/tomcat/bin/setenv.sh</code> .
Jar files	Add to <code>\$HOME/.openig/extra</code> , where <code>\$HOME/.openig</code> is the instance directory.	Add to to web container classpath; for example <code>/path/to/tomcat/webapps/ROOT/WEB-INF/lib</code> .

Prepare to install

Requirements

Make sure that your installation meets the requirements in [Requirements](#).

Create an IG service account

To limit the impact of a security breach, install and run IG from a dedicated service account. This is optional when you are evaluating IG, but essential in production installations.

A hacker is constrained by the rights granted to the user account where IG runs; therefore, never run IG as root user.

1. In a terminal window, use a command similar to the following to create a service account:

Linux

Windows

```
$ sudo /usr/sbin/useradd \  
--create-home \  
--comment "Account for running IG" \  
--shell /bin/bash IG
```

2. Apply the principle of least privilege to the account, for example:
 - Read/write permissions on the installation directory, `/path/to/identity-gateway`.
 - Execute permissions on the scripts in the installation `bin` directory, `/path/to/identity-gateway/bin`.

Prepare the network

Configure the network to include the hosts.

1. Add the following additional entry to your host file:

Linux

Windows

```
/etc/hosts
```

```
127.0.0.1 localhost ig.example.com app.example.com
```

```
am.example.com
```

For more information about host files, see the Wikipedia entry, [Hosts \(file\)](#).

Install IG in standalone mode

Download and start IG in standalone mode

Download the IG .zip file

1. Create a local installation directory for IG. The examples in this section use `/path/to`.
2. Download `IG-7.2.0.zip` from the [ForgeRock BackStage download site](#), and copy the `.zip` file to the installation directory:

```
$ cp IG-7.2.0.zip /path/to/IG-7.2.0.zip
```

3. Unzip the file:

```
$ unzip IG-7.2.0.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:

Linux

Windows

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

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 the instance directory:

Linux	Windows
--------------	----------------

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

2. Check that IG is running in one of the following ways:

- Ping IG at `http://ig.example.com:8080/openig/ping`, and make sure an HTTP 200 is returned.
- Access the IG welcome page at `http://ig.example.com:8080`.
- When IG is running in development mode, display the product version and build information at `http://ig.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:

- `vertex`: 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,
    "vertex": {
```

```
"maxWebSocketFrameSize": 128000,  
"maxWebSocketMessageSize": 256000,  
"compressionLevel": 4  
}  
}]  
}
```

For more information, see [AdminHttpApplication \(admin.json\)](#).

Stop IG

Use the `stop.sh` script to stop an instance of IG, specifying the instance directory as an argument. If the instance directory is not specified, IG uses the default instance directory:

Linux	Windows
<pre>\$ /path/to/identity-gateway/bin/stop.sh \$HOME/.openig</pre>	

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](#).

Using keys and certificates with IG in standalone mode

The examples in this doc set use self-signed certificates, but your deployment is likely to use certificates issued by a certificate authority (CA certificates).

The way to obtain CA certificates depends on the certificate authority that you are using, and is not described in this document. As an example, see [Let's Encrypt](#) [↗](#).

When IG is in web container mode, the way to integrate CA certificates depends on the web container type; see your web container documentation for more information. When IG is in standalone mode, integrate CA certificates by using secret stores:

- For PEM files, use a [FileSystemSecretStore](#) and [PemPropertyFormat](#)
- For PKCS12 keystores, use a [KeyStoreSecretStore](#)

For examples, see [Serve the same certificate for TLS connections to all server names](#).

Note the following points about using secrets:

- When IG in standalone mode starts up, it listens for HTTPS connections, using the `ServerTlsOptions` configuration in `admin.json`. The keys and certificates are fetched only once, at startup.
- Keys and certificates must be present at startup.
- If keys or certificates change, you must to restart IG.

For information about secret stores provided in IG, see [Secrets object and secret stores](#).

Serve the same certificate for TLS connections to all server names

This example uses PEM files and a PKCS12 keystore for self-signed certificates, but you can adapt it to use official (non self-signed) keys and certificates.

Before you start, install IG in standalone mode, as described in [Download and start IG in standalone mode](#).

