



# Red Hat JBoss Fuse 6.1

## Security Guide

Making it safe for your systems to work together



# Red Hat JBoss Fuse 6.1 Security Guide

---

Making it safe for your systems to work together

JBoss A-MQ Docs Team

Content Services

[fuse-docs-support@redhat.com](mailto:fuse-docs-support@redhat.com)

## Legal Notice

Copyright © 2013 Red Hat.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## Abstract

This guide describes how to secure the Red Hat JBoss Fuse container, the web console, message brokers, routing and integration components, web and RESTful services, and it provides a tutorial on LDAP authentication.

## Table of Contents

<b>CHAPTER 1. SECURITY ARCHITECTURE</b> .....	<b>3</b>
1.1. OSGI CONTAINER SECURITY	3
1.2. APACHE ACTIVEMQ SECURITY	4
1.3. APACHE CAMEL SECURITY	6
<b>CHAPTER 2. SECURING THE RED HAT JBOSS FUSE CONTAINER</b> .....	<b>9</b>
2.1. JAAS AUTHENTICATION	9
2.2. ENABLING LDAP AUTHENTICATION	30
2.3. ENABLING REMOTE JMX SSL	33
2.4. CONFIGURING ROLES FOR THE ADMINISTRATIVE PROTOCOLS	37
2.5. USING ENCRYPTED PROPERTY PLACEHOLDERS	38
<b>CHAPTER 3. SECURING THE JETTY HTTP SERVER</b> .....	<b>43</b>
JETTY SERVER	43
CREATE X.509 CERTIFICATE AND PRIVATE KEY	43
ENABLING SSL/TLS	43
CONNECT TO THE SECURE CONSOLE	45
ADVANCED JETTY SECURITY CONFIGURATION	46
<b>CHAPTER 4. SECURING THE CAMEL ACTIVEMQ COMPONENT</b> .....	<b>47</b>
4.1. SECURE ACTIVEMQ CONNECTION FACTORY	47
4.2. EXAMPLE CAMEL ACTIVEMQ COMPONENT CONFIGURATION	48
<b>CHAPTER 5. SECURING THE CAMEL JETTY COMPONENT</b> .....	<b>50</b>
5.1. ENABLING SSL/TLS SECURITY	50
5.2. BASIC AUTHENTICATION WITH JAAS	56
<b>CHAPTER 6. SECURING THE CAMEL CXF COMPONENT</b> .....	<b>62</b>
6.1. THE CAMEL CXF PROXY DEMONSTRATION	62
6.2. SECURING THE WEB SERVICES PROXY	65
6.3. DEPLOYING THE APACHE CAMEL ROUTE	69
6.4. SECURING THE WEB SERVICES CLIENT	71
<b>CHAPTER 7. LDAP AUTHENTICATION TUTORIAL</b> .....	<b>78</b>
7.1. TUTORIAL OVERVIEW	78
7.2. SET-UP A DIRECTORY SERVER AND BROWSER	78
7.3. ADD USER ENTRIES TO THE DIRECTORY SERVER	81
7.4. ENABLE LDAP AUTHENTICATION IN THE OSGI CONTAINER	87
7.5. CONFIGURING ACCESS TO OSGI ADMINISTRATIVE FUNCTIONS	92
7.6. ENABLE SSL/TLS ON THE LDAP CONNECTION	94
<b>APPENDIX A. MANAGING CERTIFICATES</b> .....	<b>98</b>
A.1. WHAT IS AN X.509 CERTIFICATE?	98
A.2. CERTIFICATION AUTHORITIES	99
A.3. CERTIFICATE CHAINING	100
A.4. SPECIAL REQUIREMENTS ON HTTPS CERTIFICATES	101
A.5. CREATING YOUR OWN CERTIFICATES	103
<b>APPENDIX B. ASN.1 AND DISTINGUISHED NAMES</b> .....	<b>109</b>
B.1. ASN.1	109
B.2. DISTINGUISHED NAMES	109
<b>INDEX</b> .....	<b>112</b>



# CHAPTER 1. SECURITY ARCHITECTURE

## Abstract

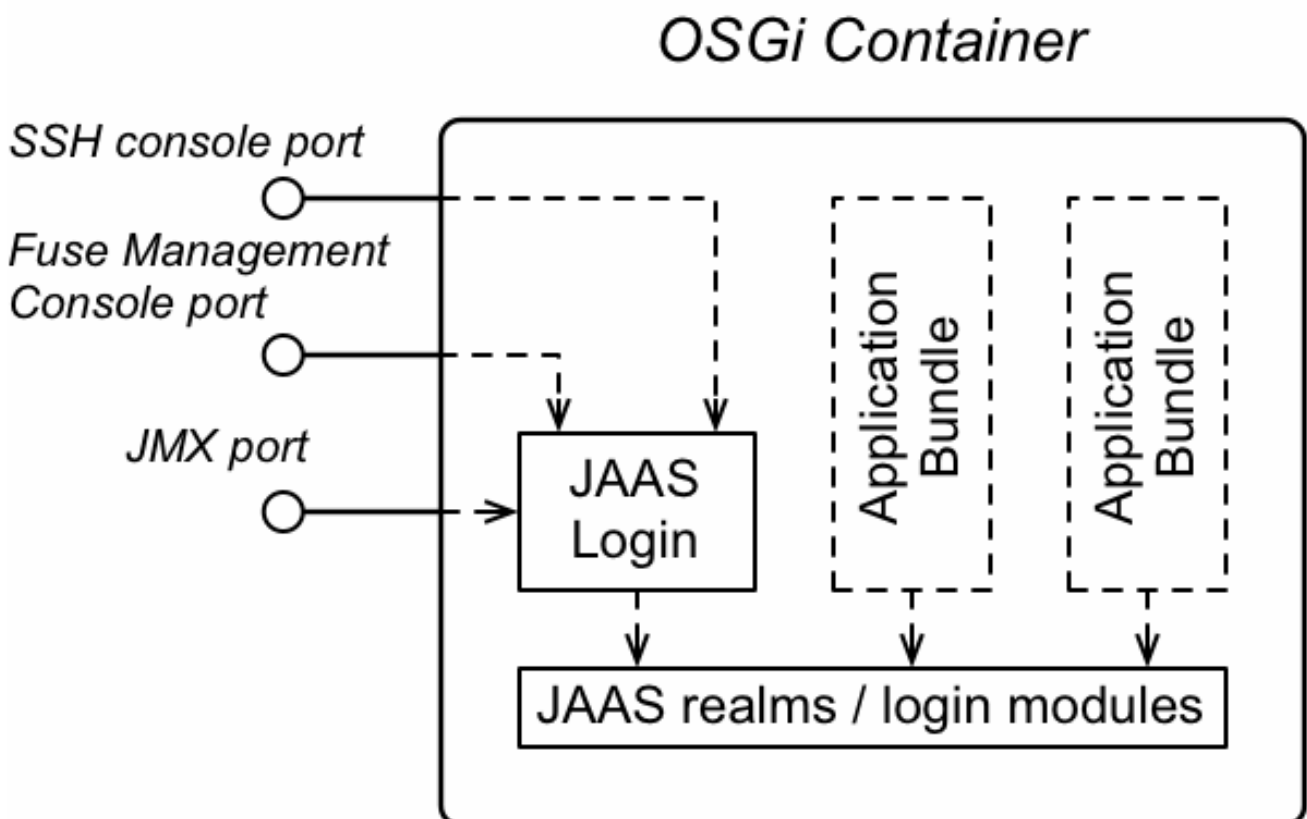
In the OSGi container, it is possible to deploy applications supporting a variety of security features. Currently, only the Java Authentication and Authorization Service (JAAS) is based on a common, container-wide infrastructure. Other security features are provided separately by the individual products and components deployed in the container.

## 1.1. OSGI CONTAINER SECURITY

### Overview

Figure 1.1, “OSGi Container Security Architecture” shows an overview of the security infrastructure that is used across the container and is accessible to all bundles deployed in the container. This common security infrastructure currently consists of a mechanism for making JAAS realms (or login modules) available to all application bundles. Other aspects of security are currently implemented separately in each of the Red Hat JBoss Fuse component products (Apache ActiveMQ, Apache Camel, and Apache CXF).

Figure 1.1. OSGi Container Security Architecture



### JAAS realms

A JAAS realm or login module is a plug-in module that provides authentication and authorization data to Java applications, as defined by the [Java Authentication and Authorization Service \(JAAS\)](#) specification.

Red Hat JBoss Fuse supports a special mechanism for defining JAAS login modules (in either a Spring or a blueprint file), which makes the login module accessible to all bundles in the container. This makes it

easy for multiple applications running in the OSGi container to consolidate their security data into a single JAAS realm.

## karaf realm

The OSGi container has a predefined JAAS realm, the **karaf** realm. Red Hat JBoss Fuse uses the **karaf** realm to provide authentication for remote administration of the OSGi runtime, for the Fuse Management Console, and for JMX management. The **karaf** realm uses a simple file-based repository, where authentication data is stored in the ***InstallDir/etc/users.properties*** file.

You can use the **karaf** realm in your own applications. Simply configure **karaf** as the name of the JAAS realm that you want to use. Your application then performs authentication using the data from the **users.properties** file.

## Console port

You can administer the OSGi container remotely either by connecting to the console port with a Karaf client or using the Karaf **ssh:ssh** command (see [chapter "Using Remote Connections to Manage a Container"](#) in ["Configuring and Running Red Hat JBoss Fuse"](#)). The console port is secured by a JAAS login feature that connects to the **karaf** realm. Users that try to connect to the console port will be prompted to enter a username and password that must match one of the accounts from the **karaf** realm.

## JMX port

You can manage the OSGi container by connecting to the JMX port (for example, using Java's JConsole). The JMX port is also secured by a JAAS login feature that connects to the **karaf** realm.

## Application bundles and JAAS security

Any application bundles that you deploy into the OSGi container can access the container's JAAS realms. The application bundle simply references one of the existing JAAS realms by name (which corresponds to an instance of a JAAS login module).

It is essential, however, that the JAAS realms are defined using the OSGi container's own login configuration mechanism—by default, Java provides a simple file-based login configuration implementation, but you *cannot* use this implementation in the context of the OSGi container.

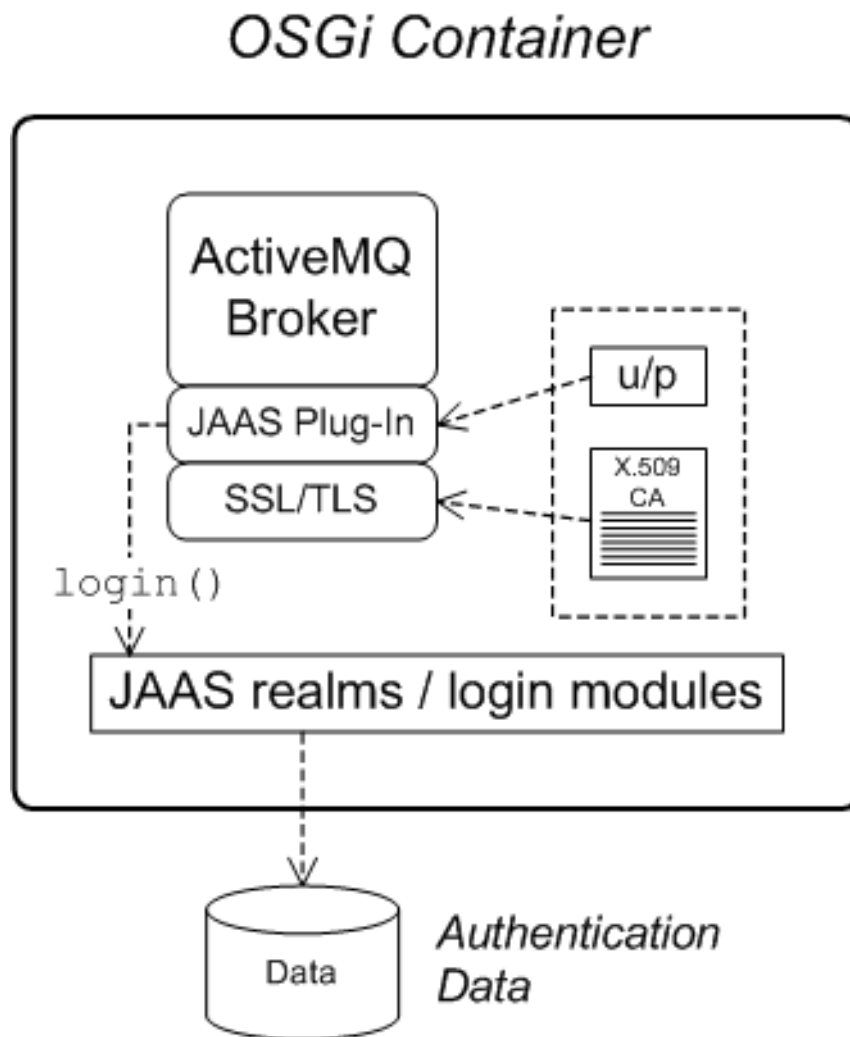
# 1.2. APACHE ACTIVEMQ SECURITY

## Overview

[Figure 1.2, "Apache ActiveMQ Security Architecture"](#) shows an overview of the Apache ActiveMQ security architecture. The main security features supported by Apache ActiveMQ are the SSL/TLS security layer and the JAAS security layer. The SSL/TLS security layer provides message encryption and identifies the broker to its clients, while the JAAS security layer identifies clients to the broker.



Figure 1.2. Apache ActiveMQ Security Architecture



### SSL/TLS security

Apache ActiveMQ supports the use of SSL/TLS to secure client-to-broker and broker-to-broker connections, where the underlying SSL/TLS implementation is provided by the Java Secure Socket Extension (JSSE). When deploying brokers and clients in an OSGi container, you cannot configure SSL/TLS security using JSSE system properties, however. You must either use XML configuration (for example, in a Spring or a blueprint file) or set the security properties by programming.

### JAAS security

Apache ActiveMQ also supports JAAS security, which typically requires clients to log on to the broker by providing username and password credentials. When deployed in an OSGi container, the broker's JAAS security must be integrated with the container's JAAS security (as described in [Section 1.1, "OSGi Container Security"](#)).

### JAAS plug-ins

To enable JAAS security in a broker, you install one of the supported JAAS plug-ins. Each of the JAAS plug-ins supports a different kind of credentials or implements a somewhat different login procedure. The following JAAS plug-ins are currently supported by Apache ActiveMQ:

- **jaasAuthenticationPlugin** supports authentication using JMS username/password credentials.

- **jaasCertificateAuthenticationPlugin** supports additional checking of the X.509 certificate received from a client (usable only in combination with SSL/TLS security).
- **jaasDualAuthenticationPlugin** is a hybrid version of the other two plug-ins. This plug-in checks the client's X.509 certificate, if and only if SSL/TLS is enabled. Otherwise, it falls back to checking the JMS username/password credentials.

## JAAS login modules

Apache ActiveMQ provides a number of different JAAS login module implementations, which are used to define JAAS realms. The role of a JAAS login module is to store authentication and authorization data. The following JAAS login modules are currently implemented by Apache ActiveMQ:

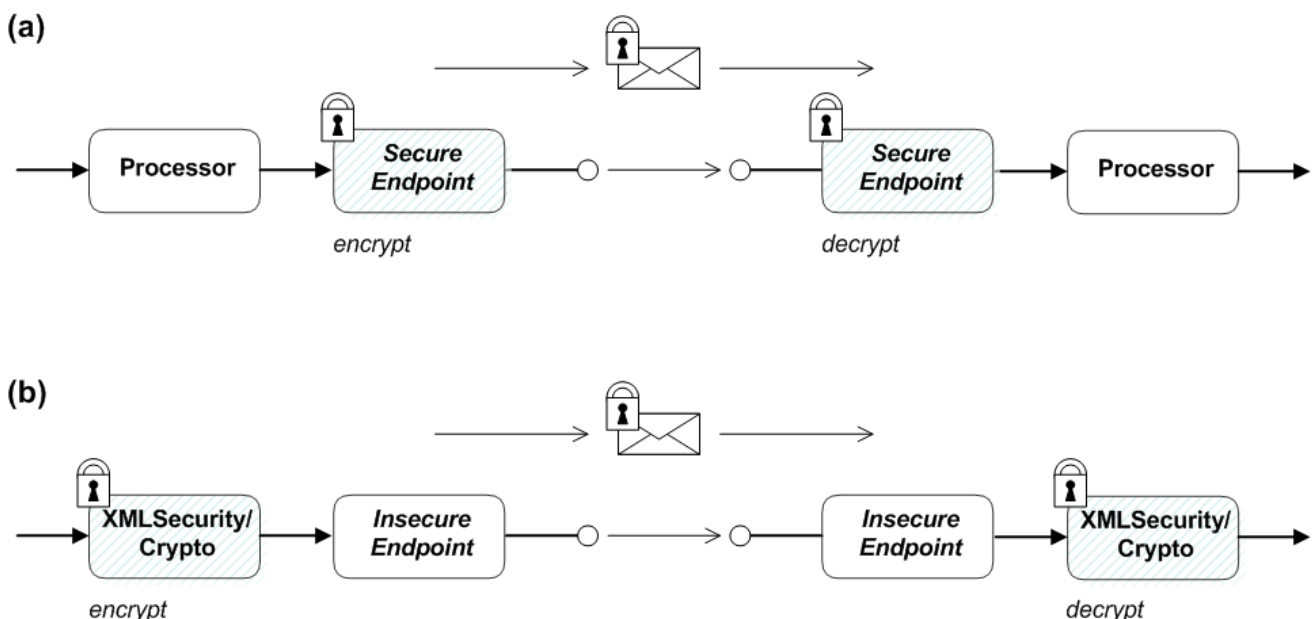
- **PropertiesLoginModule**—stores username/password credentials and user group data in a pair of plain text files.
- **LDAPLoginModule**—an adapter that enables you to store username/password credentials and group data in an LDAP database.
- **GuestLoginModule**—logs all users into a default guest account. This login module is usually used in combination with a preceding login module (defined in the same login entry), where the guest login module is activated only when the preceding login attempt has failed.
- **TextFileCertificateLoginModule**—tests the X.509 certificate received from the client by comparing the Distinguished Name (DN) embedded in the client certificate with the list of DNs stored in a plain text file.

## 1.3. APACHE CAMEL SECURITY

### Overview

Figure 1.3, “[Apache Camel Security Architecture](#)” shows an overview of the basic options for securely routing messages between Apache Camel endpoints.

Figure 1.3. Apache Camel Security Architecture



## Alternatives for Apache Camel security

As shown in [Figure 1.3, “Apache Camel Security Architecture”](#), you have the following options for securing messages:

- *Endpoint security*—part (a) shows a message sent between two routes with secure endpoints. The producer endpoint on the left opens a secure connection (typically using SSL/TLS) to the consumer endpoint on the right. Both of the endpoints support security in this scenario.

With endpoint security, it is typically possible to perform some form of peer authentication (and sometimes authorization).

- *Payload security*—part (b) shows a message sent between two routes where the endpoints are both *insecure*. To protect the message from unauthorized snooping in this case, use a *payload processor* that encrypts the message before sending and decrypts the message after it is received.

A limitation of payload security is that it does *not* provide any kind of authentication or authorization mechanisms.

## Endpoint security

There are several Camel components that support security features. It is important to note, however, that these security features are implemented by the individual components, *not* by the Camel core. Hence, the kinds of security feature that are supported, and the details of their implementation, vary from component to component. Some of the Camel components that currently support security are, as follows:

- JMS and ActiveMQ—SSL/TLS security and JAAS security for client-to-broker and broker-to-broker communication.
- Jetty—HTTP Basic Authentication and SSL/TLS security.
- CXF—SSL/TLS security and WS-Security.
- Crypto—creates and verifies digital signatures in order to guarantee message integrity.
- Netty—SSL/TLS security.
- MINA—SSL/TLS security.
- Cometd—SSL/TLS security.
- glogin and gauth—authorization in the context of Google applications.

## Payload security

Apache Camel provides the following payload security implementations, where the encryption and decryption steps are exposed as data formats on the `marshal()` and `unmarshal()` operations

- [the section called “XMLSecurity data format”](#).
- [the section called “Crypto data format”](#).

## XMLSecurity data format

The XMLSecurity data format is specifically designed to encrypt XML payloads. When using this data format, you can specify which XML element to encrypt. The default behavior is to encrypt *all* XML elements. This feature uses a symmetric encryption algorithm.

For more details, see <http://camel.apache.org/xmlsecurity-dataformat.html>.

## **Crypto data format**

The crypto data format is a general purpose encryption feature that can encrypt any kind of payload. It is based on the Java Cryptographic Extension and it uses asymmetric encryption.

For more details, see <http://camel.apache.org/crypto.html>.

# CHAPTER 2. SECURING THE RED HAT JBOSS FUSE CONTAINER

## Abstract

The Red Hat JBoss Fuse container is secured using JAAS. By defining JAAS realms, you can configure the mechanism used to retrieve user credentials. You can also refine access to the container's administrative interfaces by changing the default roles.

## 2.1. JAAS AUTHENTICATION

### Abstract

The Java Authentication and Authorization Service (JAAS) provides a general framework for implementing authentication in a Java application. The implementation of authentication is modular, with individual JAAS modules (or plug-ins) providing the authentication implementations.

For background information about JAAS, see the [JAAS Reference Guide](#).

### 2.1.1. Default JAAS Realm

#### Overview

This section describes how to manage user data in a for the default JAAS realm in a standalone container.

#### Default JAAS realm

The Red Hat JBoss Fuse container has a predefined JAAS realm, the **karaf** realm, which is used by default to secure all aspects of the container.

#### How to integrate an application with JAAS

You can use the **karaf** realm in your own applications. Simply configure **karaf** as the name of the JAAS realm that you want to use.

#### Default JAAS login modules

When you start JBoss Fuse for the first time, the container is configured as a standalone container and uses the **karaf** default realm. In this default configuration, the **karaf** realm deploys two JAAS login modules, which are enabled simultaneously. To see the deployed login modules, enter the **jaas:realms** console command, as follows:

```
JBossFuse:karaf@root> jaas:realms
Index Realm          Module Class
  1 karaf
org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
  2 karaf
org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
```



## IMPORTANT

In a standalone container, *both* the properties login module and the public key login module are enabled. When JAAS authenticates a user, it tries first of all to authenticate the user with the properties login module. If that fails, it then tries to authenticate the user with the public key login module. If that module also fails, an error is raised.

### Configuring the properties login module

The properties login module is used to store username/password credentials in a flat file format. To create a new user in the properties login module, open the *InstallDir/etc/users.properties* file using a text editor and add a line with the following syntax:

```
Username=Password[,Role1][,Role2]...
```

For example, to create the **jd**oe user with password, **topsecret**, and role, **admin**, you could create an entry like the following:

```
jd=oe=topsecret,admin
```

Where the **admin** role gives full administrative privileges to the **jd**oe user.

### Configuring the public key login module

The public key login module is used to store SSH public key credentials in a flat file format. To create a new user in the public key login module, open the *InstallDir/etc/keys.properties* file using a text editor and add a line with the following syntax:

```
Username=PublicKey,Role1,Role2,...
```

For example, you can create the **jd**oe user with the **admin** role by adding the following entry to the *InstallDir/etc/keys.properties* file (on a single line):

```
jd=oe=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s50f2EbdSP09EAMMeP4C2USZpRV1AI
1H7WT2NWPq/xfW6MPbLm1Vs14E7
gB00b/JmYLdrnVC1pJ+f6AR7ECLCT7up1/63xhv401fnfqimFQ8E+4P208UewwI1VBNaFpEy9n
Xzrith1yrv8iIDGZ3RSAHHAFAAFQCX
YFCPFSMLzLKSuYKi64QL8Fgc9QAAAnEA9+GghdabPd7LvKtcNrhXuXmUr7v60uqC+VdMCz0Hgm
dRWVe0utRZT+ZxBxCBGLRjFnEj6Ewo
Fh03zwkyjMim4TwWeotifI0o4K0uHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hwu
WfBpKLZ16Ae1U1ZAFM0/7PSSoAAACB
AKKSU2PF1/q0LxIwmBZPPIcJshVe7bVUpFvy13BbJDow8rXfSk18w0630zP/qLmcJM0+JbcRU/
53Jj7uyk31drV2qxhIOsLDC9dGCWj4
7Y7TyhPdXh/0dthTRBy6bqGtRPxGa7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYX0+rx,admin
```



## IMPORTANT

Do not insert the entire contents of an **id\_rsa.pub** file here. Insert just the block of symbols which represents the public key itself.

### Encrypting the stored passwords

By default, passwords are stored in the *InstallDir/etc/users.properties* file in plaintext format.

To protect the passwords in this file, you must set the file permissions of the `users.properties` file so that it can be read only by administrators. To provide additional protection, you can optionally encrypt the stored passwords using a message digest algorithm.

To enable the password encryption feature, edit the `installDir/etc/org.apache.karaf.jaas.cfg` file and set the encryption properties as described in the comments. For example, the following settings would enable basic encryption using the MD5 message digest algorithm:

```
encryption.enabled = true
encryption.name = basic
encryption.prefix = {CRYPT}
encryption.suffix = {CRYPT}
encryption.algorithm = MD5
encryption.encoding = hexadecimal
```



## NOTE

The encryption settings in the `org.apache.karaf.jaas.cfg` file are applied *only* to the default `karaf` realm in a standalone container. They have no effect on a fabric container and no effect on a custom realm.

For more details about password encryption, see [Section 2.1.8, “Encrypting Stored Passwords”](#).

## Overriding the default realm

If you want to customise the JAAS realm, the most convenient approach to take is to override the default `karaf` realm by defining a higher ranking `karaf` realm. This ensures that all of the Red Hat JBoss Fuse security components switch to use your custom realm. For details of how to define and deploy custom JAAS realms, see [Section 2.1.2, “Defining JAAS Realms”](#).

## 2.1.2. Defining JAAS Realms

### Overview

When defining a JAAS realm in the OSGi container, you *cannot* put the definitions in a conventional JAAS [login configuration](#) file. Instead, the OSGi container uses a special `jaas:config` element for defining JAAS realms in a blueprint configuration file. The JAAS realms defined in this way are made available to *all* of the application bundles deployed in the container, making it possible to share the JAAS security infrastructure across the whole container.

### Namespace

The `jaas:config` element is defined in the `http://karaf.apache.org/xmlns/jaas/v1.0.0` namespace. When defining a JAAS realm you will need to include the line shown in [Example 2.1, “JAAS Blueprint Namespace”](#).

#### Example 2.1. JAAS Blueprint Namespace

```
xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
```

## Configuring a JAAS realm

The syntax for the `jaas:config` element is shown in [Example 2.2, “Defining a JAAS Realm in Blueprint XML”](#).

### Example 2.2. Defining a JAAS Realm in Blueprint XML

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0">

    <jaas:config name="JaasRealmName"
                [rank="IntegerRank"]>
        <jaas:module className="LoginModuleClassName"
                    [flags="
[required|requisite|sufficient|optional]">
            Property=Value
            ...
        </jaas:module>
        ...
        <!-- Can optionally define multiple modules -->
        ...
    </jaas:config>
</blueprint>
```

The elements are used as follows:

#### `jaas:config`

Defines the JAAS realm. It has the following attributes:

- **name**—specifies the name of the JAAS realm.
- **rank**—specifies an optional rank for resolving naming conflicts between JAAS realms . When two or more JAAS realms are registered under the same name, the OSGi container always picks the realm instance with the highest rank.

#### `jaas:module`

Defines a JAAS login module in the current realm. `jaas:module` has the following attributes:

- **className**—the fully-qualified class name of a JAAS login module. The specified class must be available from the bundle classloader.
- **flags**—determines what happens upon success or failure of the login operation. [Table 2.1, “Flags for Defining a JAAS Module”](#) describes the valid values.

**Table 2.1. Flags for Defining a JAAS Module**

Value	Description
-------	-------------



Value	Description
<b>required</b>	Authentication of this login module must succeed. Always proceed to the next login module in this entry, irrespective of success or failure.
<b>requisite</b>	Authentication of this login module must succeed. If success, proceed to the next login module; if failure, return immediately without processing the remaining login modules.
<b>sufficient</b>	Authentication of this login module is not required to succeed. If success, return immediately without processing the remaining login modules; if failure, proceed to the next login module.
<b>optional</b>	Authentication of this login module is not required to succeed. Always proceed to the next login module in this entry, irrespective of success or failure.

The contents of a **jaas:module** element is a space separated list of property settings, which are used to initialize the JAAS login module instance. The specific properties are determined by the JAAS login module and must be put into the proper format.



#### NOTE

You can define multiple login modules in a realm.

## Converting standard JAAS login properties to XML

Red Hat JBoss Fuse uses the same properties as a standard Java login configuration file, however JBoss Fuse requires that they are specified slightly differently. To see how the JBoss Fuse approach to defining JAAS realms compares with the standard Java login configuration file approach, consider how to convert the login configuration shown in [Example 2.3, “Standard JAAS Properties”](#), which defines the **PropertiesLogin** realm using the Apache ActiveMQ properties login module class, **PropertiesLoginModule**:

### Example 2.3. Standard JAAS Properties

```
PropertiesLogin {
    org.apache.activemq.jaas.PropertiesLoginModule required
        org.apache.activemq.jaas.properties.user="users.properties"
        org.apache.activemq.jaas.properties.group="groups.properties";
};
```

The equivalent JAAS realm definition, using the `jaas:config` element in a blueprint file, is shown in [Example 2.4, "Blueprint JAAS Properties"](#).

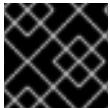
#### Example 2.4. Blueprint JAAS Properties

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"

           xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
           ext/v1.0.0">

    <jaas:config name="PropertiesLogin">
        <jaas:module
            className="org.apache.activemq.jaas.PropertiesLoginModule"
                flags="required">
            org.apache.activemq.jaas.properties.user=users.properties
            org.apache.activemq.jaas.properties.group=groups.properties
        </jaas:module>
    </jaas:config>

</blueprint>
```



#### IMPORTANT

You **do not** use double quotes for JAAS properties in the blueprint configuration.

#### Example

Red Hat JBoss Fuse also provides an adapter that enables you to store JAAS authentication data in an X.500 server. [Example 2.5, "Configuring a JAAS Realm"](#) defines the **LDAPLogin** realm to use JBoss Fuse's **LDAPLoginModule** class, which connects to the LDAP server located at `ldap://localhost:10389`.

#### Example 2.5. Configuring a JAAS Realm

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
           ext/v1.0.0">

    <jaas:config name="LDAPLogin" rank="1">
        <jaas:module
            className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
                flags="required">
            initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
            connection.username=uid=admin,ou=system
            connection.password=secret
            connection.protocol=
            connection.url = ldap://localhost:10389
            user.base.dn = ou=users,ou=system
            user.filter = (uid=%u)
            user.search.subtree = true
            role.base.dn = ou=users,ou=system
```

```

    role.filter = (uid=%u)
    role.name.attribute = ou
    role.search.subtree = true
    authentication = simple
  </jaas:module>
</jaas:config>
</blueprint>

```

For a detailed description and example of using the LDAP login module, see [Section 2.2, “Enabling LDAP Authentication”](#).

### 2.1.3. JAAS Properties Login Module

#### Overview

The JAAS properties login module stores user data in a flat file format (where the stored passwords can optionally be encrypted using a message digest algorithm). The user data can either be edited directly, using a simple text editor, or managed using the `jaas:*` console commands.

For example, a standalone container uses the JAAS properties login module by default and stores the associated user data in the `InstallDir/etc/users.properties` file.

#### Supported credentials

The JAAS properties login module authenticates username/password credentials, returning the list of roles associated with the authenticated user.

#### Implementation classes

The following classes implement the JAAS properties login module:

##### `org.apache.karaf.jaas.modules.properties.PropertiesLoginModule`

Implements the JAAS login module.

##### `org.apache.karaf.jaas.modules.properties.PropertiesBackingEngineFactory`

Must be exposed as an OSGi service. This service makes it possible for you to manage the user data using the `jaas:*` console commands from the Apache Karaf shell (see [chapter “JAAS Console Commands”](#) in [“Console Reference”](#)).

#### Options

The JAAS properties login module supports the following options:

##### `users`

Location of the user properties file.

#### Format of the user properties file

The user properties file is used to store username, password, and role data for the properties login module. Each user is represented by a single line in the user properties file, where a line has the following form:

```
Username=Password[,Role][,Role]...
```

## Sample Blueprint configuration

The following Blueprint configuration shows how to define a new **karaf** realm using the properties login module, where the default **karaf** realm is overridden by setting the **rank** attribute to **2**:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-
cm/v1.1.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0">

    <type-converters>
        <bean
class="org.apache.karaf.jaas.modules.properties.PropertiesConverter"/>
    </type-converters>

    <!-- Allow usage of System properties, especially the karaf.base
property -->
    <ext:property-placeholder placeholder-prefix="$[" placeholder-
suffix="]"/>

    <jaas:config name="karaf" rank="2">
        <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
            flags="required">
            users = $[karaf.base]/etc/users.properties
        </jaas:module>
    </jaas:config>

    <!-- The Backing Engine Factory Service for the PropertiesLoginModule
-->
    <service
interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
        <bean
class="org.apache.karaf.jaas.modules.properties.PropertiesBackingEngineFac
tory"/>
    </service>

</blueprint>
```

Remember to export the **BackingEngineFactory** bean as an OSGi service, so that the **jaas:\*** console commands can manage the user data.

### 2.1.4. JAAS OSGi Config Login Module

#### Overview

The JAAS OSGi config login module leverages the *OSGi Config Admin Service* to store user data. This login module is fairly similar to the JAAS properties login module (for example, the syntax of the user entries is the same), but the mechanism for retrieving user data is based on the OSGi Config Admin Service.

The user data can be edited directly by creating a corresponding OSGi configuration file, **etc/PersistentID.cfg** or using any method of configuration that is supported by the OSGi Config Admin Service. The **jaas:\*** console commands are not supported, however.

## Supported credentials

The JAAS OSGi config login module authenticates username/password credentials, returning the list of roles associated with the authenticated user.

## Implementation classes

The following classes implement the JAAS OSGi config login module:

### **org.apache.karaf.jaas.modules.osgi.OsgiConfigLoginModule**

Implements the JAAS login module.



## NOTE

There is no backing engine factory for the OSGi config login module, which means that this module cannot be managed using the **jaas:\*** console commands.

## Options

The JAAS OSGi config login module supports the following options:

### **pid**

The *persistent ID* of the OSGi configuration containing the user data. In the OSGi Config Admin standard, a persistent ID references a set of related configuration properties.

## Location of the configuration file

The location of the configuration file follows the usual convention where the configuration for the persistent ID, **PersistentID**, is stored in the following file:

```
InstallDir/etc/PersistentID.cfg
```

## Format of the configuration file

The **PersistentID.cfg** configuration file is used to store username, password, and role data for the OSGi config login module. Each user is represented by a single line in the configuration file, where a line has the following form:

```
Username=Password[,Role][,Role]...
```

This is the same format that is used in a users property file.

## Sample Blueprint configuration

The following Blueprint configuration shows how to define a new **karaf** realm using the OSGi config login module, where the default **karaf** realm is overridden by setting the **rank** attribute to **2**:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-
cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0">

  <jaas:config name="karaf" rank="2">
    <jaas:module
className="org.apache.karaf.jaas.modules.osgi.OsgiConfigLoginModule"
      flags="required">
      pid = org.jboss.example.osgiconfigloginmodule
    </jaas:module>
  </jaas:config>

</blueprint>
```

In this example, the user data will be stored in the file, ***InstallDir/etc/org.jboss.example.osgiconfigloginmodule.cfg***, and it is not possible to edit the configuration using the **jaas: \*** console commands.

### 2.1.5. JAAS Public Key Login Module

#### Overview

The JAAS public key login module stores user data in a flat file format, which can be edited directly using a simple text editor. The **jaas: \*** console commands are not supported, however.

For example, a standalone container uses the JAAS public key login module by default and stores the associated user data in the ***InstallDir/etc/keys.properties*** file.

#### Supported credentials

The JAAS public key login module authenticates SSH key credentials. When a user tries to log in, the SSH protocol uses the stored public key to challenge the user. The user must possess the corresponding private key in order to answer the challenge. If login is successful, the login module returns the list of roles associated with the user.

#### Implementation classes

The following classes implement the JAAS public key login module:

##### **org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule**

Implements the JAAS login module.



## NOTE

There is no backing engine factory for the public key login module, which means that this module cannot be managed using the `jaas:*` console commands.

## Options

The JAAS public key login module supports the following options:

### users

Location of the user properties file for the public key login module.

## Format of the user properties file

The user properties file is used to store username, public key, and role data for the public key login module. Each user is represented by a single line in the user properties file, where a line has the following form:

```
Username=PublicKey[,Role][,Role]...
```

Where the *PublicKey* is the public key part of an SSH key pair (typically found in a user's home directory in `~/.ssh/id_rsa.pub` in a UNIX system).

For example, to create the user `jdoe` with the `admin` role, you would create an entry like the following:

```
jdoe=AAAAB3NzaC1kc3MAAACBAP1/U4EddRIpUt9KnC7s50f2EbdSP09EAMMeP4C2USZpRV1AI
1H7WT2NWPq/xfw6MPbLm1Vs14E7
gB00b/JmYldrmVC1pJ+f6AR7ECLCT7up1/63xhv401fnfqimFQ8E+4P208UewwI1VBNaFpEy9n
Xzrith1yrv8iIDGZ3RSAHHAAAAFQCX
YFCPFSMLzLKSuYKi64QL8Fgc9QAAAEAnEA9+GghdabPd7LvKtcNrhXuXmUr7v60uqC+VdMCz0Hgm
dRWVe0utRZT+ZxBxCBgLRJFnEj6Ewo
Fh03zwkyjMim4TwWeotifI0o4K0uHiuzpnWRbqN/C/ohNWLx+2J6ASQ7zKTxvqhRkImog9/hWu
WfBpKLZl6Ae1U1ZAFM0/7PSSoAAACB
AKKSU2PF1/q0LxIwmBZPPIcJshVe7bVUpFvy13BbJDow8rXfSk18w0630zP/qLmcJM0+JbcRU/
53Jj7uyk31drV2qxhI0sLDC9dGCWj4
7Y7TyhPdXh/0dthTRBy6bqGtRPxGa7gJov1xm/UuYYXPIUR/3x9MAZvZ5xvE0kYX0+rx, admin
```



## IMPORTANT

Do not insert the entire contents of the `id_rsa.pub` file here. Insert just the block of symbols which represents the public key itself.

## Sample Blueprint configuration

The following Blueprint configuration shows how to define a new `karaf` realm using the public key login module, where the default `karaf` realm is overridden by setting the `rank` attribute to `2`:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-
cm/v1.1.0"
```

```

        xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0">

        <!-- Allow usage of System properties, especially the karaf.base
property -->
        <ext:property-placeholder placeholder-prefix="$[" placeholder-
suffix="]"/>

        <jaas:config name="karaf" rank="2">
            <jaas:module
className="org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule"
                flags="required">
                users = $[karaf.base]/etc/keys.properties
            </jaas:module>
        </jaas:config>

</blueprint>

```

In this example, the user data will be stored in the file, *InstallDir/etc/keys.properties*, and it is not possible to edit the configuration using the `jaas:*` console commands.

## 2.1.6. JAAS JDBC Login Module

### Overview

The JAAS JDBC login module enables you to store user data in a database back-end, using Java Database Connectivity (JDBC) to connect to the database. Hence, you can use any database that supports JDBC to store your user data. To manage the user data, you can use either the native database client tools or the `jaas:*` console commands (where the backing engine uses configured SQL queries to perform the relevant database updates).

### Supported credentials

The JAAS JDBC Login Module authenticates username/password credentials, returning the list of roles associated with the authenticated user.

### Implementation classes

The following classes implement the JAAS JDBC Login Module:

#### `org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule`

Implements the JAAS login module.

#### `org.apache.karaf.jaas.modules.jdbc.JDBCBackingEngineFactory`

Must be exposed as an OSGi service. This service makes it possible for you to manage the user data using the `jaas:*` console commands from the Apache Karaf shell (see [chapter "JAAS Console Commands" in "Console Reference"](#)).

### Options

The JAAS JDBC login module supports the following options:



## datasource

The JDBC data source, specified either as an OSGi service or as a JNDI name. You can specify a data source's OSGi service using the following syntax:

```
osgi:ServiceInterfaceName[/ServicePropertiesFilter]
```

The *ServiceInterfaceName* is the interface or class that is exported by the data source's OSGi service (usually `javax.sql.DataSource`).

Because multiple data sources can be exported as OSGi services in a container, it is usually necessary to specify a filter, *ServicePropertiesFilter*, to select the particular data source that you want. Filters on OSGi services are applied to the service property settings and follow a syntax that is borrowed from LDAP filter syntax.

## query.password

The SQL query that retrieves the user's password. The query can contain a single question mark character, `?`, which is substituted by the username at run time.

## query.role

The SQL query that retrieves the user's roles. The query can contain a single question mark character, `?`, which is substituted by the username at run time.

## insert.user

The SQL query that creates a new user entry. The query can contain two question marks, `?`, characters: the first question mark is substituted by the username and the second question mark is substituted by the password at run time.

## insert.role

The SQL query that adds a role to a user entry. The query can contain two question marks, `?`, characters: the first question mark is substituted by the username and the second question mark is substituted by the role at run time.

## delete.user

The SQL query that deletes a user entry. The query can contain a single question mark character, `?`, which is substituted by the username at run time.

## delete.role

The SQL query that deletes a role from a user entry. The query can contain two question marks, `?`, characters: the first question mark is substituted by the username and the second question mark is substituted by the role at run time.

## delete.roles

The SQL query that deletes multiple roles from a user entry. The query can contain a single question mark character, `?`, which is substituted by the username at run time.

## Example of setting up a JDBC login module

To set up a JDBC login module, perform the following main steps:

1. the section called “Create the database tables”
2. the section called “Create the data source”
3. the section called “Specify the data source as an OSGi service”

## Create the database tables

Before you can set up the JDBC login module, you must set up a users table and a roles table in the backing database to store the user data. For example, the following SQL commands show how to create a suitable **users** table and **roles** table:

```
CREATE TABLE users (  
    username varchar(255) NOT NULL,  
    password varchar(255) NOT NULL,  
    PRIMARY KEY (username)  
);  
CREATE TABLE roles (  
    username varchar(255) NOT NULL,  
    role varchar(255) NOT NULL,  
    PRIMARY KEY (username, role)  
);
```

The **users** table stores username/password data and the **roles** table associates a username with one or more roles.

## Create the data source

To use a JDBC datasource with the JDBC login module, the correct approach to take is to create a data source instance and export the data source as an OSGi service. The JDBC login module can then access the data source by referencing the exported OSGi service. For example, you could create a MySQL data source instance and expose it as an OSGi service (of **javax.sql.DataSource** type) using code like the following in a Blueprint file:

```
<blueprint xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">  
    <bean class="com.mysql.jdbc.jdbc2.optional.MysqlDataSource"  
id="mysqlDatasource">  
        <property name="serverName" value="localhost"></property>  
        <property name="databaseName" value="DBName"></property>  
        <property name="port" value="3306"></property>  
        <property name="user" value="DBUser"></property>  
        <property name="password" value="DBPassword"></property>  
    </bean>  
  
    <service id="mysqlDS" interface="javax.sql.DataSource"  
ref="mysqlDatasource">  
        <service-properties>  
            <entry key="osgi.jndi.service.name" value="jdbc/karafdb"/>  
        </service-properties>  
    </service>  
</blueprint>
```

The preceding Blueprint configuration should be packaged and installed in the container as an OSGi bundle.

## Specify the data source as an OSGi service

After the data source has been instantiated and exported as an OSGi service, you are ready to configure the JDBC login module. In particular, the **datasource** option of the JDBC login module can reference the data source's OSGi service using the following syntax:

```
osgi:javax.sql.DataSource/(osgi.jndi.service.name=jdbc/karafdb)
```

Where **javax.sql.DataSource** is the interface type of the exported OSGi service and the filter, **(osgi.jndi.service.name=jdbc/karafdb)**, selects the particular **javax.sql.DataSource** instance whose **osgi.jndi.service.name** service property has the value, **jdbc/karafdb**.