1. Locate a directory for the secrets, for example, `/path/to/secrets`.
2. Create self-signed keys in one of the following ways. If you have your own keys, use them and skip this step.

▼ [Use your own keys](#)

If you have your own keys, use them and skip this step.

▼ [Set up a self-signed certificate in a \(PKCS12\) keystore](#)

1. Create the keystore, replacing `/path/to/secrets` with your path:

```
$ keytool \  
-genkey \  
-alias https-connector-key \  
-keyalg RSA \  
-keystore /path/to/secrets/IG-keystore \  
-storepass password \  
-keypass password \  
-dname "CN=ig.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`.

2. In the secrets directory, add a file called `keystore.pass`, containing the keystore password `password`:

```
$ cd /path/to/secrets/  
$ echo -n 'password' > keystore.pass
```

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

▼ [Set up self-signed certificate stored in PEM file](#)

- a. Locate a directory for secrets, and go to it:

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

- b. Create the following secret key and certificate pair as PEM files:

```
$ openssl req \  
-newkey rsa:2048 \  
-new \  
-nodes \  
-x509 \  
-days 3650 \  
-subj \  
"/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C=fr" \  
\  
-keyout ig.example.com-key.pem \  
-out ig.example.com-certificate.pem
```

Two PEM files are created, one for the secret key, and another for the associated certificate.

- c. Map the key and certificate to the same secret ID in IG:

```
$ cat ig.example.com-key.pem ig.example.com-  
certificate.pem > key.manager.secret.id.pem
```

3. Set up TLS on IG in one of the following ways:

▼ [Keys stored in a \(PKCS12\) keystore](#)

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

Linux	Windows
--------------	----------------

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

```

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

▼ [Keys stored in PEM file](#)

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

Linux

Windows

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

```
{
  "connectors": [
    {
      "port": 8080
    },
    {
      "port": 8443,
      "tls": "ServerTlsOptions-1"
    }
  ],
  "heap": [
    {
      "name": "ServerTlsOptions-1",
      "type": "ServerTlsOptions",
      "config": {
        "keyManager": {
          "type": "SecretsKeyManager",
          "config": {
            "signingSecretId": "key.manager.secret.id",
            "secretsProvider": "ServerIdentityStore"
          }
        }
      }
    }
  ]
}
```

```

    }
  }
},
{
  "name": "ServerIdentityStore",
  "type": "FileSystemSecretStore",
  "config": {
    "format": "PLAIN",
    "directory": "/path/to/secrets",
    "suffix": ".pem",
    "mappings": [{
      "secretId": "key.manager.secret.id",
      "format": {
        "type": "PemPropertyFormat"
      }
    }]
  }
}
]
}

```

Notice how this file differs to that for the keystore-based approach:

- The ServerIdentityStore is a FileSystemSecretStore.
- The FileSystemSecretStore reads the keys that are stored as file in the PEM standard format.

4. Start IG:

Linux	Windows
<pre> \$ /path/to/identity-gateway/bin/start.sh started in 1234ms on ports : [8080 8443] </pre>	

Serve different certificates for TLS connections to different server names

This example uses PEM files for self-signed certificates, but you can adapt it to use official (non self-signed) keys and certificates.


```
-keyout wildcard.example.com-key.pem \  
-out wildcard.example.com-certificate.pem
```

ii. Map the key and certificate to the same secret ID in IG:

```
$ cat wildcard.example.com-key.pem  
wildcard.example.com-certificate.pem >  
wildcard.secret.id.pem
```

c. For other, unmapped servers

i. Create a key and certificate:

```
$ openssl req \  
-newkey rsa:2048 \  
-new \  
-nodes \  
-x509 \  
-days 3650 \  
-subj  
"/CN=un.mapped.com/OU=example/O=com/L=fr/ST=fr/C=fr  
" \  
-keyout default.example.com-key.pem \  
-out default.example.com-certificate.pem
```

ii. Map the key and certificate to the same secret ID in IG:

```
$ cat default.example.com-key.pem  
default.example.com-certificate.pem >  
default.secret.id.pem
```