For example, you can use the following Blueprint configuration to override the **karaf** realm with a JDBC login module that references the sample MySQL data source:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-
cm/v1.1.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0">

    <!-- Allow usage of System properties, especially the karaf.base
property -->
    <ext:property-placeholder placeholder-prefix="$[" placeholder-
suffix="]"/>

    <jaas:config name="karaf" rank="2">
        <jaas:module
className="org.apache.karaf.jaas.modules.jdbc.JDBCLoginModule"
            flags="required">
            datasource =
osgi:javax.sql.DataSource/(osgi.jndi.service.name=jdbc/karafdb)
            query.password = SELECT PASSWORD FROM USERS WHERE USERNAME=?
            query.role = SELECT ROLE FROM ROLES WHERE USERNAME=?
            insert.user = INSERT INTO USERS VALUES(?,?)
            insert.role = INSERT INTO ROLES VALUES(?,?)
            delete.user = DELETE FROM USERS WHERE USERNAME=?
            delete.role = DELETE FROM ROLES WHERE USERNAME=? AND ROLE=?
            delete.roles = DELETE FROM ROLES WHERE USERNAME=?
        </jaas:module>
    </jaas:config>

    <!-- The Backing Engine Factory Service for the JDBCLoginModule -->
    <service
interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
        <bean
class="org.apache.karaf.jaas.modules.jdbc.JDBCBackingEngineFactory"/>
    </service>

</blueprint>
```

**NOTE**

The SQL statements shown in the preceding configuration are in fact the default values of these options. Hence, if you create user and role tables consistent with these SQL statements, you could omit the options settings and rely on the defaults.

In addition to creating a `JDBCLoginModule`, the preceding Blueprint configuration also instantiates and exports a `JDBCBackingEngineFactory` instance, which enables you to manage the user data using the `jaas:*` console commands.

## 2.1.7. JAAS LDAP Login Module

### Overview

The JAAS LDAP login module enables you to store user data in an LDAP database. To manage the stored user data, use a standard LDAP client tool. The `jaas:*` console commands are *not* supported.

For more details about using LDAP with Red Hat JBoss Fuse, see [Chapter 7, LDAP Authentication Tutorial](#).

### Supported credentials

The JAAS LDAP Login Module authenticates username/password credentials, returning the list of roles associated with the authenticated user.

### Implementation classes

The following classes implement the JAAS LDAP Login Module:

#### `org.apache.karaf.jaas.modules.ldap.LDAPLoginModule`

Implements the JAAS login module.

**NOTE**

There is no backing engine factory for the LDAP Login Module, which means that this module cannot be managed using the `jaas:*` console commands.

### Options

The JAAS LDAP login module supports the following options:

#### `connection.url`

The LDAP connection URL—for example, `ldap://hostname`.

#### `connection.username`

Admin username to connect to the LDAP server. This parameter is optional: if it is not provided, the LDAP connection will be anonymous.

#### `connection.password`

Admin password to connect to the LDAP server. Used only if the `connection.username` is also specified.

**user.base.dn**

The LDAP base DN used to look up roles—for example, `ou=role,dc=apache,dc=org`.

**user.filter**

The LDAP filter used to look up a user's role—for example, `(member:=uid=%u)`.

**user.search.subtree**

If `true`, the user lookup is recursive (**SUBTREE**). If `false`, the user lookup is performed only at the first level (**ONELEVEL**).

**role.base.dn**

The LDAP base DN used to look up roles—for example, `ou=role,dc=apache,dc=org`.

**role.filter**

The LDAP filter used to look up a user's role—for example, `(member:=uid=%u)`. It is also possible to use the placeholder `%dn`, which gets replaced by the DN of the user's LDAP entry at run time.

**role.name.attribute**

The LDAP role attribute containing the role value used by Apache Karaf—for example, `cn`.

**role.search.subtree**

If `true`, the role lookup is recursive (**SUBTREE**). If `false`, the role lookup is performed only at the first level (**ONELEVEL**).

**authentication**

Define the authentication back-end used on the LDAP server. The default is `simple`.

**initial.context.factory**

Define the initial context factory used to connect to the LDAP server. The default is `com.sun.jndi.ldap.LdapCtxFactory`.

**ssl**

If `true` or if the protocol on the `connection.url` is `ldaps`, an SSL connection will be used.

**ssl.provider**

Specifies the SSL provider.

**ssl.protocol**

The protocol version to use. You *must* set this property to `TLSv1`, in order to prevent the SSLv3 protocol from being used (POODLE vulnerability).

**ssl.algorithm**

The algorithm to use for the **KeyManagerFactory** and the **TrustManagerFactory**—for example, **PKIX**.

### ssl.keystore

The ID of the keystore that stores the LDAP client's own X.509 certificate (required only if SSL client authentication is enabled on the LDAP server). The keystore must be deployed using a **jaas:keystore** element (see [the section called "Sample Blueprint configuration"](#)).

### ssl.keyalias

The keystore alias of the LDAP client's own X.509 certificate (required only if there is more than one certificate stored in the keystore specified by **ssl.keystore**).

### ssl.truststore

The ID of the keystore that stores trusted CA certificates, which are used to verify the LDAP server's certificate (the LDAP server's certificate chain must be signed by one of the certificates in the truststore). The keystore must be deployed using a **jaas:keystore** element.

## Sample Blueprint configuration

The following Blueprint configuration shows how to define a new **karaf** realm using the LDAP login module, where the default **karaf** realm is overridden by setting the **rank** attribute to **2**:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-
cm/v1.1.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0">

  <!-- Allow usage of System properties, for example the karaf.home
property -->
  <ext:property-placeholder placeholder-prefix="${" placeholder-
suffix="}"/>

  <jaas:config name="karaf" rank="2">
    <jaas:module
className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
      flags="required">
      connection.url = ldaps://localhost:10636
      user.base.dn = ou=users,ou=system
      user.filter = (uid=%u)
      user.search.subtree = true
      role.base.dn = ou=groups,ou=system
      role.filter = (uniqueMember=uid=%u)
      role.name.attribute = cn
      role.search.subtree = true
      authentication = simple
      ssl.protocol=TLSv1
      ssl.truststore=ks
      ssl.algorithm=PKIX
    </jaas:module>
```

```

</jaas:config>

<jaas:keystore name="ks"
  path="file:///${karaf.home}/etc/trusted.ks"
  keystorePassword="secret" />

</blueprint>

```



### IMPORTANT

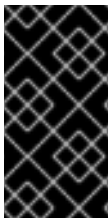
You must set **ssl.protocol** to **TLSv1**, in order to protect against the [Poodle vulnerability \(CVE-2014-3566\)](#)

## 2.1.8. Encrypting Stored Passwords

### Overview

By default, the JAAS login modules store passwords in plaintext format. Although you can (and should) protect such data by setting file permissions appropriately, you can provide additional protection to passwords by storing them in an obscured format (using a *message digest* algorithm).

Red Hat JBoss Fuse provides a set of options for enabling password encryption, which can be combined with *any* of the JAAS login modules (except for the public key login module, where it is not needed).



### IMPORTANT

Although message digest algorithms are not easy to crack, they are not invulnerable to attack (for example, see the [Wikipedia article on cryptographic hash functions](#)). Always use file permissions to protect files containing passwords, in addition to using password encryption.

### Options

Password encryption for JAAS login modules can optionally be enabled by setting the following login module properties:

#### **encryption.enabled**

Set to **true**, to enable password encryption.

#### **encryption.name**

Name of the encryption service, which has been registered as an OSGi service.

#### **encryption.prefix**

Prefix for encrypted passwords.

#### **encryption.suffix**

Suffix for encrypted passwords.

#### **encryption.algorithm**

Specifies the name of the encryption algorithm—for example, **MD5** or **SHA-1**. You can specify one of the following encryption algorithms:

- MD2
- MD5
- SHA-1
- SHA-256
- SHA-384
- SHA-512

**encryption.encoding**

Encrypted passwords encoding: **hexadecimal** or **base64**.

**encryption.providerName** (*Jasypt only*)

Name of the `java.security.Provider` instance that is to provide the digest algorithm.

**encryption.providerClassName** (*Jasypt only*)

Class name of the security provider that is to provide the digest algorithm

**encryption.iterations** (*Jasypt only*)

Number of times to apply the hash function recursively.

**encryption.saltSizeBytes** (*Jasypt only*)

Size of the salt used to compute the digest.

**encryption.saltGeneratorClassName** (*Jasypt only*)

Class name of the salt generator.

**role.policy**

Specifies the policy for identifying role principals. Can have the values, **prefix** or **group**.

**role.discriminator**

Specifies the discriminator value to be used by the role policy.

## Encryption services

An encryption service can be defined by inheriting from the `org.apache.karaf.jaas.modules.EncryptionService` interface and exporting an instance of the encryption service as an OSGi service. Two alternative implementations of the encryption service are provided:

- [the section called “Basic encryption service”](#).
- [the section called “Jasypt encryption”](#).

### Basic encryption service



The basic encryption service is installed in the standalone container by default and you can reference it by setting the **encryption.name** property to the value, **basic**. In the basic encryption service, the message digest algorithms are provided by the [SUN](#) security provider (the default security provider in the Oracle JDK).

## Jasypt encryption

The Jasypt encryption service can be installed in the standalone container by installing the **jasypt-encryption** feature. For example, you can install Jasypt encryption by entering the following console command:

```
JBossFuse:karaf@root> features:install jasypt-encryption
```

This command installs the requisite Jasypt bundles and exports Jasypt encryption as an OSGi service, so that it is available for use by JAAS login modules. To access the Jasypt encryption service, set the **encryption.name** property to the value, **jasypt**.

For more information about Jasypt encryption, see the [Jasypt documentation](#).

## Example of a login module with Jasypt encryption

Assuming that you have already installed the **jasypt-encryption** feature, you could deploy a properties login module with Jasypt encryption using the following Blueprint configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
           xmlns:cm="http://aries.apache.org/blueprint/xmlns/blueprint-
cm/v1.1.0"
           xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0">

    <type-converters>
        <bean
class="org.apache.karaf.jaas.modules.properties.PropertiesConverter"/>
    </type-converters>

    <!-- Allow usage of System properties, especially the karaf.base
property -->
    <ext:property-placeholder placeholder-prefix="$[" placeholder-
suffix="]"/>

    <jaas:config name="karaf" rank="2">
        <jaas:module
className="org.apache.karaf.jaas.modules.properties.PropertiesLoginModule"
           flags="required">
            users = $[karaf.base]/etc/users.properties
            encryption.enabled = true
            encryption.name = jasypt
            encryption.algorithm = SHA-256
            encryption.encoding = base64
            encryption.iterations = 100000
            encryption.saltSizeBytes = 16
        </jaas:module>
    </jaas:config>
```

```

    <!-- The Backing Engine Factory Service for the PropertiesLoginModule
-->
    <service
interface="org.apache.karaf.jaas.modules.BackingEngineFactory">
        <bean
class="org.apache.karaf.jaas.modules.properties.PropertiesBackingEngineFac
tory"/>
        </service>
</blueprint>

```

## 2.2. ENABLING LDAP AUTHENTICATION

### Overview

Red Hat JBoss Fuse supplies a JAAS login module that enables it to use LDAP to authenticate users. The JBoss Fuse JAAS LDAP login module is implemented by the **org.apache.karaf.jaas.modules.ldap.LDAPLoginModule** class. It is preloaded in the container, so you do not need to install its bundle.

### Procedure

To enable JBoss Fuse to use LDAP for user authentication you need to create a JAAS realm that includes the JBoss Fuse LDAP login module. As shown in [Example 2.6, “Red Hat JBoss Fuse LDAP JAAS Login Module”](#), this is done by adding a **jaas:module** element to the realm and setting its **className** attribute to **org.apache.karaf.jaas.modules.ldap.LDAPLoginModule**.

#### Example 2.6. Red Hat JBoss Fuse LDAP JAAS Login Module

```

<jaas:config ... >
  <jaas:module
className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
          flags="required">
    ...
  </jaas:module>
</jaas:config>

```

You will also need to provide values for the properties described in [Table 2.2, “Properties for the Red Hat JBoss Fuse LDAP Login Module”](#).

### LDAP properties

[Table 2.2, “Properties for the Red Hat JBoss Fuse LDAP Login Module”](#) describes the properties used to configure the JBoss Fuse JAAS LDAP login module.

**Table 2.2. Properties for the Red Hat JBoss Fuse LDAP Login Module**

Property	Description
----------	-------------

Property	Description
connection.url	Specifies specify the location of the directory server using an ldap URL, <code>ldap://Host:Port</code> . You can optionally qualify this URL, by adding a forward slash, <code>/</code> , followed by the DN of a particular node in the directory tree.
connection.username	Specifies the DN of the user that opens the connection to the directory server. For example, <b>uid=admin,ou=system</b> .
connection.password	Specifies the password that matches the DN from connection.username. In the directory server, the password is normally stored as a <b>userPassword</b> attribute in the corresponding directory entry.
user.base.dn	Specifies the DN of the subtree of the DIT to search for user entries.
user.filter	Specifies the LDAP search filter used to locate user credentials. It is applied to the subtree selected by user.base.dn. Before being passed to the LDAP search operation, the value is subjected to string substitution such that all occurrences of <b>%u</b> are replaced by the user name extracted from the incoming credentials.
user.search.subtree	Specifies if the user entry search's scope includes the subtrees of the tree selected by user.base.dn.
role.base.dn	Specifies the DN of the subtree of the DIT to search for role entries.
role.filter	<p>Specifies the LDAP search filter used to locate roles. It is applied to the subtree selected by role.base.dn. Before being passed to the LDAP search operation, the value is subjected to string substitution, as follows:</p> <ul style="list-style-type: none"> <li>• <b>%u</b> is replaced by the user name extracted from the incoming credentials, and</li> <li>• <b>%dn</b> is replaced by the DN of the corresponding user in the LDAP server (which was found by matching against the <b>user.filter</b> filter).</li> </ul>
role.name.attribute	Specifies the attribute type of the role entry that contains the name of the role/group. If you omit this option, the role search feature is effectively disabled.

Property	Description
role.search.subtree	Specifies if the role entry search's scope includes the subtrees of the tree selected by role.base.dn.
authentication	Specifies the authentication method used when binding to the LDAP server. Valid values are <ul style="list-style-type: none"> <li>• <b>simple</b>—bind with user name and password authentication</li> <li>• <b>none</b>—bind anonymously</li> </ul>
initial.context.factory	Specifies the class of the context factory used to connect to the LDAP server. This must always be set to <b>com.sun.jndi.ldap.LdapCtxFactory</b> .
ssl	Specifies if the connection to the LDAP server is secured via SSL. If connection.url starts with ldaps:// SSL is used regardless of this property.
ssl.provider	Specifies the SSL provider to use for the LDAP connection. If not specified, the default SSL provider is used.
ssl.protocol	Specifies the protocol to use for the SSL connection. You <i>must</i> set this property to <b>TLSv1</b> , in order to prevent the SSLv3 protocol from being used (POODLE vulnerability).
ssl.algorithm	Specifies the algorithm used by the trust store manager.
ssl.keystore	Specifies the keystore name.
ssl.keyalias	Specifies the name of the private key in the keystore.
ssl.truststore	Specifies the trust keystore name.

All of the properties are mandatory except the SSL properties.

## Example

[Example 2.7, “Configuring a JAAS Realm that Uses LDAP Authentication”](#) defines a JAAS realm that uses the LDAP server located at ldap://localhost:10389.

### Example 2.7. Configuring a JAAS Realm that Uses LDAP Authentication

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0">

  <jaas:config name="karaf" rank="1">
    <jaas:module
      className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
        flags="sufficient">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      connection.username=uid=admin,ou=system
      connection.password=secret
      connection.protocol=
      connection.url = ldaps://localhost:10636
      user.base.dn = ou=users,ou=system
      user.filter = (uid=%u)
      user.search.subtree = true
      role.base.dn = ou=roles,ou=system,dc=jbossfuse
      role.filter = (uid=%u)
      role.name.attribute = cn
      role.search.subtree = true
      authentication = simple
      ssl.protocol=TLSv1
      ssl.truststore=truststore
      ssl.algorithm=PKIX
    </jaas:module>
    ...
  </jaas:config>
</blueprint>

```



### IMPORTANT

You must set **ssl.protocol** to **TLSv1**, in order to protect against the [Poodle vulnerability \(CVE-2014-3566\)](#)

## 2.3. ENABLING REMOTE JMX SSL

### Overview

Red Hat JBoss Fuse provides a JMX port that allows remote monitoring and management of Fuse containers via MBeans. To prevent unauthorized access to data and applications, you need to secure JMX communications by configuring JMX over SSL.

Configuring JMX over SSL access, involves:

- [Creating the `jbossweb.keystore`](#)
- [Creating the `keystore.xml` file](#)
- [Ensuring that the `keystore.xml` file is loaded at the appropriate time](#)
- [Adding the necessary properties to `org.apache.karaf.management.cfg`](#)

After you have configured JMX over SSL access, you should test the connection.



### WARNING

If you are planning to enable SSL/TLS security, you must ensure that you explicitly disable the SSLv3 protocol, in order to safeguard against the [Poodle vulnerability \(CVE-2014-3566\)](#). For more details, see [Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x](#).



### NOTE

If you configure JMX over SSL while Red Hat JBoss Fuse is running, you will need to restart it.

## Prerequisites

If you haven't already done so, you need to:

- Set your **JAVA\_HOME** environment variable
- Configure a JBoss Fuse user with the **admin** role

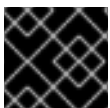
Edit the `<installDir>/jboss-fuse-6.1.0.redhat-379/etc/users.properties` file to enable the default **admin** user by uncommenting the line `#admin=admin,admin`. Alternatively, you can define another user with the **admin** role.

- `cd` to `<installDir>/jboss-fuse-6.1.0.redhat-379/etc`.

## Create the `jbossweb.keystore`

At the command line, using `-dname` values appropriate for your application, type this command:

```
$JAVA_HOME/bin/keytool -genkey -v -alias jbossalias -keyalg RSA -keysize
1024 -keystore
jbossweb.keystore -validity 3650 -keypass JbossPassword -storepass
JbossPassword -dname "CN=127.0.0.1, OU=RedHat Software Unit, O=RedHat,
L=Boston, S=Mass, C=USA"
```



### IMPORTANT

Type the entire command on the same command line.

The command returns output that looks like this:

```
Generating 1,024 bit RSA key pair and self-signed certificate
(SHA256withRSA) with a validity of 3,650 days
for: CN=127.0.0.1, OU=RedHat Software Unit, O=RedHat, L=Boston, ST=Mass,
C=USA
```

```

New certificate (self-signed):
[
[
  Version: V3
  Subject: CN=127.0.0.1, OU=RedHat Software Unit, O=RedHat, L=Boston,
ST=Mass, C=USA
  Signature Algorithm: SHA256withRSA, OID = 1.2.840.113549.1.1.11

  Key:  Sun RSA public key, 1024 bits
  modulus:
1123086025790567043604962990501918169461098372864273201795342440080393808

15941007760750086474599109914138063728007229476701664078149017544591007202
79046

39446218137381773240310642603826594831938261774487620304376693183910726198
67218

03697233521083906272245608532830105836205236924847365988048833871135195983
5357
  public exponent: 65537
  Validity: [From: Thu Jun 05 12:19:52 EDT 2014,
              To: Sun Jun 02 12:19:52 EDT 2024]
  Issuer: CN=127.0.0.1, OU=RedHat Software Unit, O=RedHat, L=Boston,
ST=Mass, C=USA
  SerialNumber: [ 4666e4e6]

Certificate Extensions: 1
[1]: ObjectID: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: AC 44 A5 F2 E6 2F B2 5A   5F 88 FE 69 60 B4 27 7D   .D.../.Z_..i`.'.
0010: B9 81 23 9C                               ..#.
]
]
]
  Algorithm: [SHA256withRSA]
  Signature:
0000: 01 1D 95 C0 F2 03 B0 FD   CF 3A 1A 14 F5 2E 04 E5   .....:.....
0010: DD 18 DD 0E 24 60 00 54   35 AE FE 36 7B 38 69 4C   ....$`.T5..6.8iL
0020: 1E 85 0A AF AE 24 1B 40   62 C9 F4 E5 A9 02 CD D3   ....$.@b.....
0030: 91 57 60 F6 EF D6 A4 84   56 BA 5D 21 11 F7 EA 09   .W`.....V.].!....
0040: 73 D5 6B 48 4A A9 09 93   8C 05 58 91 6C D0 53 81   s.kHJ.....X.l.S.
0050: 39 D8 29 59 73 C4 61 BE   99 13 12 89 00 1C F8 38   9.)Ys.a.....8
0060: E2 BF D5 3C 87 F6 3F FA   E1 75 69 DF 37 8E 37 B5   ...<..?..ui.7.7.
0070: B7 8D 10 CC 9E 70 E8 6D   C2 1A 90 FF 3C 91 84 50   .....p.m....<..P
]
[Storing jbossweb.keystore]

```

Check whether `<installDir>/jboss-fuse-6.1.0.redhat-379/etc` now contains the file `jbossweb.keystore`.

## Create the keystore.xml file

1. Using your favorite xml editor, create and save the **keystore.xml** file in the `<installDir>/jboss-fuse-6.1.0.redhat-379/etc` directory.
2. Include this text in the file:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0">
  <jaas:keystore name="sample_keystore"
    rank="1"
    path="file:etc/jbossweb.keystore"
    keystorePassword="JbossPassword"
    keyPasswords="jbossalias=JbossPassword" />
</blueprint>
```

Check whether `<installDir>/jboss-fuse-6.1.0.redhat-379/etc` now contains the file **keystore.xml**.

## Ensure that the keystore.xml file is loaded at the appropriate time

Create the **org.apache.felix.fileinstall-keystore.cfg** file.

1. Using your favorite text editor, create and save the **org.apache.felix.fileinstall-keystore.cfg** file in the `<installDir>/jboss-fuse-6.1.0.redhat-379/etc` directory.
2. Include these properties in the file:

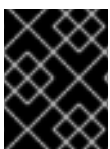
```
felix.fileinstall.dir      = ${karaf.base}/etc
felix.fileinstall.filter   = keystore\\.xml
felix.fileinstall.poll     = 1000
felix.fileinstall.noInitialDelay = true
felix.fileinstall.log.level = 3
felix.fileinstall.start.level = 25
```

Check whether `<installDir>/jboss-fuse-6.1.0.redhat-379/etc` now contains the file **org.apache.felix.fileinstall-keystore.cfg**.

## Add the required properties to org.apache.karaf.management.cfg

Edit the `<installDir>/jboss-fuse-6.1.0.redhat-379/etc/org.apache.karaf.management.cfg` file to include these properties at the end of the file:

```
secured = true
secureProtocol = TLSv1
keyAlias = jbossalias
keyStore = sample_keystore
trustStore = sample_keystore
```



### IMPORTANT

You must set **secureProtocol** to **TLSv1**, in order to protect against the [Poodle vulnerability \(CVE-2014-3566\)](#)



## Testing the Secure JMX connection

1. Restart Red Hat JBoss Fuse, if necessary.



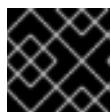
### NOTE

When you restart Red Hat JBoss Fuse, `fuse.log` will contain an **INFO** message that the `OsgiKeystoreManager` could not find the `sample_keystore` you configured in the `org.apache.karaf.management.cfg` file. This occurs because the `sample_keystore` was not loaded at the time the OSGi management bundle was loading.

But the `org.pache.felix.fileinstall-keystore.cfg` file you created forces the OSGi management bundle to reload and scan the keystore, enabling JConsole to connect successfully to Red Hat JBoss Fuse over SSL.

2. Open a terminal, and start up JConsole by entering this command:

```
jconsole -J-Djavax.net.debug=all -J-
Djavax.net.ssl.trustStore=/NotBackedUp/FuseSource/
jboss-fuse-6.1.0.redhat-379/etc/jbossweb.keystore -J-
Djavax.net.ssl.trustStoreType=JKS
-J-Djavax.net.ssl.trustStorePassword=JbossPassword
```



### IMPORTANT

Type the entire command on the same command line.



### NOTE

The system property `-J-Djavax.net.debug=all` shows all transport information, including SSL handshake messages, confirming that communication between JConsole and Red Hat JBoss Fuse occurs over SSL.

3. When JConsole opens, select the option **Remote Process** in the **New Connection** wizard.
4. Under the **Remote Process** option, enter these values for the `service:jmx:<protocol>:<sap>` url, **Username**, and **Password**:

```
service:jmx:<protocol>:<sap>:
service:jmx:rmi://localhost:44444/jndi/rmi://localhost:1099/karaf-
root
Username: admin
Password: admin
```

## 2.4. CONFIGURING ROLES FOR THE ADMINISTRATIVE PROTOCOLS

### Overview

By configuring each of the administrative functions to use a different role for authorization, you can provide fine grained control over who can monitor and manipulate running containers.

## Administration protocols

You can independently configure roles for the following different administrative protocols:

- SSH (remote console login)
- JMX management

### Default role

The default role name for all of the administration protocols is set by the `karaf.admin.role` property in the Red Hat JBoss Fuse's `etc/system.properties` file. For example, the default setting of `karaf.admin.role` is:

```
karaf.admin.role=admin
```

You have the option of overriding the default `admin` role set by `karaf.admin.role` for each of the administrative protocols.

### Changing the remote console's role

To override the default role for the remote console add a `sshRole` property to the `org.apache.karaf.shell` PID. The following sets the role to `admin`:

```
sshRole=admin
```

### Changing the JMX role

To override the default role for JMX add a `jmxRole` property to the `org.apache.karaf.management` PID. The following sets the role to `jmx`:

```
jmxRole=jmx
```

## 2.5. USING ENCRYPTED PROPERTY PLACEHOLDERS

### Overview

When securing a container it is undesirable to use plain text passwords in configuration files. They create easy to target security holes. One way to avoid this problem is to use encrypted property placeholders when ever possible.

Red Hat JBoss Fuse includes an extension to OSGi Blueprint that enables you to use Jasypt to decrypt property placeholders in blueprint files. It requires that you:

1. Create a properties file with encrypted values.
2. Add the proper namespaces to your blueprint file.
3. Import the properties using the Aries property placeholder extension.
4. Configure the Jasypt encryption algorithm.

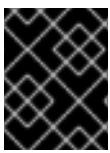
5. Use the placeholders in your blueprint file.
6. Ensure that the Jasypt features are installed into the JBoss Fuse container.

## Encrypted properties

Encrypted properties are stored in plain properties files. They are identified by wrapping them in the `ENC()` function as shown in [Example 2.8, “Property File with an Encrypted Property”](#).

### Example 2.8. Property File with an Encrypted Property

```
#ldap.properties
ldap.password=ENC(amIsvdqno9iSwnd7kAlLYQ==)
ldap.url=ldap://192.168.1.74:10389
```



### IMPORTANT

You will need to remember the password and algorithm used to encrypt the values. You will need this information to configure Jasypt.

## Namespaces

To use encrypted properties in your configuration, you will need to add the following namespaces to your blueprint file:

- Aries extensions—`http://aries.apache.org/blueprint/xmlns/blueprint-ext/v1.0.0`
- Apache Karaf Jasypt—`http://karaf.apache.org/xmlns/jasypt/v1.0.0`

[Example 2.9, “Encrypted Property Namespaces”](#) shows a blueprint file with the required namespaces.

### Example 2.9. Encrypted Property Namespaces

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0"
  xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">
  ...
</blueprint>
```

## Placeholder extension

In order to use encrypted property placeholders in a blueprint file you need to include an Aries `property-placeholder` element to your blueprint file. As shown in [Example 2.10, “Aries Placeholder Extension”](#), it must come before the Jasypt configuration or the use of placeholders.