3. Add the following file to IG, replacing `/path/to/secrets` with your path, and then restart IG:

Linux

Windows

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

```
{  
  "connectors": [  
    {  
      "port": 8080  
    },  
    {
```

```

    "port": 8443,
    "tls": "ServerTlsOptions-1"
  }
],
"heap": [
  {
    "name": "ServerTlsOptions-1",
    "type": "ServerTlsOptions",
    "config": {
      "sni": {
        "serverNames": {
          "ig.example.com": "key.manager.secret.id",
          "*.example.com": "wildcard.secret.id"
        },
        "defaultSecretId": "default.secret.id",
        "secretsProvider": "ServerIdentityStore"
      }
    }
  },
  {
    "name": "ServerIdentityStore",
    "type": "FileSystemSecretStore",
    "config": {
      "format": "PLAIN",
      "directory": "path/to/secrets",
      "suffix": ".pem",
      "mappings": [
        {
          "secretId": "key.manager.secret.id",
          "format": {
            "type": "PemPropertyFormat"
          }
        },
        {
          "secretId": "wildcard.secret.id",
          "format": {
            "type": "PemPropertyFormat"
          }
        },
        {
          "secretId": "default.secret.id",
          "format": {
            "type": "PemPropertyFormat"
          }
        }
      ]
    }
  }
]

```

```
    ]
  }
}
]
}
```

Notice the following features of the file:

- The `ServerTlsOptions` object maps two servers to secret IDs, and includes a default secret ID
 - The secret IDs correspond to the secret IDs in the `FileSystemSecretStore`, and the PEM files generated in an earlier step.
4. Run the following commands to request TLS connections to different servers, using different certificates:
- a. Connect to `ig.example.com`, and note that the certificate subject corresponds to the certificate created for `ig.example.com`:

```
$ openssl s_client -connect localhost:8443 -servername
ig.example.com

...
Server certificate
-----BEGIN CERTIFICATE-----
MII...dZC
-----END CERTIFICATE-----
subject=/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/
C=fr
issuer=/CN=ig.example.com/OU=example/O=com/L=fr/ST=fr/C
=fr
```

- b. Connect to `other.example.com`, and note that the certificate subject corresponds to the certificate created with the wildcard, `*.example.com`:

```
$ openssl s_client -connect localhost:8443 -servername
other.example.com

...
Server certificate
-----BEGIN CERTIFICATE-----
MII...fY=
-----END CERTIFICATE-----
subject=/CN=*.example.com/OU=example/O=com/L=fr/ST=fr/C
=fr
```

```
issuer=/CN=*.example.com/OU=example/O=com/L=fr/ST=fr/C=fr
```

- c. Connect to `unmapped.site.com`, and note that the certificate subject corresponds to the certificate created for the default secret ID:

```
$ openssl s_client -connect localhost:8443 -servername unmapped.site.com

...
Server certificate
-----BEGIN CERTIFICATE-----
MII..rON
-----END CERTIFICATE-----
subject=/CN=un.mapped.com/OU=example/O=com/L=fr/ST=fr/C=fr
issuer=/CN=un.mapped.com/OU=example/O=com/L=fr/ST=fr/C=fr
```

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:

Linux

Windows

```
$ IG_ROUTER_SCAN_INTERVAL='2 seconds' /path/to/identity-
```

```
gateway/bin/start.sh
```

Define environment variables for startup, runtime, and stop

IG provides the following environment variables for Java runtime options:

IG_OPTS

(Optional) Java runtime options for IG and its startup process, such as JVM memory sizing options.

Include all options that are not shared with the `stop` script.

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

Linux

Windows

```
# 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 IG_OPTS environment variable
export IG_OPTS="${IG_OPTS} ${JVM_OPTS} ${JSSE_OPTS}"
```

JAVA_OPTS

(Optional) Java runtime options for IG include all options that are shared by the start and stop script.

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 a directory to hold .jar files for IG extensions:

Linux

Windows

```
$HOME/.openig/extra
```

When IG starts up, the JVM loads .jar files in the `extra` directory.

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-7.2.0.war from the [ForgeRock BackStage download site](#).

4. Copy the IG-7.2.0.war to the Tomcat webapps directory, as ROOT.war :

```
$ cp IG-7.2.0.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://ig.example.com:8080/openig/ping>, and make sure an HTTP 200 is returned.
- Access the IG welcome page at <http://ig.example.com:8080>.
- When IG is running in development mode, display the product version and build information at <http://ig.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 \(client-side\)](#).

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.

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

a. Locate a directory for secrets, and go to it:

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

b. Create a keystore:

```
$ keytool \  
-genkey \  
-alias tomcat \  
-keyalg RSA \  
-keystore keystore \  
-storetype PKCS12 \  
-storepass password \  
-keypass password \  
-dname "CN=ig.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 .

2. Add an entry similar to the following in `/path/to/tomcat/conf/server.xml` , replacing `/path/to/tomcat/conf/keystore` with the path to your keystore:

```
<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>
```

3. Restart Tomcat.

Configure SameSite for HTTP session cookies in Tomcat

1. Change the cookie processor element in `/path/to/tomcat/webapps/manager/META-INF/context.xml`, to one of the following values:

- `none` : The browser always sends cookies in cross-site requests
- `lax` : The browser sends cookies only in same-site requests and cross-site top-level GET requests
- `strict` : The browser never sends cookies in cross-site requests

For example, the following line sets the value to `none` :

```
<CookieProcessor  
  className="org.apache.tomcat.util.http.Rfc6265CookiePro  
  cessor" sameSiteCookies="none" />
```

When you access Tomcat through HTTPS, the `secure` flag is automatically set on the cookie.

2. Restart Tomcat.

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 [Load Balancing 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.

SAML in Deployments With Multiple Instances of IG

IG uses a Java fedlet to implement SAML. When IG acts as a SAML service provider, the session information is stored in the fedlet, not the session cookie. In deployments that use multiple instances of IG as a SAML service provider, it is therefore necessary to set up sticky sessions so that requests always hit the instance where the SAML interaction was started.

For information, see [Session state considerations](#) in AM's *SAML v2.0 guide*.

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-7.2.0.war` from the [ForgeRock BackStage download site](#).
3. Copy the `.war` file:

```
$ cp IG-7.2.0.war /path/to/jetty/webapps/IG-7.2.0.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://ig.example.com:8080/openig/ping>, and make sure an HTTP 200 is returned.
- Access the IG welcome page at <http://ig.example.com:8080>.
- When IG is running in development mode, display the product version and build information at <http://ig.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=ig.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,console-capture,http,https

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

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

NOTE

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

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

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

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-7.2.0.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://ig.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 SameSite for HTTP session cookies in Jetty

1. Add a session `<cookie-config>` element in `/path/to/jetty/WEB-INF/web.xml`, and configure the `<comment>` element with one of the following values:
 - `SAME_SITE_NONE`: The browser always sends cookies in cross-site requests
 - `SAME_SITE_LAX`: The browser sends cookies only in same-site requests and cross-site top-level GET requests
 - `SAME_SITE_STRICT`: The browser never sends cookies in cross-site requests

The following example sets the value to `none` :

```
<session-config>
  <cookie-config>
    <name>IG_SESSIONID</name>
```

```
<comment>__SAME_SITE_NONE__</comment>
<http-only>true</http-only>
</cookie-config>
</session-config>
```

2. Restart Jetty.

Configure access to 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.

SAML in Deployments With Multiple Instances of IG

IG uses a Java fedlet to implement SAML. When IG acts as a SAML service provider, the session information is stored in the fedlet, not the session cookie. In deployments that use multiple instances of IG as a SAML service provider, it is therefore necessary to set up sticky sessions so that requests always hit the instance where the SAML interaction was started.

For information, see [Session state considerations](#) in *AM's SAML v2.0 guide*.

Install IG in JBoss EAP

Downloading and starting 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-7.2.0.war` from the [ForgeRock BackStage download site](#).
4. Copy the `IG-7.2.0.war` to the JBoss deployment directory:

```
$ cp IG-7.2.0.war
/path/to/jboss/standalone/deployments/IG-7.2.0.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://ig.example.com:8080/openig/ping>, and make sure an HTTP 200 is returned.
 - Access the IG welcome page at <http://ig.example.com:8080>.
 - When IG is running in development mode, display the product version and build information at <http://ig.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:ig.example.com}")
/interface=public/:write-attribute(name="inet-address",
value="{jboss.bind.address:ig.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=ig.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:

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

8. Make sure IG is deployed on port 8080 :

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

```
--controller=ig.example.com:9990 command="deployment list"
```

9. Enable SSL:

- Enable the SSL server:

```
$ $JBOSS_HOME/bin/jboss-cli.sh --connect \  
--controller=ig.example.com:9990 command="security \  
enable-ssl-http-server \  
--key-store-path=$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://ig.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 SameSite for HTTP session cookies in JBoss

IMPORTANT

JBoss supports SameSite from version 7.3.2. If you use an earlier version, and the following error occurs, upgrade to JBoss 7.3.2.