### Example 2.10. Aries Placeholder Extension

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
```

```

xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0"
xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">

<ext:property-placeholder>
  <ext:location>file:etc/ldap.properties</ext:location>
</ext:property-placeholder>

...
</blueprint>

```

The Aries **property-placeholder** element's **ext:location** child specifies the location of the property file that contains the properties to use for the configuration. You can specify multiple files by using multiple **ext:location** children.

## Jasypt configuration

You configure Jasypt using the Apache Karaf **property-placeholder** element. It has one child, **encoder**, that contains the actual Jasypt configuration.

The **encoder** element's mandatory **class** attribute specifies the fully qualified classname of the Jasypt encryptor to use for decrypting the properties. The **encoder** element can take a **property** child that defines a Jasypt **PBEConfig** bean for configuring the encryptor.

For detailed information on how to configure the different Jasypt encryptors, see the [Jasypt documentation](#).

[Example 2.11, “Jasypt Blueprint Configuration”](#) shows configuration for using the string encryptor and retrieving the password from an environment variable.

### Example 2.11. Jasypt Blueprint Configuration

```

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
  ext/v1.0.0"
  xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">

  <ext:property-placeholder>
    <ext:location>file://ldap.properties</ext:location>
  </ext:property-placeholder>

  <enc:property-placeholder>
    <enc:encryptor
class="org.jasypt.encryption.pbe.StandardPBEStrngEncryptor">
      <property name="config">
        <bean
class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
          <property name="algorithm" value="PBEwithMD5AndDES" />
          <property name="password" value="FUSE_ENCRYPTION_PASSWORD" />
        </bean>
      </property>
    </enc:encryptor>

```

```

    </enc:property-placeholder>
    ...
</blueprint>

```

## Placeholders

The placeholder you use for encrypted properties are the same as you use for regular properties. The use the form `${prop.name}`.

[Example 2.12, “Jasypt Blueprint Configuration”](#) shows an LDAP JAAS realm that uses the properties file in [Example 2.8, “Property File with an Encrypted Property”](#).

### Example 2.12. Jasypt Blueprint Configuration

```

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0"
  xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">

  <ext:property-placeholder>
    <ext:location>file://ldap.properties</ext:location>
  </ext:property-placeholder>

  <enc:property-placeholder>
    <enc:encryptor
class="org.jasypt.encryption.pbe.StandardPBEStrngEncryptor">
      <property name="config">
        <bean
class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
          <property name="algorithm" value="PBEwithMD5AndDES" />
          <property name="password" value="FUSE_ENCRYPTION_PASSWORD" />
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>

  <jaas:config name="karaf" rank="1">
    <jaas:module
className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
flags="required">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      debug=true
      connectionURL=${ldap.url}

      connectionUsername=cn=mqbroker,ou=Services,ou=system,dc=jbossfuse,dc=com
      connectionPassword=${ldap.password}
      connectionProtocol=
      authentication=simple
      roleName=cn
      userBase = ou=User,ou=ActiveMQ,ou=system,dc=jbossfuse,dc=com
      userSearchMatching=(uid={0})
      userSearchSubtree=true
      roleBase = ou=Group,ou=ActiveMQ,ou=system,dc=jbossfuse,dc=com
      roleName=cn

```

```
        roleSearchMatching= (member:=uid={1})
        roleSearchSubtree=true
    </jaas:module>
</jaas:config>

</blueprint>
```

The `#{ldap.password}` placeholder will be replaced with the decrypted value of the `ldap.password` property from the properties file.

## Installing the Jasypt features

By default, JBoss Fuse does not have the Jasypt encryption libraries installed. In order to use encrypted property placeholders, you will need to install the **jasypt-encryption** feature using JBoss Fuse's `features:install` command as shown in [Example 2.13, "Installing the Jasypt Feature"](#).

### Example 2.13. Installing the Jasypt Feature

```
JBossFuse:karaf@root> features:install jasypt-encryption
```

## CHAPTER 3. SECURING THE JETTY HTTP SERVER

### Abstract

You can configure the built-in Jetty HTTP server to use SSL/TLS security by adding the relevant configuration properties to the `etc/org.ops4j.pax.web.cfg` configuration file. In particular, you can add SSL/TLS security to the Fuse Management Console in this way.

## JETTY SERVER

The JBoss Fuse container is pre-configured with a Jetty server, which acts as a general-purpose HTTP server and HTTP servlet container. Through a single HTTP port (by default, `http://Host:8181`), the Jetty container can host multiple services, for example:

- Fuse Management Console (by default, `http://Host:8181/hawtio`)
- Apache CXF Web services endpoints (if the host and port are left unspecified in the endpoint configuration)
- Some Apache Camel endpoints

If you use the default Jetty server for all of your HTTP endpoints, you can conveniently add SSL/TLS security to these HTTP endpoints by following the steps described here.

## CREATE X.509 CERTIFICATE AND PRIVATE KEY

Before you can enable SSL, you must create an X.509 certificate and private key for the Web console. The certificate and private key must be in Java keystore format. For details of how to create a signed certificate and private key, see [Appendix A, Managing Certificates](#).

## ENABLING SSL/TLS

To enable SSL/TLS:

1. Open `etc/org.ops4j.pax.web.cfg` in a text editor.
2. Disable the insecure HTTP port by adding the `org.osgi.service.http.enabled` and setting it to `false` as shown in [Example 3.1, “Pax Web Property for Disabling the HTTP Port”](#).

### Example 3.1. Pax Web Property for Disabling the HTTP Port

```
org.osgi.service.http.enabled=false
```

3. Enable the secure HTTPS port by adding the `org.osgi.service.http.secure.enabled` and setting it to `true` as shown in [Example 3.2, “Pax Web Property for Enabling the HTTPS Port”](#).

### Example 3.2. Pax Web Property for Enabling the HTTPS Port

```
org.osgi.service.http.secure.enabled=true
```

4. If you followed the preceding instructions, the `etc/org.ops4j.pax.web.cfg` file should now have the following contents:

```
# Configures the SMX Web Console to use SSL
org.ops4j.pax.web.config.file=etc/jetty.xml

org.osgi.service.http.enabled=false
org.osgi.service.http.port=8181

org.osgi.service.http.secure.enabled=true
```

5. Edit the `etc/jetty.xml` file and add the following `Call` element to configure the SSL connector for Jetty:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//
DTD Configure//EN" "http://jetty.mortbay.org/configure.dtd">

<Configure class="org.eclipse.jetty.server.Server">

    <!-- =====
-->
    <!-- Set connectors
-->
    <!-- =====
-->
    <!-- One of each type!
-->
    <!-- =====
-->
    ...
    <Call name="addConnector">
        <Arg>
            <!-- The SslSelectChannelConnector class uses the Java
            NIO SslEngine -->
            <New
            class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">
                <Arg>
                    <New
                    class="org.eclipse.jetty.http.ssl.SslContextFactory">
                        <!-- Protect against the POODLE security
                        vulnerability -->
                        <Set name="ExcludeProtocols">
                            <Array type="java.lang.String">
                                <Item>SSLv3</Item>
                            </Array>
                        </Set>
                        <Set
                        name="keyStore">/home/jdoe/Documents/server.keystore</Set>
                        <Set
                        name="keyStorePassword">mykeystorepass</Set>
                        <Set
                        name="keyManagerPassword">mykeypass</Set>
                    </New>
                </Arg>
```



```

        <Set name="port">8183</Set>
        <Set name="maxIdleTime">30000</Set>
    </New>
</Arg>
</Call>
<Call name="addConnector">
    ...
</Call>

<Call name="addBean">
    ...
</Call>
</Configure>

```



### IMPORTANT

The preceding configuration explicitly disables the SSLv3 protocol, in order to safeguard against the [Poodle vulnerability \(CVE-2014-3566\)](#). For more details, see [Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x](#).

6. Customize the properties of the **SslSocketConnector** instance defined in the `etc/jetty.xml` file, as follows:

#### port

The secure HTTPS port number.

#### keyStore

The location of the Java keystore file on the file system. Relative paths are resolved relative to the `KARAF_HOME` environment variable (by default, the install directory).

#### keyStorePassword

The *store password* that unlocks the Java keystore file.

#### keyManagerPassword

The *key password* that decrypts the private key stored in the keystore (usually the same as the store password).

7. Restart the JBoss Fuse container, in order for the configuration changes to take effect.

## CONNECT TO THE SECURE CONSOLE

After configuring SSL security for the Jetty server in the Pax Web configuration file, you should be able to open the Fuse Management Console by browsing to the following URL:

<https://localhost:8183>



### NOTE

Remember to type the **https:** scheme, instead of **http:**, in this URL.

Initially, the browser will warn you that you are using an untrusted certificate. Skip this warning and you will be presented with the login screen for the Fuse Management Console.

## ADVANCED JETTY SECURITY CONFIGURATION

The Jetty server provides flexible and sophisticated options for configuring security. You can exploit these advanced options by editing the `etc/jetty.xml` file and configuring it as described in the Jetty security documentation:

- [Jetty/Howto/Configure SSL](#)
- [Jetty/Reference/SSL Connectors](#)
- [SslContextFactory](#)
- [API documentation \(all Jetty versions\)](#)

## CHAPTER 4. SECURING THE CAMEL ACTIVEMQ COMPONENT

### Abstract

The Camel ActiveMQ component enables you to define JMS endpoints in your routes that can connect to an Apache ActiveMQ broker. In order to make your Camel ActiveMQ endpoints secure, you must create an instance of a Camel ActiveMQ component that uses a *secure* connection factory.

### 4.1. SECURE ACTIVEMQ CONNECTION FACTORY

#### Overview

Apache Camel provides an Apache ActiveMQ component for defining Apache ActiveMQ endpoints in a route. The Apache ActiveMQ endpoints are effectively Java clients of the broker and you can either define a consumer endpoint (typically used at the start of a route to *poll* for JMS messages) or define a producer endpoint (typically used at the end or in the middle of a route to *send* JMS messages to a broker).

When the remote broker is secure (SSL security, JAAS security, or both), the Apache ActiveMQ component must be configured with the required client security settings.

#### Programming the security properties

Apache ActiveMQ enables you to program SSL security settings (and JAAS security settings) by creating and configuring an instance of the **ActiveMQSslConnectionFactory** JMS connection factory. Programming the JMS connection factory is the correct approach to use in the context of the containers such as OSGi, J2EE, Tomcat, and so on, because these settings are local to the application using the JMS connection factory instance.



#### NOTE

A standalone broker can configure SSL settings using *Java system properties*. For clients deployed in a container, however, this is *not* a practical approach, because the configuration must apply only to individual bundles, not the entire OSGi container. A Camel ActiveMQ endpoint is effectively a kind of Apache ActiveMQ Java client, so this restriction applies also to Camel ActiveMQ endpoints.

#### Defining a secure connection factory

[Example 4.1, “Defining a Secure Connection Factory Bean”](#) shows how to create a secure connection factory bean in Spring XML, enabling both SSL/TLS security *and* JAAS authentication.

##### Example 4.1. Defining a Secure Connection Factory Bean

```
<bean id="jmsConnectionFactory"
      class="org.apache.activemq.ActiveMQSslConnectionFactory">
  <property name="brokerURL" value="ssl://localhost:61617" />
  <property name="userName" value="Username"/>
  <property name="password" value="Password"/>
  <property name="trustStore" value="/conf/client.ts"/>
  <property name="trustStorePassword" value="password"/>
</bean>
```

The following properties are specified on the **ActiveMQSslConnectionFactory** class:

**brokerURL**

The URL of the remote broker to connect to, where this example connects to an SSL-enabled OpenWire port on the local host. The broker must also define a corresponding transport connector with compatible port settings.

**userName and password**

Any valid JAAS login credentials, *Username* and *Password*.

**trustStore**

Location of the Java keystore file containing the certificate trust store for SSL connections. The location is specified as a classpath resource. If a relative path is specified, the resource location is relative to the **org/jbossfuse/example** directory on the classpath.

**trustStorePassword**

The password that unlocks the keystore file containing the trust store.

It is also possible to specify **keyStore** and **keyStorePassword** properties, but these would only be needed, if SSL mutual authentication is enabled (where the client presents an X.509 certificate to the broker during the SSL handshake).

## 4.2. EXAMPLE CAMEL ACTIVEMQ COMPONENT CONFIGURATION

### Overview

This section describes how to initialize and configure a sample Camel ActiveMQ component instance, which you can then use to define ActiveMQ endpoints in a Camel route. This makes it possible for a Camel route to send or receive messages from a broker.

### Prerequisites

The **activemq-camel** feature, which defines the bundles required for the Camel ActiveMQ component, is *not* installed by default. To install the **activemq-camel** feature, enter the following console command:

```
JBossFuse:karaf@root> features:install activemq-camel
```

### Sample Camel ActiveMQ component

The following Spring XML sample shows a complete configuration of a Camel ActiveMQ component that has both SSL/TLS security and JAAS authentication enabled. The Camel ActiveMQ component instance is defined to with the **activemqssl** bean ID, which means it is associated with the **activemqssl** scheme (which you use when defining endpoints in a Camel route).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ... >
```

```

...
<!--
  Configure the activemqssl component:
-->
<bean id="jmsConnectionFactory"
      class="org.apache.activemq.ActiveMQSslConnectionFactory">
  <property name="brokerURL" value="ssl://localhost:61617" />
  <property name="userName" value="Username"/>
  <property name="password" value="Password"/>
  <property name="trustStore" value="/conf/client.ts"/>
  <property name="trustStorePassword" value="password"/>
</bean>

<bean id="pooledConnectionFactory"
      class="org.apache.activemq.pool.PooledConnectionFactory">
  <property name="maxConnections" value="8" />
  <property name="maximumActive" value="500" />
  <property name="connectionFactory" ref="jmsConnectionFactory" />
</bean>

<bean id="jmsConfig"
class="org.apache.camel.component.jms.JmsConfiguration">
  <property name="connectionFactory" ref="pooledConnectionFactory"/>
  <property name="transacted" value="false"/>
  <property name="concurrentConsumers" value="10"/>
</bean>

<bean id="activemqssl"
      class="org.apache.activemq.camel.component.ActiveMQComponent">
  <property name="configuration" ref="jmsConfig"/>
</bean>

</beans>

```

## Sample Camel route

The following Camel route defines a sample endpoint that sends messages securely to the **security.test** queue on the broker, using the **activemqssl** scheme to reference the Camel ActiveMQ component defined in the preceding example:

```

<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  ...
  <camelContext xmlns="http://camel.apache.org/schema/spring">
    <route>
      <from uri="timer://myTimer?fixedRate=true&period=5000"/>
      <transform><constant>Hello world!</constant></transform>
      <to uri="activemqssl:security.test"/>
    </route>
  </camelContext>
  ...
</beans>

```

# CHAPTER 5. SECURING THE CAMEL JETTY COMPONENT

## Abstract

## 5.1. ENABLING SSL/TLS SECURITY

### Overview

This section explains how to enable SSL/TLS security on the Apache Camel Jetty component, which is used to create a HTTPS Web server. The key step is to customize the Jetty component by setting the `sslSocketConnectorProperties` property, which configures SSL/TLS. You must also change the protocol scheme on the Jetty URI from `http` to `https`.

### Tutorial steps

To configure SSL/TLS security for a Camel Jetty endpoint deployed in the OSGi container, perform the following steps:

1. [the section called “Generate a Maven project”](#).
2. [the section called “Customize the POM file”](#).
3. [the section called “Install sample keystore files”](#).
4. [the section called “Configure Jetty with SSL/TLS”](#).
5. [the section called “Build the bundle”](#).
6. [the section called “Install the camel-jetty feature”](#).
7. [the section called “Deploy the bundle”](#).
8. [the section called “Test the bundle”](#).
9. [the section called “Uninstall the bundle”](#).

### Generate a Maven project

The `maven-archetype-quickstart` archetype creates a generic Maven project, which you can then customize for whatever purpose you like. To generate a Maven project with the coordinates, `org.jbossfuse.example:jetty-security`, enter the following command:

```
mvn archetype:create
-DarchetypeArtifactId=maven-archetype-quickstart
-DgroupId=org.jbossfuse.example
-DartifactId=jetty-security
```

The result of this command is a directory, `ProjectDir/jetty-security`, containing the files for the generated project.

**NOTE**

*Be careful not to choose a group ID for your artifact that clashes with the group ID of an existing product! This could lead to clashes between your project's packages and the packages from the existing product (because the group ID is typically used as the root of a project's Java package names).*

**Customize the POM file**

You must customize the POM file in order to generate an OSGi bundle. Follow the POM customization steps described in [section "Generating a Bundle Project" in "Deploying into the Container"](#).

Alternatively, edit the `jetty-security/pom.xml` file and replace its contents with the following XML code:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.jbossfuse.example</groupId>
  <artifactId>jetty-security</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>bundle</packaging>

  <name>jetty-security</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <defaultGoal>install</defaultGoal>
    <plugins>
      <plugin>
        <groupId>org.apache.felix</groupId>
        <artifactId>maven-bundle-plugin</artifactId>
        <version>2.3.7</version>
        <extensions>>true</extensions>
        <configuration>
          <instructions>
            <Bundle-SymbolicName>${project.groupId}.${project.artifactId}
</Bundle-SymbolicName>
            <Import-Package>*</Import-Package>
          </instructions>

```

```

        </configuration>
    </plugin>

    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
            <source>1.6</source>
            <target>1.6</target>
        </configuration>
    </plugin>
</plugins>
</build>

</project>

```

## Install sample keystore files

The certificates used in this demonstration are taken from a sample in the Apache CXF 2.7.0.redhat-610379 distribution, which is included in the ***InstallDir/extras*** directory. Using a standard archive utility, expand the CXF archive file and extract the contents to a convenient location on your filesystem. You will find the sample certificates in the ***CXFInstallDir/samples/wsd1\_first\_https/src/main/config*** directory.

Copy the ***clientKeystore.jks*** certificate and the ***serviceKeystore.jks*** certificate from the ***CXFInstallDir/samples/wsd1\_first\_https/src/main/config*** directory to the ***EsbInstallDir/etc/certs*** directory (where you will need to create the ***etc/certs*** sub-directory). After copying, you should have the following directory structure under ***EsbInstallDir/etc/***:

```

EsbInstallDir/etc/
|
|--certs/
|
|--clientKeystore.jks
serviceKeystore.jks

```

Where ***clientKeystore.jks***, and ***serviceKeystore.jks*** are the keystores that are used in this demonstration.



### WARNING

The demonstration key store and trust store are provided for testing purposes only. *Do not deploy these certificates in a production system.* To set up a genuinely secure SSL/TLS system, you must generate custom certificates, as described in [Appendix A, Managing Certificates](#).

## Configure Jetty with SSL/TLS

The Jetty Web server is created by defining a Jetty endpoint at the start of an Apache Camel route. The



route is then responsible for processing the incoming HTTP request and generating a reply. The current example simply sends back a small HTML page in the reply. For a more realistic application, you would typically process the incoming message using a bean, which accesses the message through the Java servlet API.

Create the following directory to hold the Spring configuration files:

```
ProjectDir/jetty-security/src/main/resources/META-INF/spring
```

In the **spring** directory that you just created, use your favorite text editor to create the file, **jetty-spring.xml**, containing the following XML configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
    http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://activemq.apache.org/schema/core
http://activemq.apache.org/schema/core/activemq-core-5.4.0.xsd
    http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd">

    <bean id="jetty"
class="org.apache.camel.component.jetty.JettyHttpComponent">
        <property name="sslContextParameters" ref="sslContextParameters"
/>
    </bean>

    <sslContextParameters id="sslContextParameters"
xmlns="http://camel.apache.org/schema/spring">
        <secureSocketProtocols>
            <!-- Do NOT enable SSLv3 (POODLE vulnerability) -->
            <secureSocketProtocol>TLSv1</secureSocketProtocol>
            <secureSocketProtocol>TLSv1.1</secureSocketProtocol>
            <secureSocketProtocol>TLSv1.2</secureSocketProtocol>
        </secureSocketProtocols>
        <keyManagers keyPassword="skpass">
            <keyStore resource="etc/certs/serviceKeystore.jks"
password="sspass"/>
        </keyManagers>
        <trustManagers>
            <keyStore resource="etc/certs/serviceKeystore.jks"
password="sspass"/>
        </trustManagers>
    </sslContextParameters>

    <camelContext trace="true"
xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="jetty:https://0.0.0.0:8282/services?
matchOnUriPrefix=true"/>
            <transform>
                <constant>&lt;html&lt;body>Hello from Fuse ESB
server&lt;/body&lt;/html&lt;/constant>
            </transform>
        </route>
```

```

    </camelContext>
</beans>

```

The **jetty** bean defines a new instance of the Apache Camel Jetty component, overriding the default component defined in the **camel-jetty** JAR file. This Jetty component is configured using the **sslContextParameters** element, as follows:

### secureSocketProtocols

Explicitly lists the SSL/TLS protocols supported by the Jetty server.



#### IMPORTANT

This configuration explicitly disables the SSLv3 protocol, in order to safeguard against the [Poodle vulnerability \(CVE-2014-3566\)](#). For more details, see [Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x](#).

### keyManagers/@keyPassword

The password that decrypts the private key stored in the keystore (usually having the same value as **password**).

### keyManagers/keyStore/@resource

The location of the Java keystore file (in JKS format) containing the Jetty server's own X.509 certificate and private key. This location is specified on the filesystem (*not* on the classpath), relative to the directory where the OSGi container is started.

### keyManagers/keyStore/@password

The keystore password that unlocks the keystore.

### trustManagers/@resource

The location of the Java keystore file containing one or more trusted certificates (that is, the CA certificates that have been used to sign X.509 certificates from trusted clients). This location is specified on the filesystem (*not* on the classpath), relative to the directory where the OSGi container is started.

Strictly speaking, this property is not needed, if clients do not send certificates to the Jetty service.

### trustManagers/@password

The keystore password that unlocks the **truststore** trust store.

You must also modify the URI at the start of the route (the **uri** attribute of the **from** element). Make sure that the scheme of the URI matches the secure Jetty component, **jetty**, that you have just created. You must also change the protocol scheme from **http** to **https**.



#### NOTE

Always double-check you have changed the protocol scheme to **https**. This is such a small change, it is easy to forget.

## Build the bundle

Use Maven to build the bundle. Open a command prompt, switch the current directory to *ProjectDir/jetty-security*, and enter the following command:

```
mvn install -Dmaven.test.skip=true
```

This command builds the bundle and installs it in your local Maven repository.

## Install the camel-jetty feature

If you have not already done so, start up the Apache ServiceMix console (and container instance) by entering the following command in a new command prompt:

```
./fuse
```

The **camel-jetty** feature, which defines the bundles required for the Camel/Jetty component, is *not* installed by default. To install the **camel-jetty** feature, enter the following console command:

```
JBossFuse:karaf@root> features:install camel-jetty
```

## Deploy the bundle

To deploy and activate the bundle, enter the following console command:

```
JBossFuse:karaf@root> osgi:install -s mvn:org.jbossfuse.example/jetty-security
```

The preceding command loads the bundle from your local Maven repository. You might need to configure the Mvn URL handler with the location of your local Maven repository, if the bundle cannot be found (see [section "Mvn URL Handler" in "Deploying into the Container"](#)).

## Test the bundle

To test the Jetty service, open your favorite Web browser and navigate to the following URL:

```
https://localhost:8282/services
```

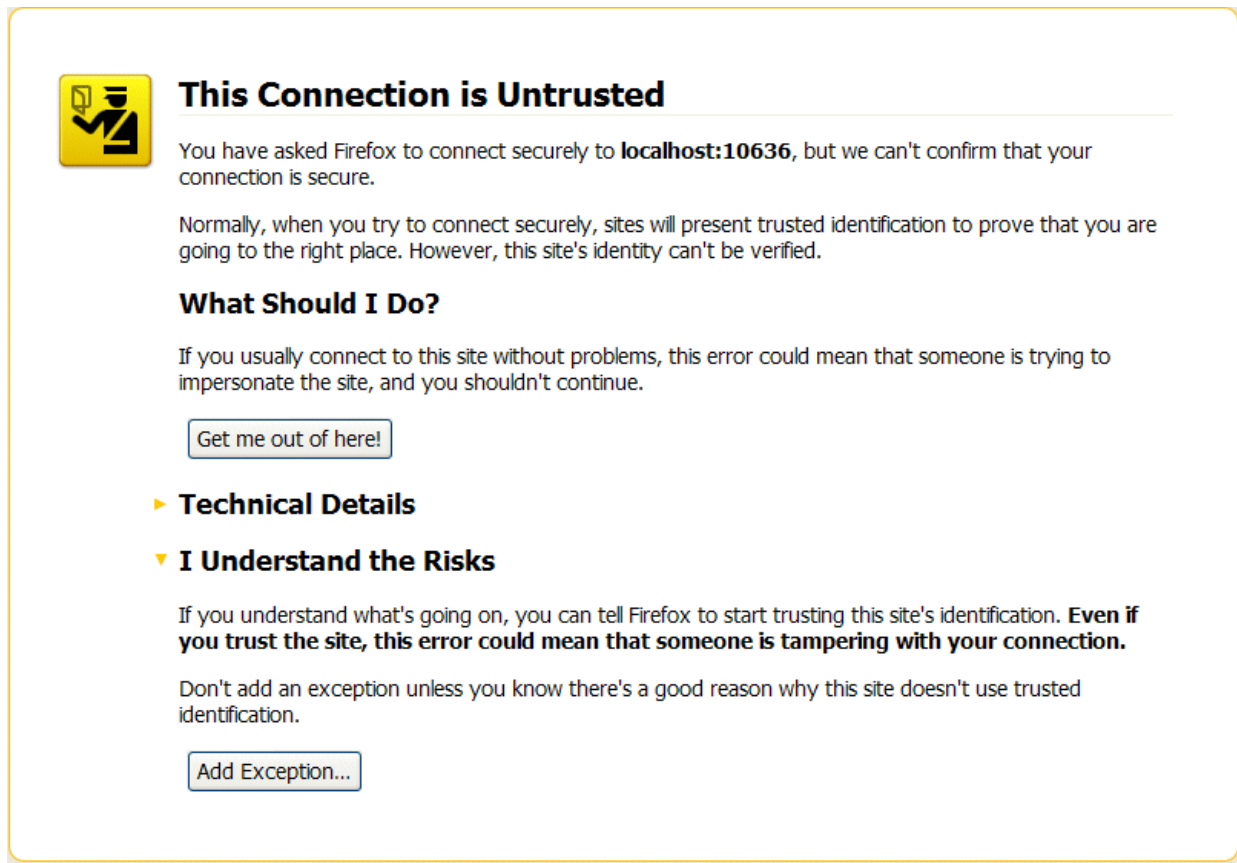


### NOTE

Don't forget to use **https:** instead of **http:** in the URL!

Because the Jetty service uses an untrusted certificate, your browser will initially present you with a warning about the untrusted certificate. For example, the Firefox browser displays the following warning screen:

Figure 5.1. Untrusted Certificate Warning



**This Connection is Untrusted**

You have asked Firefox to connect securely to **localhost:10636**, but we can't confirm that your connection is secure.

Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.

**What Should I Do?**

If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

▶ **Technical Details**

▼ **I Understand the Risks**

If you understand what's going on, you can tell Firefox to start trusting this site's identification. **Even if you trust the site, this error could mean that someone is tampering with your connection.**

Don't add an exception unless you know there's a good reason why this site doesn't use trusted identification.

To proceed with contacting the Jetty service, click **I Understand the Risks** and then click **Add Exception**, which brings up the **Add Security Exception** dialog. In the **Add Security Exception** dialog, make sure that the **Permanently store this exception option** is unchecked and click **Confirm Security Exception**.

The browser window should now display the following text:

```
Hello from Fuse ESB server
```

## Uninstall the bundle

To uninstall the broker bundle, you need to know its bundle ID, *BundleID*, in which case you can uninstall it by entering the following console command:

```
JBossFuse:karaf@root> osgi:uninstall BundleID
```

## 5.2. BASIC AUTHENTICATION WITH JAAS

### Overview

The HTTP BASIC authentication protocol is a simple username/password authentication mechanism that is integrated into HTTP and is supported by most Web browsers. To enable BASIC authentication in Jetty, you use the Jetty security API, which enables BASIC authentication by associating a security handler with the Jetty endpoint.

Jetty also enables you to plug in a JAAS login module to perform the credentials check. Using this feature, it is possible to integrate credentials checking with any JAAS realm provided by the Red Hat

JBoss Fuse OSGi container. In the example shown here, the Jetty authentication is integrated with the default JAAS realm, **karaf**.

## Prerequisites

This example builds on the project created in [Section 5.1, “Enabling SSL/TLS Security”](#). You must complete the steps in the Jetty SSL/TLS example before proceeding with this tutorial.



### NOTE

In any case, it is highly recommended that you *always* enable SSL/TLS in combination with BASIC authentication, in order to protect against password snooping.

## Authentication steps

To configure HTTP BASIC authentication for a Camel Jetty endpoint deployed in the OSGi container, perform the following steps:

1. [the section called “Add the Jetty security handler configuration”](#).
2. [the section called “Modify Camel Jetty endpoint”](#).
3. [the section called “Add required package imports to POM”](#).
4. [the section called “Build the bundle”](#).
5. [the section called “Install the required features”](#).
6. [the section called “Deploy the bundle”](#).
7. [the section called “Test the bundle”](#).

## Add the Jetty security handler configuration

In the **jetty-security** project, edit the **jetty-spring.xml** file from the **src/main/resources/META-INF/spring** directory. To configure the Jetty security handler with BASIC authentication, add the following bean definitions:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
    ...
    <!-- -->
    <bean id="loginService"
class="org.eclipse.jetty.plus.jaas.JAASLoginService">
        <property name="name" value="karaf"/>
        <property name="loginModuleName" value="karaf"/>
        <property name="roleClassNames">
            <list>
<value>org.apache.karaf.jaas.boot.principal.RolePrincipal</value>
            </list>
        </property>
    </bean>

    <bean id="identityService"
```

```

class="org.eclipse.jetty.security.DefaultIdentityService"/>

    <bean id="constraint"
class="org.eclipse.jetty.util.security.Constraint">
    <property name="name" value="BASIC"/>
    <property name="roles" value="admin"/>
    <property name="authenticate" value="true"/>
    </bean>

    <bean id="constraintMapping"
class="org.eclipse.jetty.security.ConstraintMapping">
    <property name="constraint" ref="constraint"/>
    <property name="pathSpec" value="/*"/>
    </bean>

    <bean id="securityHandler"
class="org.eclipse.jetty.security.ConstraintSecurityHandler">
    <property name="authenticator">
    <bean
class="org.eclipse.jetty.security.authentication.BasicAuthenticator"/>
    </property>
    <property name="constraintMappings">
    <list>
    <ref bean="constraintMapping"/>
    </list>
    </property>
    <property name="loginService" ref="loginService"/>
    <property name="strict" value="false"/>
    <property name="identityService" ref="identityService"/>
    </bean>
    ...
</beans>

```

Two aspects of Jetty authentication are configured by the preceding bean definitions:

- *HTTP BASIC authentication*—the **constraint** bean enables HTTP BASIC authentication on the Jetty security handler.
- *JAAS login service*—the **loginService** bean specifies that the requisite authentication data is extracted from a JAAS realm. The **loginModuleName** property specifies that the Jetty login service uses the **karaf** JAAS realm, which is the OSGi container's default JAAS realm (see [Section 1.1, “OSGi Container Security”](#)).

## Modify Camel Jetty endpoint

After creating the Jetty **securityHandler** bean, you must modify the Jetty endpoint URI in the Apache Camel route, so that it hooks into the security handler. To add the security handler to the Jetty endpoint, set the **handlers** option equal to the security handler's bean ID, as shown in the following example:

```

<beans ...>
    <camelContext trace="true"
xmlns="http://camel.apache.org/schema/spring">
    <route>
    <from
uri="jetty:https://0.0.0.0:8282/services?handlers=securityHandler&matc

```

```

hOnUriPrefix=true"/>
    <transform>
        <constant>&lt;html>&lt;body>Hello from Fuse ESB
server&lt;/body>&lt;/html></constant>
    </transform>
</route>
</camelContext>
</beans>

```



## NOTE

URI options must be separated by the **&amp;** entity, instead of the plain **&** character, in the context of an XML file.

## Add required package imports to POM

Edit the **jetty-security** project's POM file, **jetty-security/pom.xml**. Near the start of the POM file, define the **jetty-version** property as follows:

```

<project ... >
    ...
    <properties>
        ...
        <jetty-version>7.2.2.v20101205</jetty-version>
    </properties>
    ...
</project>

```

Further down the POM file, in the configuration of the Maven bundle plug-in, modify the bundle instructions to import additional Java packages, as follows:

```

<project ... >
    ...
    <build>
        ...
        <plugin>
            <groupId>org.apache.felix</groupId>
            <artifactId>maven-bundle-plugin</artifactId>
            <extensions>true</extensions>
            <configuration>
                <instructions>
                    <Bundle-SymbolicName>
                        ${project.artifactId}
                    </Bundle-SymbolicName>
                    <Import-Package>
                        javax.security.auth,
                        javax.security.auth.callback,
                        javax.security.auth.login,
                        javax.security.auth.spi,
                        org.apache.karaf.jaas.modules,
                        org.apache.karaf.jaas.boot.principal,
                        org.eclipse.jetty.plus.jaas;version=${jetty-version},
                        org.eclipse.jetty.security;version=${jetty-version},
                        *
                </instructions>
            </configuration>
        </plugin>
    </build>
</project>

```

```

        </Import-Package>
        <Private-Package>org.apache.camel.jaas</Private-Package>

    </instructions>
  </configuration>
</plugin>
</plugins>
</build>
...
</project>

```

**NOTE**

These extra imports are required, because the Maven bundle plug-in is not capable of scanning Spring files to determine their package dependencies automatically.

**Build the bundle**

Use Maven to build the bundle. Open a command prompt, switch the current directory to *ProjectDir/jetty-security*, and enter the following command:

```
mvn install
```

**Install the required features**

If you have not already done so, start up the Apache ServiceMix console (and container instance) by entering the following command in a new command prompt:

```
servicemix
```

Install the **jetty** and **camel-jetty** features, by entering the following console commands:

```
karaf@root> features:install jetty
karaf@root> features:install camel-jetty
```

**Deploy the bundle**

To deploy and activate the bundle, enter the following console command:

```
JBossFuse:karaf@root> osgi:install -s mvn:org.jbossfuse.example/jetty-
security
```

**Test the bundle**

To test the Jetty service, open your favorite Web browser and navigate to the following URL:

```
https://localhost:8282/services
```

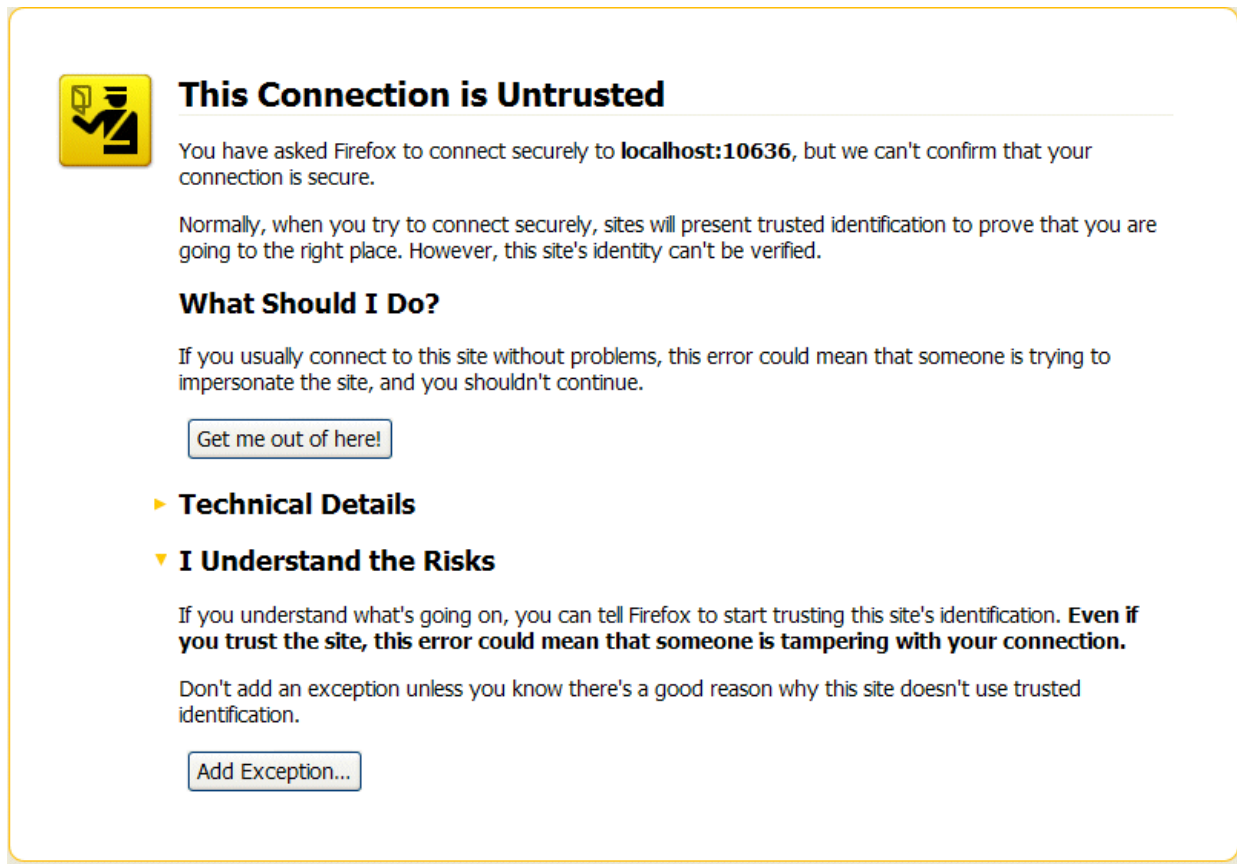
**NOTE**

Don't forget to use **https:** instead of **http:** in the URL!



If you closed your browser since running the Jetty SSL/TLS demonstration (thus re-initializing the security status of your browser), your browser will initially present you with a warning about the untrusted certificate. For example, the Firefox browser displays the following warning screen:

**Figure 5.2. Untrusted Certificate Warning**



To proceed with contacting the Jetty service, click **I Understand the Risks** and then click **Add Exception**, which brings up the **Add Security Exception** dialog. In the **Add Security Exception** dialog, make sure that the **Permanently store this exception option** is unchecked and click **Confirm Security Exception**.

You will now be prompted to authenticate yourself with a username and password (this is the BASIC authentication step). Enter the username, **smx**, and the password, **smx**, and click **Ok** (the valid credentials you can use for this step are specified in the ***EsbInstallDir/etc/users.properties*** file). The browser window should now display the following text:

```
Hello from Fuse ESB server
```

## CHAPTER 6. SECURING THE CAMEL CXF COMPONENT

### Abstract

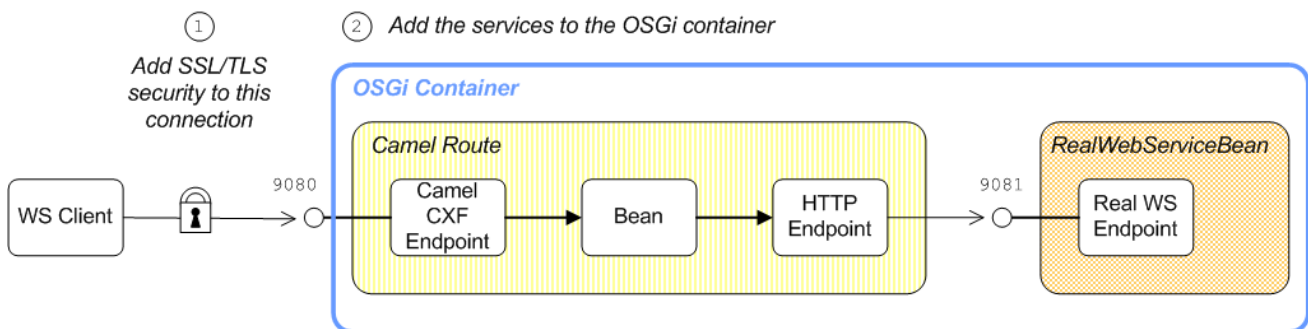
This chapter explains how to enable SSL/TLS security on a Camel CXF endpoint, using the Camel CXF proxy demonstration as the starting point. The Camel CXF component enables you to add Apache CXF endpoints to your Apache Camel routes. This makes it possible to simulate a Web service in Apache Camel or you could interpose a route between a WS client and a Web service to perform additional processing (which is the case considered here).

### 6.1. THE CAMEL CXF PROXY DEMONSTRATION

#### Overview

In order to explain how to secure a Camel CXF endpoint in OSGi, this tutorial builds on an example available from the standalone distribution of Apache Camel, the *Camel CXF proxy* demonstration. [Figure 6.1, “Camel CXF Proxy Overview”](#) gives an overview of how this demonstration works

**Figure 6.1. Camel CXF Proxy Overview**



The *report incident* Web service, which is implemented by the **RealWebServiceBean**, receives details of an incident (for example, a traffic accident) and returns a tracking code to the client. Instead of sending its requests directly to the real Web service, however, the WS client connects to a Camel CXF endpoint, which is interposed between the WS client and the real Web service. The Apache Camel route performs some processing on the WSDL message (using the **enrichBean**) before forwarding it to the real Web service.



#### WARNING

If you enable SSL/TLS security, you must ensure that you explicitly disable the SSLv3 protocol, in order to safeguard against the [Poodle vulnerability \(CVE-2014-3566\)](#). For more details, see [Disabling SSLv3 in JBoss Fuse 6.x and JBoss A-MQ 6.x](#).

#### Modifications

In order to demonstrate how to enable SSL/TLS on a Camel CXF endpoint in the context of OSGi, this chapter contains instructions on how to modify the basic demonstration as follows:

1. SSL/TLS security is enabled on the connection between the WS client and the Camel CXF endpoint.
2. The Apache Camel route and the **RealWebServiceBean** bean are both deployed into the OSGi container.

## Obtaining the demonstration code

The Camel CXF proxy demonstration is available only from the standalone distribution of Apache Camel, which is included in the **InstallDir/extras** directory. Using a standard archive utility, expand the Camel archive file and extract the contents to a convenient location on your filesystem.

Assuming that you have installed Apache Camel in *CamelInstallDir*, you can find the Camel CXF proxy demonstration in the following directory:

```
CamelInstallDir/examples/camel-example-cxf-proxy
```

## Obtaining the sample certificates

This demonstration needs X.509 certificates. In a real deployment, you should generate these certificates yourself using a private certificate authority. For this demonstration, however, we use some sample certificates from the Apache CXF **wSDL\_first\_http** example. This demonstration is available from the standalone distribution of Apache CXF, which is included in the **InstallDir/extras** directory. Using a standard archive utility, expand the CXF archive file and extract the contents to a convenient location on your filesystem.

Assuming that you have installed Apache CXF in *CXFInstallDir*, you can find the **wSDL\_first\_http** demonstration in the following directory:

```
CXFInstallDir/samples/wSDL_first_http
```

## Physical part of the WSDL contract

The physical part of the WSDL contract refers to the **wSDL:service** and **wSDL:port** elements. These elements specify the transport details that are needed to connect to a specific Web services endpoint. For the purposes of this demonstration, this is the most interesting part of the contract and it is shown in [Example 6.1, “The ReportIncidentEndpointService WSDL Service”](#).

### Example 6.1. The ReportIncidentEndpointService WSDL Service

```
<wSDL:definitions xmlns:soap="http://schemas.xmlsoap.org/wSDL/soap/"
  ...
  xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/"
  targetNamespace="http://reportincident.example.camel.apache.org">
  ...
  <!-- Service definition -->
  <wSDL:service name="ReportIncidentEndpointService">
    <wSDL:port name="ReportIncidentEndpoint"
binding="tns:ReportIncidentBinding">
      <soap:address location="http://localhost:9080/camel-
example-cxf-proxy/webservices/incident"/>
    </wSDL:port>
```

```
</wsdl:service>
</wsdl:definitions>
```



#### NOTE

The address URL appearing in the WSDL contract (the value of the **soap:address** element's **location** attribute) is not important here, because the application code overrides the default value of the address URL.