```
Error 'invalid token' during authentication because of 'The supplied token is invalid'
```

For more information, see the JBoss documentation.

1. Add a configuration element in `/path/to/jboss/webapps/ROOT/WEB-INF/undertow-handlers.conf`, to set one of the following values for the SameSite cookie mode:
 - `none`: The browser always sends cookies in cross-site requests
 - `lax`: The browser sends cookies only in same-site requests and cross-site top-level GET requests
 - `strict`: The browser never sends cookies in cross-site requests

The following example sets the the SameSite cookie mode to `lax`:

```
path(/)->samesite-cookie(Lax)
```

2. Restart JBoss.

Set up logs and configuration files

The following table summarizes the default location of the IG configuration and logs.

Purpose	Default location on Linux	Default location on Windows
Log messages from IG and third-party dependencies	\$HOME/.openig/logs	%appdata%\OpenIG\logs
Administration (admin.json) Gateway (config.json)	\$HOME/.openig/config	%appdata%\OpenIG\config
Routes (Route)	\$HOME/.openig/config/routes	%appdata%\OpenIG\config\routes
SAML 2.0	\$HOME/.openig/SAML	%appdata%\OpenIG\OpenIG\SAML
Groovy scripts for scripted filters and handlers, and other objects	\$HOME/.openig/scripts/groovy	%appdata%\OpenIG\scripts\groovy
Temporary directory To change the directory, configure <code>temporaryDirectory</code> in admin.json	\$HOME/.openig/tmp	%appdata%\OpenIG\OpenIG\tmp
JSON schema for custom audit To change the directory, configure <code>topicsSchemasDirectory</code> in AuditService .	\$HOME/.openig/audit-schemas	%appdata%\OpenIG\OpenIG\audit-schemas

Secure the configuration and logs

For the `/logs` , `/tmp` , and all configuration directories, allow the following access:

- Highest privilege the IG system account.
- Least privilege for specific accounts, on a case-by-case basis
- No privilege for all other accounts, by default

Change the default location of the configuration

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

Linux	Windows
<pre>\$HOME/.openig</pre>	

For IG in standalone mode, installed with the Windows startup batch script, the base location is configured in the batch script. For other installations, change the default base location in the following ways:

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

Linux	Windows
<pre>\$ export IG_INSTANCE_DIR=/path/to/instance-dir</pre>	

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

Linux	Windows
<pre>\$ /path/to/identity-gateway/bin/start.sh \$HOME/.openig</pre>	

- 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
```

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

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

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

```
$ keytool -importcert \  
-alias ig-sampleapp \  
-file tls/sampleapp-cert.pem \  
-keystore reverseproxy-truststore.p12 \  
-storetype pkcs12 \  
-storepass password  
  
...  
Trust this certificate? [no]: yes  
  
Certificate was added to keystore
```

NOTE

Because `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.

6. Add the following route to serve static resources, such as .css, for the sample application:

Linux

Windows

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

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

7. Add the following route to IG:

Linux

Windows

```
$HOME/.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.

8. Test the setup:

- a. Start the sample application

```
$ java -jar sampleapp_install_dir/IG-sample-application-7.2.0.jar
```

- b. Go to <http://ig.example.com:8080/home/client-side-https>.

The request is proxied transparently to the sample application, on the TLS port 8444 .

- c. Check the route log for a line like this:

```
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 \  
-password password
```

```
-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:

Linux

Windows

```
$HOME/.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,
      "headers": {
        "Content-Type": ["text/plain; charset=UTF-8"]
      },
      "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:

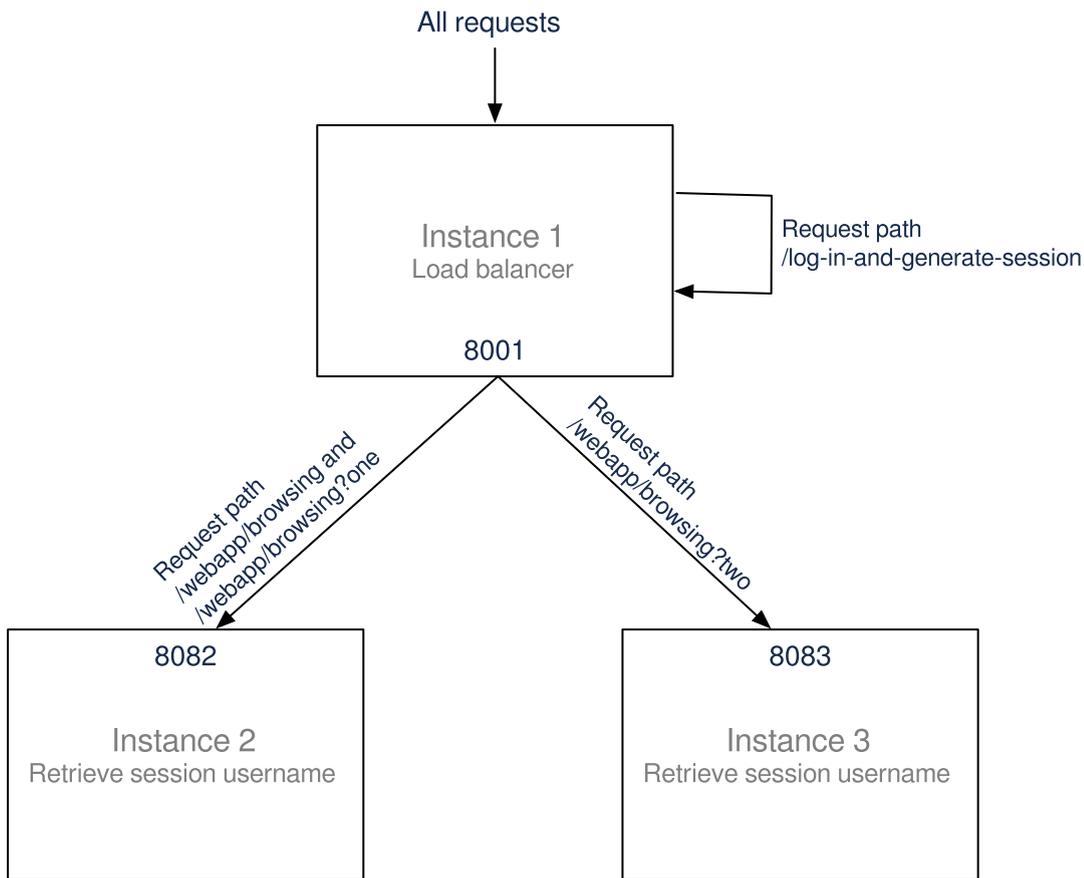
```
$ export KEYSTORE_SECRET_ID='cGFzc3dvcmQ='
```

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

Share JWT sessions 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:

Linux

Windows

```
$HOME/.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": "${find(request.uri.path,
'/webapp/browsing') and
(contains(request.uri.query, 'one') or
empty(request.uri.query))}",
        "baseURI": "http://ig.example.com:8082",
        "handler": "ReverseProxyHandler"
      }, {
        "condition": "${find(request.uri.path,
'/webapp/browsing') and contains(request.uri.query,
'two')}",
        "baseURI": "http://ig.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;
charset=UTF-8" ]
              },
              "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.

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 8001 :

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

4. 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; charset=UTF-8" ]
      },
      "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.

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 8082 :

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

6. 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; charset=UTF-8" ]
    },
    "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`.

7. 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/
```

8. Test the setup:

- Access instance 1, to generate a session:

```
$ curl -v http://ig.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://ig.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://ig.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 `SamIFederationHandler` 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`, `AuthorizationCodeOAuth2ClientFilters`, `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 standalone mode, and acting as a server, the TLS connection is configured in `admin.json`. When IG is running in web container mode, and acting as a server, the TLS connection is configured in the container.

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](#) .

Was this helpful?  

Copyright © 2010-2024 ForgeRock, all rights reserved.