## WSDL addressing details

A WS client needs three pieces of information to connect to a WSDL service: the *WSDL service name*, the *WSDL port name*, and the *address URL* of the Web service. The following addressing details are used to connect to the proxy Web service and to the real Web service in this example:

### WSDL service name

The full QName of the WSDL service is as follows:

```
{http://reportincident.example.camel.apache.org}ReportIncidentEndpointService
```

### WSDL port name

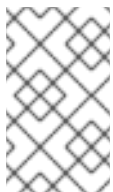
The full QName of the WSDL port is as follows:

```
{http://reportincident.example.camel.apache.org}ReportIncidentEndpoint
```

### Address URL

The address URL of the *proxy Web service* endpoint (which uses the HTTPS protocol) is as follows:

```
https://localhost:9080/camel-example-cxf-proxy/webservices/incident
```



#### NOTE

The preceding address is specified when the **reportIncident** bean is created using a **cxf:cxfEndpoint** element in the bundle's Spring configuration file, **src/main/resources/META-INF/spring/camel-config.xml**.

The address URL of the *real Web service* endpoint (using the HTTP protocol) is as follows:

```
http://localhost:9081/real-webservice
```



#### NOTE

The preceding address is specified when the **realWebService** bean is created in the bundle's Spring configuration file, **src/main/resources/META-INF/spring/camel-config.xml**.

## 6.2. SECURING THE WEB SERVICES PROXY

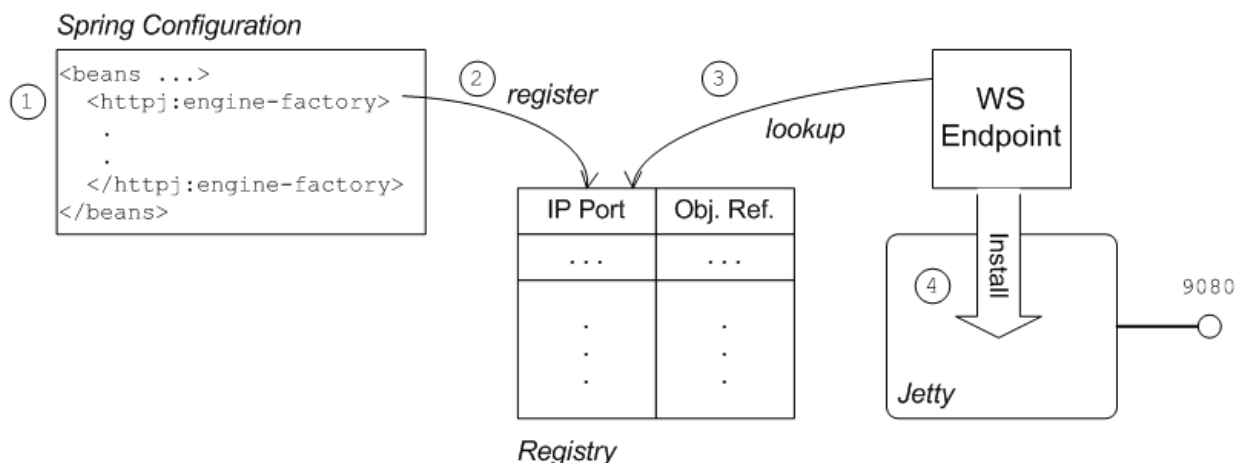
### Overview

This section explains how to enable SSL/TLS security on the Camel CXF endpoint, which acts as a proxy for the real Web service. Assuming that you already have the X.509 certificates available, all that is required is to add a block of configuration data to the Spring configuration file (where the configuration data is contained in a `httpj:engine-factory` element). There is just one slightly subtle aspect to this, however: you need to understand how the Camel CXF endpoint gets associated with the SSL/TLS configuration details.

### Implicit configuration

A WS endpoint can be configured by creating the endpoint in Spring and then configuring SSL/TLS properties on its Jetty container. The configuration can be somewhat confusing, however, for the following reason: the Jetty container (which is configured by a `httpj:engine-factory` element in Spring) *does not explicitly reference the WS endpoints it contains* and the WS endpoints *do not explicitly reference the Jetty container* either. The connection between the Jetty container and its contained endpoints is established implicitly, in that they are both configured to use the same IP port, as illustrated by Figure 6.2, “WS Endpoint Implicitly Configured by `httpj:engine-factory` Element”.

**Figure 6.2. WS Endpoint Implicitly Configured by `httpj:engine-factory` Element**



The connection between the Web service endpoint and the `httpj:engine-factory` element is established as follows:

1. The Spring container loads and parses the file containing the `httpj:engine-factory` element.
2. When the `httpj:engine-factory` bean is created, a corresponding entry is created in the registry, storing a reference to the bean. The `httpj:engine-factory` bean is also used to initialize a Jetty container that listens on the specified IP port.
3. When the WS endpoint is created, it scans the registry to see if it can find a `httpj:engine-factory` bean with the same IP port as the IP port in the endpoint's address URL.
4. If one of the beans matches the endpoint's IP port, the WS endpoint installs itself into the corresponding Jetty container. If the Jetty container has SSL/TLS enabled, the WS endpoint shares those security settings.

### Steps to add SSL/TLS security to the Jetty container

To add SSL/TLS security to the Jetty container, thereby securing the WS proxy endpoint, perform the following steps:

1. the section called “Add certificates to the bundle resources”.
2. the section called “Modify POM to switch off resource filtering”.
3. the section called “Instantiate the CXF Bus”.
4. the section called “Add the httpj:engine-factory element to Spring configuration”.
5. the section called “Define the cxfcore:, sec: and httpj: prefixes”.
6. the section called “Modify proxy address URL to use HTTPS”.

## Add certificates to the bundle resources

The certificates used in this demonstration are taken from a sample in the Apache CXF 2.7.0.redhat-610379 product. If you install the standalone version of Apache CXF (available in the *InstallDir/extras/* directory), you will find the sample certificates in the *CXFInstallDir/samples/wsd1\_first\_https/src/main/config* directory.

Copy the *clientKeystore.jks* and *serviceKeystore.jks* keystores from the *CXFInstallDir/samples/wsd1\_first\_https/src/main/config* directory to the *CamelInstallDir/examples/camel-example-cxf-proxy/src/main/resources/certs* directory (you must first create the *certs* sub-directory).

## Modify POM to switch off resource filtering

Including the certificates directly in the bundle as resource is the most convenient way to deploy them. But when you deploy certificates as resources in a Maven project, you must remember to disable Maven resource filtering, which corrupts binary files.

To disable filtering of *.jks* files in Maven, open the project POM file, *CamelInstallDir/examples/camel-example-cxf-proxy/pom.xml*, with a text editor and add the following *resources* element as a child of the *build* element:

```
<?xml version="1.0" encoding="UTF-8"?>
...
<project ...>
...
  <build>
    <plugins>
      ...
    </plugins>

    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>>true</filtering>
        <excludes>
          <exclude>**/*.jks</exclude>
        </excludes>
      </resource>
    </resources>
  </build>
</project>
```

```

        <directory>src/main/resources</directory>
        <filtering>>false</filtering>
        <includes>
            <include>**/*.jks</include>
        </includes>
    </resource>
</resources>
</build>

</project>

```

## Instantiate the CXF Bus

You should instantiate the CXF bus explicitly in the Spring XML (this ensures that it will be available to the Jetty container, which is instantiated by the `httpj:engine-factory` element in the next step). Edit the `camel-config.xml` file in the `src/main/resources/META-INF/spring` directory, adding the `cxfcore:bus` element as a child of the `beans` element, as follows:

```

<beans ... >
    ...
    <cxfcore:bus/>
    ...
</beans>

```



### NOTE

The `cxfcore:` namespace prefix will be defined in a later step.

## Add the `httpj:engine-factory` element to Spring configuration

To configure the Jetty container that listens on IP port 9080 to use SSL/TLS security, edit the `camel-config.xml` file in the `src/main/resources/META-INF/spring` directory, adding the `httpj:engine-factory` element as shown in [Example 6.2, “httpj:engine-factory Element with SSL/TLS Enabled”](#).

In this example, the `required` attribute of the `sec:clientAuthentication` element is set to `false`, which means that a connecting client is *not* required to present an X.509 certificate to the server during the SSL/TLS handshake (although it may do so, if it has such a certificate).

### Example 6.2. `httpj:engine-factory` Element with SSL/TLS Enabled

```

<beans ... >
    ...
    <httpj:engine-factory bus="cxf">
        <httpj:engine port="{proxy.port}">
            <httpj:tlsServerParameters secureSocketProtocol="TLSv1">
                <sec:keyManagers keyPassword="skpass">
                    <sec:keyStore resource="certs/serviceKeystore.jks"
password="sspass" type="JKS"/>
                </sec:keyManagers>
                <sec:trustManagers>
                    <sec:keyStore resource="certs/serviceKeystore.jks"
password="sspass" type="JKS"/>
                </sec:trustManagers>
            </httpj:tlsServerParameters>
        </httpj:engine>
    </httpj:engine-factory>
    ...
</beans>

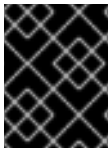
```

```

        </sec:trustManagers>
        <sec:cipherSuitesFilter>
            <sec:include>.*_WITH_3DES_.*</sec:include>
            <sec:include>.*_WITH_DES_.*</sec:include>
            <sec:exclude>.*_WITH_NULL_.*</sec:exclude>
            <sec:exclude>.*_DH_anon_.*</sec:exclude>
        </sec:cipherSuitesFilter>
        <sec:clientAuthentication want="true" required="false"/>
    </httpj:tlsServerParameters>
</httpj:engine>
</httpj:engine-factory>

</beans>

```



### IMPORTANT

You must set `secureSocketProtocol` to `TLSv1` on the server side, in order to protect against the [Poodle vulnerability \(CVE-2014-3566\)](#)

## Define the `cxfcORE:`, `sec:` and `httpj:` prefixes

Define the `cxfcORE:`, `sec:` and `httpj:` namespace prefixes, which appear in the definitions of the `cxfcORE:bus` element and the `httpj:engine-factory` element, by adding the following highlighted lines to the `beans` element in the `camel-config.xml` file:

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:camel="http://camel.apache.org/schema/spring"
    xmlns:cxf="http://camel.apache.org/schema/cxf"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:cxfcORE="http://cxf.apache.org/core"
    xmlns:sec="http://cxf.apache.org/configuration/security"
    xmlns:httpj="http://cxf.apache.org/transports/http-
jetty/configuration"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://camel.apache.org/schema/spring
http://camel.apache.org/schema/spring/camel-spring.xsd
        http://camel.apache.org/schema/cxf
http://camel.apache.org/schema/cxf/camel-cxf.xsd
        http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
        http://cxf.apache.org/core http://cxf.apache.org/schemas/core.xsd
        http://cxf.apache.org/configuration/security
http://cxf.apache.org/schemas/configuration/security.xsd
        http://cxf.apache.org/transports/http-jetty/configuration
http://cxf.apache.org/schemas/configuration/http-jetty.xsd
    ">

```





## NOTE

It is essential to specify the locations of the <http://cxf.apache.org/configuration/security> schema and the <http://cxf.apache.org/transports/http-jetty/configuration> schema in the `xsi:schemaLocation` attribute. These will not automatically be provided by the OSGi container.

## Modify proxy address URL to use HTTPS

The proxy endpoint at the start of the Apache Camel route is configured by the `cxf:cxfEndpoint` element in the `camel-config.xml` file. By default, this proxy endpoint is configured to use the HTTP protocol. You must modify the address URL to use the secure HTTPS protocol instead, however. In the `camel-config.xml` file, edit the address attribute of the `cxf:cxfEndpoint` element, replacing the `http:` prefix by the `https:` prefix, as shown in the following fragment:

```
<beans ...>
  ...
  <cxf:cxfEndpoint id="reportIncident"
                  address="https://localhost:${proxy.port}/camel-
example-cxf-proxy/webservices/incident"
                  endpointName="s:ReportIncidentEndpoint"
                  serviceName="s:ReportIncidentEndpointService"
                  wsdlURL="etc/report_incident.wsdl"

xmlns:s="http://reportincident.example.camel.apache.org"/>
  ...
</beans>
```

Notice also that the address URL is configured to use the IP port, `${proxy.port}` (which has the value `9080` by default). This IP port value is the same as the value set for the Jetty container (configured by the `http:engine-factory` element), thus ensuring that this endpoint is deployed into the Jetty container. The attributes of the `cxf:cxfEndpoint` specify the WSDL addressing details as described in [the section called “WSDL addressing details”](#):

### serviceName

Specifies the WSDL service name.

### endpointName

Specifies the WSDL port name.

### address

Specifies the address URL of the proxy Web service.

## 6.3. DEPLOYING THE APACHE CAMEL ROUTE

### Overview

The Maven POM file in the basic Camel CXF proxy demonstration is already configured to generate an OSGi bundle. Hence, after building the demonstration using Maven, the demonstration bundle (which contains the Apache Camel route and the `RealWebServicesBean` bean) is ready for deployment into

the OSGi container.

## Prerequisites

Before deploying the Apache Camel route into the OSGi container, you must configure the proxy Web service to use SSL/TLS security, as described in the previous section, [Section 6.2, “Securing the Web Services Proxy”](#).

## Steps to deploy the Camel route

To deploy the Web services proxy demonstration into the OSGi container, perform the following steps:

1. [the section called “Build the demonstration”](#).
2. [the section called “Start the OSGi container”](#).
3. [the section called “Install the required features”](#).
4. [the section called “Deploy the bundle”](#).
5. [the section called “Check the console output”](#).

## Build the demonstration

Use Maven to build and install the demonstration as an OSGi bundle. Open a command prompt, switch the current directory to `CamelInstallDir/examples/camel-example-cxf-proxy`, and enter the following command:

```
mvn install -Dmaven.test.skip=true
```

## Start the OSGi container

If you have not already done so, start up the Apache ServiceMix console (and container instance) by entering the following command in a new command prompt:

```
./fuse
```

## Install the required features

The `camel-cxf` feature, which defines the bundles required for the Camel/CXF component, is *not* installed by default. To install the `camel-cxf` feature, enter the following console command:

```
JBossFuse:karaf@root> features:install camel-cxf
```

You also need the `camel-http` feature, which defines the bundles required for the Camel/HTTP component. To install the `camel-http` feature, enter the following console command:

```
JBossFuse:karaf@root> features:install camel-http
```

## Deploy the bundle

Deploy the `camel-example-cxf-proxy` bundle, by entering the following console command:

```
JBossFuse:karaf@root> install -s mvn:org.apache.camel/camel-example-cxf-proxy/2.12.0.redhat-610379
```



## NOTE

In this case, it is preferable to deploy the bundle directly using **install**, rather than using hot deploy, so that you can see the bundle output on the console screen.

If you have any difficulty using the **mvn** URL handler, see [section "Mvn URL Handler" in "Deploying into the Container"](#) for details of how to set it up.

## Check the console output

After the bundle is successfully deployed, you should see output like the following in the console window:

```
JBossFuse:karaf@root> Starting real web service...
Started real web service at: http://localhost:9081/real-webservice
```

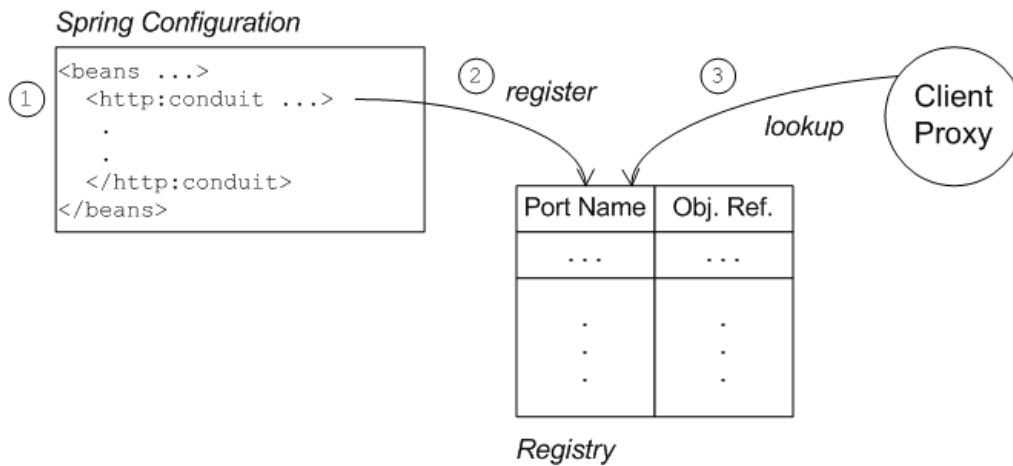
## 6.4. SECURING THE WEB SERVICES CLIENT

### Overview

In the basic Camel CXF proxy demonstration, the Web services client is actually implemented as a JUnit test under the **src/test** directory. This means that the client can easily be run using the Maven command, **mvn test**. To enable SSL/TLS security on the client, the Java implementation of the test client is completely replaced and a Spring file, containing the SSL/TLS configuration, is added to the **src/test/resources/META-INF/spring** directory. Before describing the steps you need to perform to set up the client, this section explains some details of the client's Java code and Spring configuration.

### Implicit configuration

Apart from changing the URL scheme on the endpoint address to **https:**, most of the configuration to enable SSL/TLS security on a client proxy is contained in a **http:conduit** element in Spring configuration. The way in which this configuration is applied to the client proxy, however, is potentially confusing, for the following reason: the **http:conduit** element *does not explicitly reference the client proxy* and the client proxy *does not explicitly reference the http:conduit element*. The connection between the **http:conduit** element and the client proxy is established implicitly, in that they both reference the same WSDL port, as illustrated by [Figure 6.3, "Client Proxy Implicitly Configured by http:conduit Element"](#).

**Figure 6.3. Client Proxy Implicitly Configured by http:conduit Element**

The connection between the client proxy and the **http:conduit** element is established as follows:

1. The client loads and parses the Spring configuration file containing the **http:conduit** element.
2. When the **http:conduit** bean is created, a corresponding entry is created in the registry, which stores a reference to the bean under the specified WSDL port name (where the name is stored in QName format).
3. When the JAX-WS client proxy is created, it scans the registry to see if it can find a **http:conduit** bean associated with the proxy's WSDL port name. If it finds such a bean, it automatically injects the configuration details into the proxy.

## Certificates needed on the client side

The client is configured with the following **clientKeystore.jks** keystore file from the **src/main/resources/certs** directory. This keystore contains two entries, as follows:

### Trusted cert entry

A trusted certificate entry containing the CA certificate that issued and signed both the server certificate and the client certificate.

### Private key entry

A private key entry containing the client's own X.509 certificate and private key. In fact, this certificate is not strictly necessary to run the current example, because the server does not require the client to send a certificate during the TLS handshake (see [Example 6.2, "httpj:engine-factory Element with SSL/TLS Enabled"](#)).

## Loading Spring definitions into the client

The example client is not deployed directly into a Spring container, but it requires some Spring definitions in order to define a secure HTTP conduit. So how can you create the Spring definitions without a Spring container? It turns out that it is easy to read Spring definitions into a Java-based client using the **org.apache.cxf.bus.spring.SpringBusFactory** class.

The following code shows how to read Spring definitions from the file, **META-INF/spring/cxf-client.xml**, and create an Apache CXF Bus object that incorporates those definitions:

```
// Java
```

```
import org.apache.cxf.bus.spring.SpringBusFactory;
...
protected void startCxfBus() throws Exception {
    bf = new SpringBusFactory();
    Bus bus = bf.createBus("META-INF/spring/cxf-client.xml");
    bf.setDefaultBus(bus);
}
```

## Creating the client proxy

In principle, there are several different ways of creating a WSDL proxy: you could use the JAX-WS API to create a proxy based on the contents of a WSDL file; you could use the JAX-WS API to create a proxy *without* a WSDL file; or you could use the Apache CXF-specific class, **JaxWsProxyFactoryBean**, to create a proxy.

For this SSL/TLS client, the most convenient approach is to use the JAX-WS API to create a proxy without using a WSDL file, as shown in the following Java sample:

```
// Java
import javax.xml.ws.Service;
import org.apache.camel.example.reportincident.ReportIncidentEndpoint;
...
// create the webservice client and send the request
Service s = Service.create(SERVICE_NAME);
s.addPort(
    PORT_NAME,
    "http://schemas.xmlsoap.org/soap/",
    ADDRESS_URL
);
ReportIncidentEndpoint client =
    s.getPort(PORT_NAME, ReportIncidentEndpoint.class);
```



### NOTE

In this example, you *cannot* use the **JaxWsProxyFactoryBean** approach to create a proxy, because a proxy created in this way fails to find the HTTP conduit settings specified in the Spring configuration file.

The **SERVICE\_NAME** and **PORT\_NAME** constants are the QNames of the WSDL service and the WSDL port respectively, as defined in [Example 6.1, “The ReportIncidentEndpointService WSDL Service”](#). The **ADDRESS\_URL** string has the same value as the proxy Web service address and is defined as follows:

```
private static final String ADDRESS_URL =
    "https://localhost:9080/camel-example-cxf-proxy/webservices/incident";
```

In particular, note that the address *must* be defined with the URL scheme, **https**, which selects HTTP over SSL/TLS.

## Steps to add SSL/TLS security to the client

To define a JAX-WS client with SSL/TLS security enabled, perform the following steps:

1. [the section called “Create the Java client as a test case”](#).

2. the section called “Add the http:conduit element to Spring configuration”.
3. the section called “Run the client”.

## Create the Java client as a test case

[Example 6.3, “ReportIncidentRoutesTest Java client”](#) shows the complete code for a Java client that is implemented as a JUnit test case. This client replaces the existing test, `ReportIncidentRoutesTest.java`, in the `src/test/java/org/apache/camel/example/reportincident` sub-directory of the `examples/camel-example-cxf-proxy` demonstration.

To add the client to the `CamelInstallDir/examples/camel-example-cxf-proxy` demonstration, go to the `src/test/java/org/apache/camel/example/reportincident` sub-directory, move the existing `ReportIncidentRoutesTest.java` file to a backup location, then create a new `ReportIncidentRoutesTest.java` file and paste the code from [Example 6.3, “ReportIncidentRoutesTest Java client”](#) into this file.

### Example 6.3. ReportIncidentRoutesTest Java client

```
// Java
package org.apache.camel.example.reportincident;

import org.apache.camel.spring.Main;
import org.apache.cxf.jaxws.JaxWsProxyFactoryBean;
import org.junit.Test;

import java.net.URL;
import javax.xml.namespace.QName;
import javax.xml.ws.Service;

import org.apache.cxf.Bus;
import org.apache.cxf.bus.spring.SpringBusFactory;
import org.apache.camel.example.reportincident.ReportIncidentEndpoint;
import
org.apache.camel.example.reportincident.ReportIncidentEndpointService;

import static org.junit.Assert.assertEquals;

/**
 * Unit test of our routes
 */
public class ReportIncidentRoutesTest {

    private static final QName SERVICE_NAME
        = new QName("http://reportincident.example.camel.apache.org",
"ReportIncidentEndpointService");

    private static final QName PORT_NAME =
        new QName("http://reportincident.example.camel.apache.org",
"ReportIncidentEndpoint");

    private static final String WSDL_URL =
"file:src/main/resources/etc/report_incident.wsdl";
```

```

// should be the same address as we have in our route
private static final String ADDRESS_URL =
"https://localhost:9080/camel-example-cxf-proxy/webservices/incident";

protected SpringBusFactory bf;

protected void startCxfBus() throws Exception {
    bf = new SpringBusFactory();
    Bus bus = bf.createBus("META-INF/spring/cxf-client.xml");
    bf.setDefaultBus(bus);
}

@Test
public void testRendportIncident() throws Exception {
    startCxfBus();
    runTest();
}

protected void runTest() throws Exception {

    // create input parameter
    InputReportIncident input = new InputReportIncident();
    input.setIncidentId("123");
    input.setIncidentDate("2008-08-18");
    input.setGivenName("Claus");
    input.setFamilyName("Ibsen");
    input.setSummary("Bla");
    input.setDetails("Bla bla");
    input.setEmail("davsclaus@apache.org");
    input.setPhone("0045 2962 7576");

    // create the webservice client and send the request
    Service s = Service.create(SERVICE_NAME);
    s.addPort(PORT_NAME, "http://schemas.xmlsoap.org/soap/",
ADDRESS_URL);
    ReportIncidentEndpoint client = s.getPort(PORT_NAME,
ReportIncidentEndpoint.class);

    OutputReportIncident out = client.reportIncident(input);

    // assert we got a OK back
    assertEquals("OK;456", out.getCode());
}
}

```

## Add the http:conduit element to Spring configuration

[Example 6.4, “http:conduit Element with SSL/TLS Enabled”](#) shows the Spring configuration that defines a **http:conduit** element for the **ReportIncidentEndpoint** WSDL port. The **http:conduit** element is configured to enable SSL/TLS security for any client proxies that use the specified WSDL port.

To add the Spring configuration to the client test case, create the **src/test/resources/META-INF/spring** sub-directory, use your favorite text editor to create the file, **cxf-client.xml**, and then paste the contents of [Example 6.4, “http:conduit Element with SSL/TLS Enabled”](#) into the file.

**Example 6.4. http:conduit Element with SSL/TLS Enabled**

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cxf="http://camel.apache.org/schema/cxf"
  xmlns:sec="http://cxf.apache.org/configuration/security"
  xmlns:http="http://cxf.apache.org/transports/http/configuration"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://camel.apache.org/schema/cxf
    http://camel.apache.org/schema/cxf/camel-cxf.xsd
    http://cxf.apache.org/configuration/security
    http://cxf.apache.org/schemas/configuration/security.xsd
    http://cxf.apache.org/transports/http/configuration
    http://cxf.apache.org/schemas/configuration/http-conf.xsd
  ">

  <http:conduit name="
{http://reportincident.example.camel.apache.org}ReportIncidentEndpoint.h
ttp-conduit">
    <http:tlsClientParameters disableCNCheck="true"
secureSocketProtocol="TLSv1">
        <sec:keyManagers keyPassword="ckpass">
            <sec:keyStore password="cspass" type="JKS"
                resource="certs/clientkeystore.jks" />
        </sec:keyManagers>
        <sec:trustManagers>
            <sec:keyStore password="cspass" type="JKS"
                resource="certs/clientkeystore.jks" />
        </sec:trustManagers>
        <sec:cipherSuitesFilter>
            <sec:include>.*_WITH_3DES.*</sec:include>
            <sec:include>.*_WITH_DES.*</sec:include>
            <sec:exclude>.*_WITH_NULL.*</sec:exclude>
            <sec:exclude>.*_DH_anon.*</sec:exclude>
        </sec:cipherSuitesFilter>
    </http:tlsClientParameters>
</http:conduit>

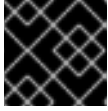
</beans>

```

Please note the following points about the preceding configuration:

- The **http:** and **sec:** namespace prefixes are needed to define the **http:conduit** element. In the **xsi:schemaLocation** element, it is also essential to specify the locations of the corresponding **http://cxf.apache.org/configuration/security** and **http://cxf.apache.org/transports/http/configuration** namespaces.
- The **disableCNCheck** attribute of the **http:tlsClientParameters** element is set to **true**. This means that the client does *not* check whether the Common Name in the server's X.509 certificate matches the server hostname. For more details, see [Appendix A, Managing Certificates](#).



**IMPORTANT**

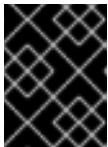
Disabling the CN check is *not* recommended in a production deployment.

- In the `sec:keystore` elements, the certificate locations are specified using the `resource` attribute, which finds the certificates on the classpath. When Maven runs the test, it automatically makes the contents of `src/main/resources` available on the classpath, so that the certificates can be read from the `src/main/resources/certs` directory.

**NOTE**

You also have the option of specifying a certificate location using the `file` attribute, which looks in the filesystem. But the `resource` attribute is more suitable for use with applications packaged in bundles.

- The `sec:cipherSuitesFilter` element is configured to exclude cipher suites matching `.*_WITH_NULL_.*` and `.*_DH_anon_.*`. These cipher suites are effectively incomplete and are *not* intended for normal use.

**IMPORTANT**

It is recommended that you always *exclude* the ciphers matching `.*_WITH_NULL_.*` and `.*_DH_anon_.*`.

- The `secureSocketProtocol` attribute should be set to TLSv1, to match the server protocol and to ensure that the SSLv3 protocol is not used ([POODLE security vulnerability \(CVE-2014-3566\)](#)).

**Run the client**

Because the client is defined as a test case, you can run the client using the standard Maven test goal. To run the client, open a new command window, change directory to `CamelInstallDir/examples/camel-example-cxf-proxy`, and enter the following Maven command:

```
mvn test
```

If the test runs successfully, you should see the following output in the OSGi console window:

```
Incident was 123, changed to 456
Invoked real web service: id=456 by Claus Ibsen
```

# CHAPTER 7. LDAP AUTHENTICATION TUTORIAL

## Abstract

This tutorial explains how to set up an X.500 directory server and configure the OSGi container to use LDAP authentication.

## 7.1. TUTORIAL OVERVIEW

### Goals

In this tutorial you will:

- install Apache Directory Server and Apache Directory Studio
- add user entries into the LDAP server
- add a group to manage security roles
- configure Red Hat JBoss Fuse to use LDAP authentication
- configure JBoss Fuse to use roles for authorization
- configure an instance of Apache ActiveMQ to use LDAP authentication
- configure SSL/TLS connections to the LDAP server

### Tutorial stages

The tutorial consists of the following stages:

1. [Section 7.2, “Set-up a Directory Server and Browser”](#).
2. [Section 7.3, “Add User Entries to the Directory Server”](#).
3. [Section 7.4, “Enable LDAP Authentication in the OSGi Container”](#).
4. [Section 7.5, “Configuring Access to OSGi Administrative Functions”](#).
5. [Section 7.6, “Enable SSL/TLS on the LDAP Connection”](#).

## 7.2. SET-UP A DIRECTORY SERVER AND BROWSER

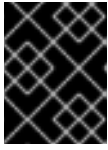
### Overview

In this stage of the tutorial you will install an X.500 directory server and browser client from the *Apache Directory* project. These applications will be used throughout the rest of this tutorial.

### Procedure

To set-up the directory server and browser:

1. Download Apache Directory Server from <http://directory.apache.org/apacheds/1.5/downloads.html>.
2. Run the downloaded installer.



### IMPORTANT

During the installation process, you will be asked whether or not to install a *default instance* of the directory server. Choose the default instance.

3. Start the directory service as described in [the section called “Starting Apache Directory Server”](#).
4. Install Apache Directory Studio as described in [the section called “Install Apache Directory Studio”](#).
5. Start Apache Directory Studio.
  - If you installed the standalone version of Apache Directory Studio, double-click the relevant icon to launch the application.
  - If you installed Apache Directory Studio into an existing Eclipse IDE:
    1. Start Eclipse.
    2. Select **Window** → **Open Perspective** → **Other**.
    3. In the **Open Perspective** dialog, select **LDAP**.
    4. Click **OK**.
6. Connect the browser to the server as described in [the section called “Connecting the browser to the server”](#).

## Starting Apache Directory Server

If you install Apache Directory Server on Windows, the default instance of the directory server is configured as a Windows service. Hence, you can stop and start the directory server using the standard **Services** administrative tool.

If you install on a Linux or Mac OS platform, follow the instructions in [Installing and Starting the Server](#) for starting and stopping the directory server.

## Install Apache Directory Studio

The Apache Directory Studio is an Eclipse-based suite of tools for administering an X.500 directory server. In particular, for this tutorial, you need the LDAP Browser feature, which enables you to create new entries in the Directory Information Tree (DIT).

There are two alternative ways of installing Apache Directory Studio:

- *Standalone application*—download the standalone distribution from the [Directory Studio downloads](#) page and follow the installation instructions from the [Apache Directory Studio User Guide](#).

- *Eclipse plug-in*—if you already use Eclipse as your development environment, you can install *Apache Directory Studio* as a set of Eclipse plug-ins. The only piece of *Apache Directory Studio* that you need for this tutorial is the *LDAP Browser* plug-in.

To install the LDAP Browser as an Eclipse plug-in, follow the install instructions from the [LDAP Browser Plug-In User Guide](#).

## Connecting the browser to the server

To connect the LDAP browser to the LDAP server:

1. Right-click inside the **Connections** view.
2. Select **New Connection....**

The **New LDAP Connection** wizard opens.

3. In the **Connection name** field, enter **Apache Directory Server**.
4. In the **Hostname** field enter **localhost**.
5. In the **Port** field, enter **10389**.

Figure 7.1. New LDAP Connection Wizard

**New LDAP Connection**

**Network Parameter**  
Please enter connection name and network parameters.

Connection name: Apache Directory Server

Network Parameter

Hostname: localhost

Port: 10389

Encryption method: No encryption

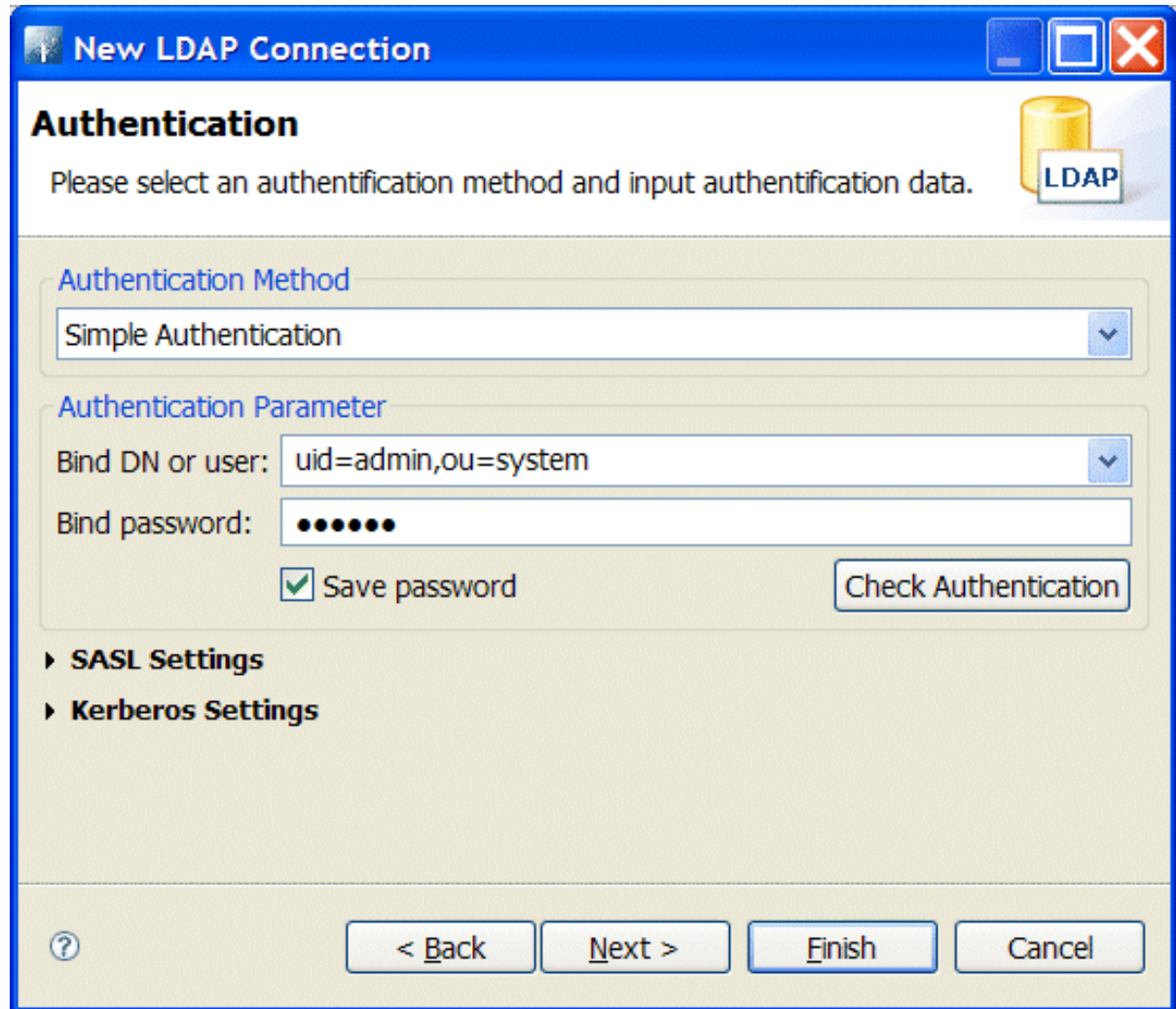
Check Network Parameter

? < Back Next > Finish Cancel

6. Click **Next**.
7. In the **Bind DN or user** field, enter **uid=admin,ou=system**).

- In the **Bind password** field, enter **secret**).

**Figure 7.2. Authentication Step of New LDAP Connection**



- Click **Finish**.

If the connection is successfully established, you should see an outline of the Directory Information Tree (DIT) in the **LDAP Browser** view.

## 7.3. ADD USER ENTRIES TO THE DIRECTORY SERVER

### Overview

The basic prerequisite for using LDAP authentication with the OSGi container is to have an X.500 directory server running and configured with a collection of user entries. For many use cases, you will also want to configure a number of groups to manage user roles.

### Goals

In this portion of the tutorial you will

- [add three user entries to the LDAP server](#)
- [add four groups to the LDAP server](#)

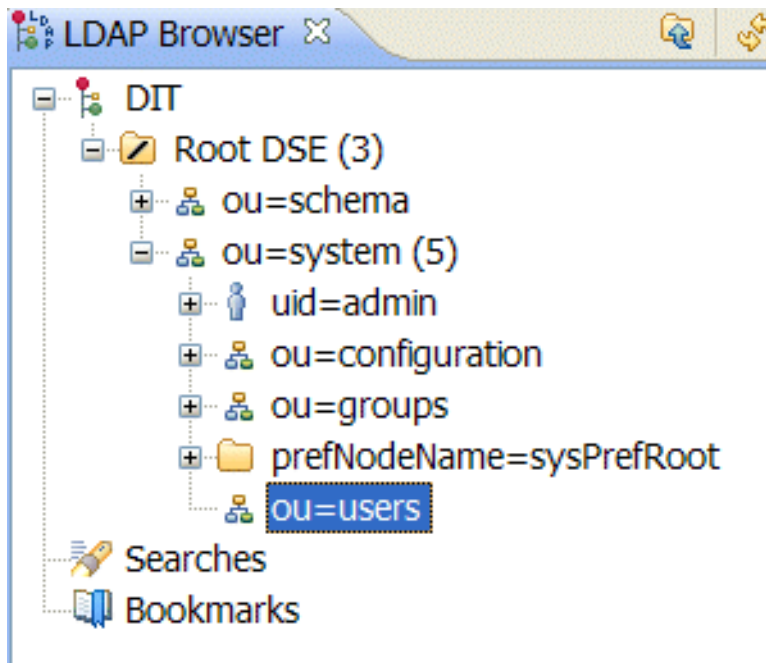
## Adding user entries

Perform the following steps to add user entries to the directory server:

1. Ensure that the LDAP server and browser are running.

See [Section 7.2, “Set-up a Directory Server and Browser”](#).

2. In the **LDAP Browser** view, drill down to the **ou=users** node.



3. Select the **ou=users** node.
4. Open the context menu.
5. Select **New** → **New Entry**.

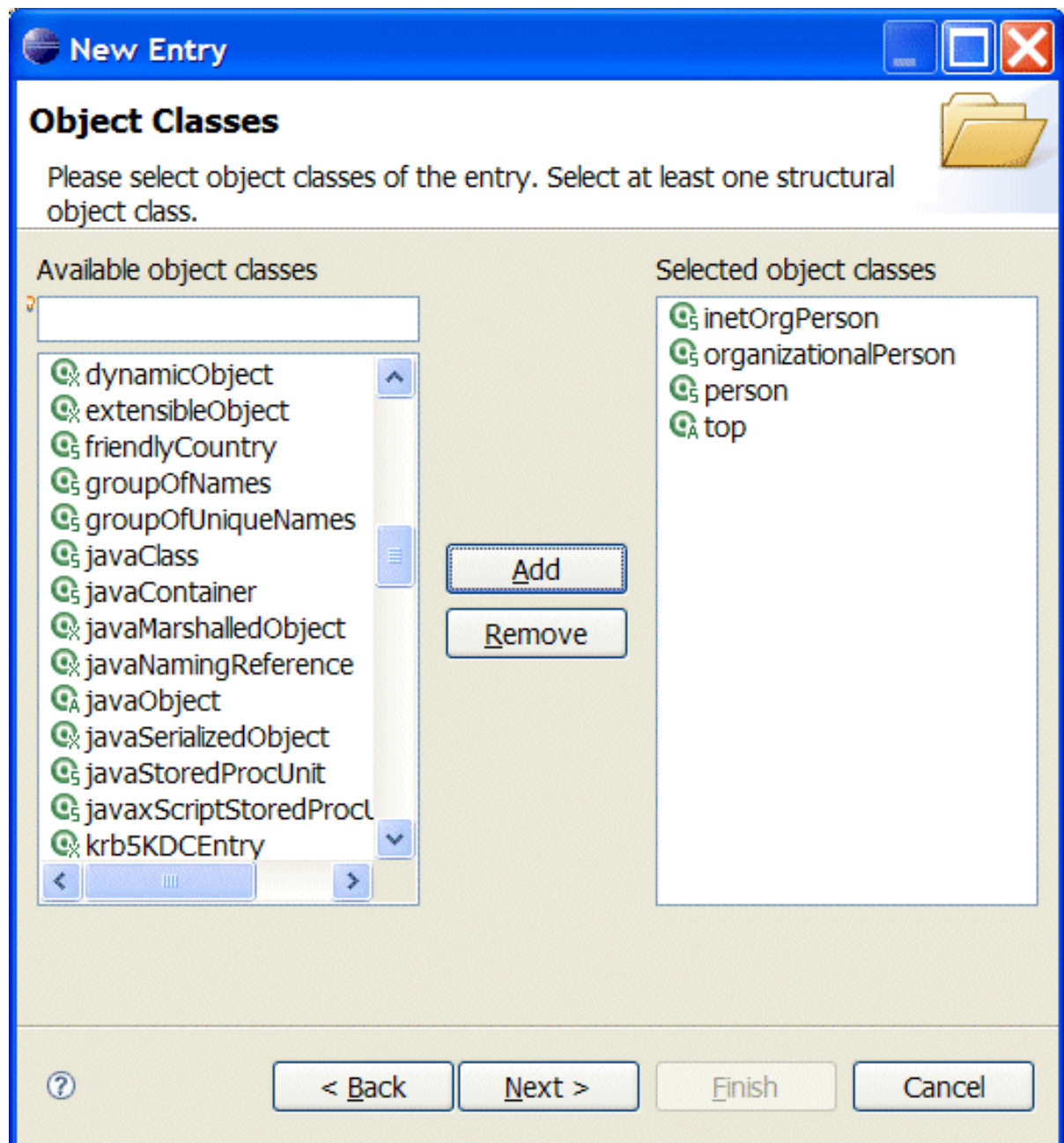
The **New Entry** wizard appears.

6. In the **Entry Creation Method** pane, check **Create entry from scratch**.
7. Click **Next**.

The **Object Classes** pane opens.

8. In the **Object Classes** pane, select **inetOrgPerson** from the list of **Available object classes** on the left.
9. Click **Add** to populate the list of **Selected object classes**.

Figure 7.3. New Entry Wizard



10. Click **Next**.

The **Distinguished Name** pane opens.

11. In the the **RDN** field, enter **uid** in front and **jdoe** after the equals sign.

Figure 7.4. Distinguished Name Step of New Entry Wizard

**New Entry**

### Distinguished Name

Please select the parent of the new entry and enter the RDN.

Parent:

RDN:  =

DN Preview:

12. Click **Next**.

The **Attributes** pane opens.

13. Fill in the remaining mandatory attributes in the **Attributes** pane.
  - a. Set the **cn** (common name) attribute to **John Doe**
  - b. Set the **sn** (surname) attribute to **Doe**.



Figure 7.5. Attributes Step of New Entry Wizard

**Attributes**

Please enter the attributes for the entry. Enter at least the MUST attributes.

DN: uid=jdoe,ou=users,ou=system

Attribute Description	Value
<i>objectClass</i>	<i>inetOrgPerson (structural)</i>
<i>objectClass</i>	<i>organizationalPerson (structural)</i>
<i>objectClass</i>	<i>person (structural)</i>
<i>objectClass</i>	<i>top (abstract)</i>
<b>cn</b>	<b>John Doe</b>
<b>sn</b>	<b>Doe</b>
uid	jdoe

< Back   Next >   Finish   Cancel

14. Add a **userPassword** attribute to the user entry.
  - a. Open the context menu in the **Attributes** pane.
  - b. Select **New Attribute**.
 

The **New Attribute** wizard appears.
  - c. From the **Attribute type** drop-down list, select **userPassword**.
  - d. Click **Finish**.
 

The **Password Editor** dialog appears.
  - e. In the **Enter New Password** field, enter the password, **secret**.
  - f. Click **OK**.

The **userPassword** attribute will appear in the attributes editor.

15. Click **Finish**.

16. Add a user **Jane Doe** by following [Step 3](#) to [Step 15](#).

In [Step 11](#), use **janedoe** for the new user's **uid**.

17. Add a user **Camel Rider** by following [Step 3](#) to [Step 15](#).

In [Step 11](#), use **crider** for the new user's **uid**.

## Adding groups for the roles

To add the groups that define the roles:

1. Create a new organizational unit to contain the role groups.

a. In the **LDAP Browser** view, select the **ou=system** node.

b. Open the context menu.

c. Select **New** → **New Entry**.

The **New Entry** wizard appears.

d. In the **Entry Creation Method** pane, check **Create entry from scratch**.

e. Click **Next**.

The **Object Classes** pane opens.

f. Select **organizationalUnit** from the list of **Available object classes** on the left.

g. Click **Add** to populate the list of **Selected object classes**.

h. Click **Next>**.

The **Distinguished Name** pane opens.

i. In the **the RDN** field, enter **ou** in front and **roles** after the equals sign.

j. Click **Next>**.

The **Attributes** pane opens.

k. Click **Finish**.



### NOTE

This step is required because Apache DS only allows administrators access to entries in **ou=system,ou=groups**.

2. In the **LDAP Browser** view, drill down to the **ou=roles** node.

3. Select the **ou=roles** node.
4. Open the context menu.
5. Select **New** → **New Entry**.

The **New Entry** wizard appears.

6. In the **Entry Creation Method** pane, check **Create entry from scratch**.
7. Click **Next**.

The **Object Classes** pane opens.

8. Select **groupOfNames** from the list of **Available object classes** on the left.
9. Click **Add** to populate the list of **Selected object classes**.
10. Click **Next**.

The **Distinguished Name** pane opens.

11. In the the **RDN** field, enter **cn** in front and **admin** after the equals sign.
12. Click **Next**.

The **Attributes** pane opens and you are presented with a DN editor.

13. Enter **uid=jdoe**.
14. Click **OK**.
15. Click **Finish**.
16. Add a **sshConsole** role by following [Step 3](#) to [Step 15](#).

In [Step 11](#), use **sshConsole** for the new group's **cn**.

In [Step 13](#), use **uid=janedoe**.

17. Add a **webconsole** role by following [Step 3](#) to [Step 15](#).

In [Step 11](#), use **webconsole** for the new group's **cn**.

In [Step 13](#), use **uid=janedoe**.

18. Add a **jmxUser** role by following [Step 3](#) to [Step 15](#).

In [Step 11](#), use **jmxUser** for the new group's **cn**.

In [Step 13](#), use **uid=crider**.

## 7.4. ENABLE LDAP AUTHENTICATION IN THE OSGI CONTAINER

### Overview

In this part of the tutorial you will configure an LDAP realm in the OSGi container. The new realm overrides the default karaf realm, so that the container authenticates credentials based on user entries stored in the X.500 directory server.

## Procedure for standalone OSGi container

To enable LDAP authentication in a standalone OSGi container:

1. Ensure that the X.500 directory server is running.
2. Start Red Hat JBoss Fuse by entering the following command in a terminal window:

```
fuse
```

3. Create a Blueprint configuration file called `ldap-module.xml`.
4. Copy [Example 7.1, “JAAS Realm for Standalone”](#) into `ldap-module.xml`.

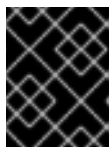
### Example 7.1. JAAS Realm for Standalone

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0">

  <jaas:config name="karaf" rank="1">
    <jaas:module
      className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
        flags="required">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      connection.username=uid=admin,ou=system
      connection.password=secret
      connection.protocol=
      connection.url=ldap://localhost:10389
      user.base.dn=ou=users,ou=system
      user.filter=(uid=%u)
      user.search.subtree=true
      role.base.dn=ou=roles,ou=system
      role.name.attribute=cn
      role.filter=(member=uid=%u)
      role.search.subtree=true
      authentication=simple
    </jaas:module>
  </jaas:config>
</blueprint>
```

This login module creates a JAAS realm called **karaf**, which is the same name as the default JAAS realm used by Red Hat JBoss Fuse. By redefining this realm with a **rank** attribute value greater than **0**, it overrides the standard **karaf** realm which has the rank **0**. For more information on configuring a JAAS realm see [Section 2.1.2, “Defining JAAS Realms”](#).

For a detailed description of configuring JBoss Fuse to use LDAP see [Section 2.2, “Enabling LDAP Authentication”](#).



### IMPORTANT

When setting the JAAS properties above, do *not* enclose the property values in double quotes.

### TIP

If you use OpenLDAP, the syntax of the role filter is **(member:=uid=%u)**.

5. To deploy the new LDAP module, copy the **ldap-module.xml** into the JBoss Fuse **deploy/** directory.

The LDAP module is automatically activated.

## Procedure for a Fabric

To enable LDAP authentication in a Fabric (affecting all of the containers in the current fabric):

1. Ensure that the X.500 directory server is running.
2. If your local Fabric container is not already running, start it now, by entering the following command in a terminal window:

```
./fuse
```



### NOTE

If the Fabric container you want to connect to is running on a remote host, you can connect to it using the **client** command-line utility in the **InstallDir/bin** directory.

3. Create a new version of the Fabric profile data, by entering the following console command:

```
JBossFuse:karaf@root> version-create
Created version: 1.1 as copy of: 1.0
```



### NOTE

In effect, this command creates a new branch named **1.1** in the Git repository underlying the ZooKeeper registry.

4. Create the new profile resource, **ldap-module.xml** (a Blueprint configuration file), in version **1.1** of the **default** profile, as follows:

```
JBossFuse:karaf@root> profile-edit --resource ldap-module.xml
default 1.1
```

The built-in profile editor opens automatically, which you can use to edit the contents of the `ldap-module.xml` resource.

5. Copy [Example 7.2, “JAAS Realm for Fabric”](#) into the `ldap-module.xml` resource, customizing the value of the `rank` attribute and the `connection.url` property, as necessary.

### Example 7.2. JAAS Realm for Fabric

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0">

  <jaas:config name="karaf" rank="2">
    <jaas:module
      className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
      flags="required">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      connection.username=uid=admin,ou=system
      connection.password=secret
      connection.protocol=
      connection.url=ldap://LDAPHost:10389
      user.base.dn=ou=users,ou=system
      user.filter=(uid=%u)
      user.search.subtree=true
      role.base.dn=ou=roles,ou=system
      role.name.attribute=cn
      role.filter=(member=uid=%u)
      role.search.subtree=true
      authentication=simple
    </jaas:module>
  </jaas:config>
</blueprint>
```

Where **LDAPHost** is the name of the host where the LDAP server is running. You must be sure to use a hostname that is accessible to all of the containers in the fabric (for example, you cannot use `localhost` as the hostname here).

Save and close the `ldap-module.xml` resource by typing Ctrl-S and Ctrl-X.

This login module creates a JAAS realm called **karaf**, which is the same name as the default JAAS realm used by Red Hat JBoss Fuse. By redefining this realm with a **rank** of **2**, it overrides all of the previously installed **karaf** realms. For more information on configuring a JAAS realm see [Section 2.1.2, “Defining JAAS Realms”](#).

For a detailed description of configuring JBoss Fuse to use LDAP see [Section 2.2, “Enabling LDAP Authentication”](#).

**IMPORTANT**

Pay particular attention to the value of the **rank** to ensure that it is higher than all previously installed **karaf** realms. If the **rank** is not sufficiently high, the new realm will not be used by the fabric.

**IMPORTANT**

When setting the JAAS properties above, do *not* enclose the property values in double quotes.

**TIP**

If you use OpenLDAP, the syntax of the role filter is (**member:=uid=%u**).

6. Edit the agent properties of version 1.1 of the **default** profile, adding an instruction to deploy the Blueprint resource file defined in the previous step. Enter the following console command:

```
JBossFuse:karaf@root> profile-edit default 1.1
```

The built-in profile editor opens automatically. Add the following line to the agent properties:

```
bundle.ldap-realm=blueprint:profile:ldap-module.xml
```

Save and close the agent properties by typing Ctrl-S and Ctrl-X.

7. The new LDAP realm is not activated, until you upgrade a container to use the new version, **1.1**. To activate LDAP on a *single* container (for example, on a container called **root**), enter the following console command:

```
JBossFuse:karaf@root> container-upgrade 1.1 root
```

To activate LDAP on *all* containers in the fabric, enter the following console command:

```
JBossFuse:karaf@root> container-upgrade --all 1.1
```

**IMPORTANT**

It is advisable to upgrade just a single container initially, to make sure that everything is working properly. This is particularly important, if you have only remote access to the fabric: if you upgrade all of the containers at once, you might not be able to reconnect to the fabric.

## Test the LDAP authentication

Test the new LDAP realm by connecting to the running container using the Red Hat JBoss Fuse **client** utility, as follows:

1. Open a new command prompt.
2. change directory to the JBoss Fuse **InstallDir/bin** directory.

3. Enter the following command to log on to the running container instance using the identity **janedoe**:

```
client -u janedoe -p secret
```

You should receive the following message:

```
Authentication failure
```

This fails because **janedoe** does not have the **admin** role which is required for using the remote console.

4. Enter the following command to log on to the running container instance using the identity **jdoue**:

```
client -u jdoue -p secret
```

You should successfully log into the container's remote console because **jdoue** does have the **admin** role.

5. Log off the remote console by entering the **logout** command.

## 7.5. CONFIGURING ACCESS TO OSGI ADMINISTRATIVE FUNCTIONS

### Overview

This tutorial explains how to configure the OSGi administrative functions to use specific roles for authorization. By configuring each of the administrative functions to use a different role for access, you can provide fine grained control over who can monitor and manipulate running containers.

When LDAP is enabled, the OSGi container expects the user role data to be stored along with the user authentication data in the LDAP directory server. The LDAP search query to extract the role data is specified by the **role.\*** properties in the **jaas:module** element.

The JAAS LDAP login module used in this tutorial, shown in [Example 7.1, "JAAS Realm for Standalone"](#), is configured to extract the role name from the **cn** property of all entries selected by the filter **member=uid=%u** which is run on the tree selected using the base DN **uo=roles,ou=system**. In [the section called "Adding groups for the roles"](#), you added three groups to the **uo=roles,ou=system** tree. The filter will match with any group that has a member specified by **uid=%u**.

For example, when you attempted to connect to the remote console as user **jdoue** the filter searched for a group with a member **uid=jdoue** and matched on the group **cn=admin,uo=roles,ou=system**. The LDAP module extracted the **cn** property's value of **admin** and used it as the role for authorizing user **jdoue**.

### Goals

You will change the role used for each of the administrative functions:

- [SSH \(remote console login\)](#)
- [JMX management](#)



## Prerequisites

Before you can perform any of the following tutorials, you must ensure that the ApacheDS server is running.

## Configure a role for the remote console

To configure a role for the remote console:

1. Open ***InstallDir/etc/org.apache.karaf.shell.cfg*** in a text editor.
2. Add the following line:

```
sshRole=sshConsole
```

3. Save the changes.
4. Start Red Hat JBoss Fuse by entering the following command in a terminal window:

```
> fuse
```

5. Open a new command prompt.
6. Change directory to the JBoss Fuse install directory.
7. Enter the following command to log on to the running container instance using the identity **janedoe**:

```
client -u janedoe -p secret
```

You should successfully log into the container's remote console because **janedoe** does have the **sshConsole** role.

## Configure a role for JMX access

To configure a role for JMX access:

1. Open ***InstallDir/etc/org.apache.karaf.management.cfg*** in a text editor.
2. Add the following line:

```
jmxRole=jmxUser
```

3. Save the changes.
4. Start JBoss Fuse by entering the following command in a terminal window:

```
> fuse
```

5. Start JConsole or another JMX console.
6. Connect to JBoss Fuse's JMX server using the following settings:

- JMX URL: `service:jmx:rmi:///jndi/rmi://localhost:1099/karaf-root`
- User: `jdoe`
- Password: `secret`

The connection will fail because `jdoe` user does not have the `jmxUser` role.

7. Connect to JBoss Fuse's JMX server as using the following settings:

- JMX URL: `service:jmx:rmi:///jndi/rmi://localhost:1099/karaf-root`
- User: `crider`
- Password: `secret`

The connection will succeed because `crider` user does have the `jmxUser` role.

## More information

See the following links for more information:

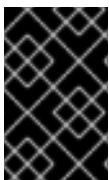
- [chapter "Configuring JMX" in "Configuring and Running Red Hat JBoss Fuse"](#) for details about JMX,
- [Section 2.2, "Enabling LDAP Authentication"](#) for configuring the JBoss Fuse LDAP login module,
- [Section 2.4, "Configuring Roles for the Administrative Protocols"](#) for configuring the JBoss Fuse administrative functions.

## 7.6. ENABLE SSL/TLS ON THE LDAP CONNECTION

### Overview

This tutorial explains how to enable SSL/TLS security on the connection between the LDAP login module and the Apache Directory Server.

The Apache Directory Server is already configured with an SSL endpoint. The default configuration creates an LDAPS endpoint that listens on the IP port `10636`. The directory server automatically generates a self-signed X.509 certificate which it uses to identify itself during the SSL/TLS handshake.



### IMPORTANT

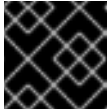
You can use the default SSL configuration for simple demonstrations, but it is *not* suitable for real deployments. For advice on how to configure a real deployment, see [the section called "Tightening up security"](#).

### Procedure

To enable SSL/TLS security on the connection to the Apache Directory Server:

1. Obtain a copy of the server's self-signed certificate.
  - a. Using a Web browser , navigate to the following URL:

https://localhost:10636

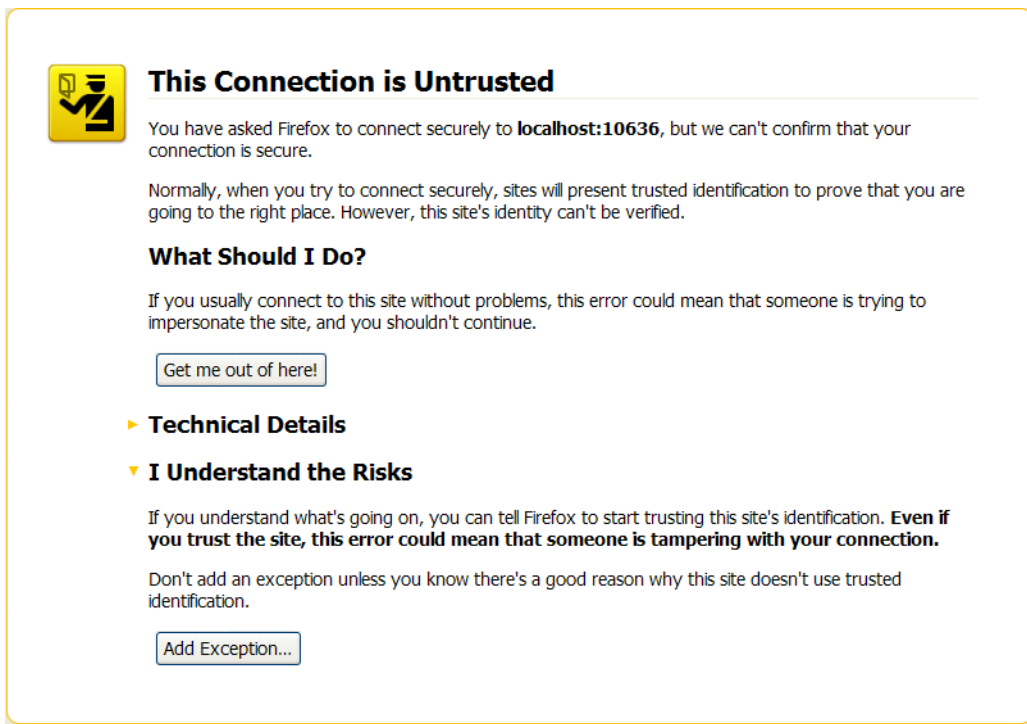


## IMPORTANT

Remember to specify the scheme as **https**, not just **http**.

The Web browser now signals an error, because the certificate it receives from the server is untrusted. In the case of Firefox, you will see the following error in the browser window:

**Figure 7.6. Obtaining the Certificate**



b. Click **I Understand the Risks**.

c. Click **Add Exception**.

The **Add Security Exception** dialog opens.

d. In the **Add Security Exception** dialog, click **Get Certificate**.

e. Click **View**.

The **Certificate Viewer** dialog opens.

f. In the **Certificate Viewer** dialog, select the **Details** tab.

g. Click **Export**.

The **Save Certificate To File** dialog opens.

h. In the **Save Certificate To File** dialog, use the drop-down list to set the **Save as type** to X.509 Certificate (DER).

i. Save the certificate, **ApacheDS.der**, to a convenient location on the filesystem.

2. Convert the DER format certificate into a keystore.
  - a. From a command prompt, change directory to the directory where you have stored the **ApacheDS.der** file.
  - b. Enter the following **keytool** command:
 

```
keytool -import -file ApacheDS.der -alias server -keystore truststore.ks -storepass secret
```
3. Copy the newly created keystore file, **truststore.ks**, into the Red Hat JBoss Fuse **etc/** directory.
4. Open the **ldap-module.xml** file you created in [Section 7.4, "Enable LDAP Authentication in the OSGi Container"](#) in a text editor.
5. Edit the connection.url to use ldaps://localhost:10636.
6. Add the highlighted lines in [Example 7.3, "LDAP Configuration for Using SSL/TLS"](#).

### Example 7.3. LDAP Configuration for Using SSL/TLS

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:jaas="http://karaf.apache.org/xmlns/jaas/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0">

  <!-- Example configuration for using LDAP based authentication.
  This example uses an JAAS LoginModule from Karaf.
  It supports authentication of users and also supports
  retrieving user roles for authorization.

  Note, this config overwrite the default karaf domain
  that is defined inside some JAR file
  by using a rank > 0 attribute.
  -->
  <jaas:config name="karaf" rank="1">
    <jaas:module
      className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
      flags="required">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      connection.username=uid=admin,ou=system
      connection.password=secret
      connection.protocol=
      connection.url = ldaps://localhost:10636
      user.base.dn = ou=users,ou=system
      user.filter = (uid=%u)
      user.search.subtree = true
      role.base.dn = ou=users,ou=system
      role.filter = (uid=%u)
      role.name.attribute = ou
      role.search.subtree = true
      authentication = simple
      ssl.protocol=TLSv1
    </jaas:module>
  </jaas:config>
</blueprint>
```

```

        ssl.truststore=truststore
        ssl.algorithm=PKIX
    </jaas:module>
</jaas:config>

    <jaas:keystore name="truststore"
        path="file:///InstallDir/etc/truststore.ks"
        keystorePassword="secret" />
</blueprint>

```

7. Copy the `ldap-module.xml` file into the Red Hat JBoss Fuse `deploy/` directory.

The LDAP module is automatically activated.

8. Test the new LDAP realm by connecting to the running container using the JBoss Fuse client utility.
  - a. Open a new command prompt.
  - b. Change to the JBoss Fuse install directory.
  - c. Enter the following command to log on to the running container instance using the identity `jdope`:

```
client -u jdope -p secret
```

You should successfully log into the container's remote console because `jdope` does have the `admin` role.

## Tightening up security

The SSL set-up described here is suitable *only* as a proof-of-concept demonstration. For a real deployment, you must make the following changes to tighten up security:

- Delete all entries from the Red Hat JBoss Fuse's `etc/users.properties` file.

If the `ldap-module.xml` bundle fails to start up properly, JAAS authentication reverts to the built-in file-based `karaf` realm, which takes its user data from the `users.properties` file.

- Disable the insecure LDAP endpoint on the Apache Directory Server.
- Create and deploy a properly signed X.509 certificate on the Apache Directory Server.

See [Appendix A, Managing Certificates](#).

- Make sure that the LDAP server is configured to use the TLSv1 protocol (POODLE vulnerability). Do *not* enable the SSLv3 protocol. For more information, see [Poodle vulnerability \(CVE-2014-3566\)](#).

## Apache Directory Server Reference

For more details of how to configure SSL/TLS security on the Apache Directory Server, see [How to enable SSL](#).

## APPENDIX A. MANAGING CERTIFICATES

### Abstract

TLS authentication uses X.509 certificates—a common, secure and reliable method of authenticating your application objects. You can create X.509 certificates that identify your Apache ActiveMQ applications.

### A.1. WHAT IS AN X.509 CERTIFICATE?

#### Role of certificates

An X.509 certificate binds a name to a public key value. The role of the certificate is to associate a public key with the identity contained in the X.509 certificate.

#### Integrity of the public key

Authentication of a secure application depends on the integrity of the public key value in the application's certificate. If an impostor replaces the public key with its own public key, it can impersonate the true application and gain access to secure data.

To prevent this type of attack, all certificates must be signed by a *certification authority* (CA). A CA is a trusted node that confirms the integrity of the public key value in a certificate.

#### Digital signatures

A CA signs a certificate by adding its *digital signature* to the certificate. A digital signature is a message encoded with the CA's private key. The CA's public key is made available to applications by distributing a certificate for the CA. Applications verify that certificates are validly signed by decoding the CA's digital signature with the CA's public key.



#### WARNING

The supplied demonstration certificates are self-signed certificates. These certificates are insecure because anyone can access their private key. To secure your system, you must create new certificates signed by a trusted CA.

#### Contents of an X.509 certificate

An X.509 certificate contains information about the certificate subject and the certificate issuer (the CA that issued the certificate). A certificate is encoded in Abstract Syntax Notation One (ASN.1), a standard syntax for describing messages that can be sent or received on a network.

The role of a certificate is to associate an identity with a public key value. In more detail, a certificate includes:

- A *subject distinguished name* (DN) that identifies the certificate owner.

- The *public key* associated with the subject.
- X.509 version information.
- A *serial number* that uniquely identifies the certificate.
- An *issuer DN* that identifies the CA that issued the certificate.
- The digital signature of the issuer.
- Information about the algorithm used to sign the certificate.
- Some optional X.509 v.3 extensions; for example, an extension exists that distinguishes between CA certificates and end-entity certificates.

## Distinguished names

A DN is a general purpose X.500 identifier that is often used in the context of security.

See [Appendix B, ASN.1 and Distinguished Names](#) for more details about DNs.

## A.2. CERTIFICATION AUTHORITIES

A CA consists of a set of tools for generating and managing certificates and a database that contains all of the generated certificates. When setting up a system, it is important to choose a suitable CA that is sufficiently secure for your requirements.

There are two types of CA you can use:

- [commercial CAs](#) are companies that sign certificates for many systems.
- [private CAs](#) are trusted nodes that you set up and use to sign certificates for your system only.

### A.2.1. Commercial Certification Authorities

#### Signing certificates

There are several commercial CAs available. The mechanism for signing a certificate using a commercial CA depends on which CA you choose.

#### Advantages of commercial CAs

An advantage of commercial CAs is that they are often trusted by a large number of people. If your applications are designed to be available to systems external to your organization, use a commercial CA to sign your certificates. If your applications are for use within an internal network, a private CA might be appropriate.

#### Criteria for choosing a CA

Before choosing a commercial CA, consider the following criteria:

- What are the certificate-signing policies of the commercial CAs?
- Are your applications designed to be available on an internal network only?

- What are the potential costs of setting up a private CA compared to the costs of subscribing to a commercial CA?

## A.2.2. Private Certification Authorities

### Choosing a CA software package

If you want to take responsibility for signing certificates for your system, set up a private CA. To set up a private CA, you require access to a software package that provides utilities for creating and signing certificates. Several packages of this type are available.

### OpenSSL software package

One software package that allows you to set up a private CA is OpenSSL, <http://www.openssl.org>. OpenSSL is derived from SSLeay, an implementation of SSL developed by Eric Young ([eyay@cryptsoft.com](mailto:eyay@cryptsoft.com)). The OpenSSL package includes basic command line utilities for generating and signing certificates. Complete documentation for the OpenSSL command line utilities is available at <http://www.openssl.org/docs>.

### Setting up a private CA using OpenSSL

To set up a private CA, see the instructions in [Section A.5, “Creating Your Own Certificates”](#).

### Choosing a host for a private certification authority

Choosing a host is an important step in setting up a private CA. The level of security associated with the CA host determines the level of trust associated with certificates signed by the CA.

If you are setting up a CA for use in the development and testing of Apache ActiveMQ applications, use any host that the application developers can access. However, when you create the CA certificate and private key, do not make the CA private key available on any hosts where security-critical applications run.

### Security precautions

If you are setting up a CA to sign certificates for applications that you are going to deploy, make the CA host as secure as possible. For example, take the following precautions to secure your CA:

- Do not connect the CA to a network.
- Restrict all access to the CA to a limited set of trusted users.
- Use an RF-shield to protect the CA from radio-frequency surveillance.

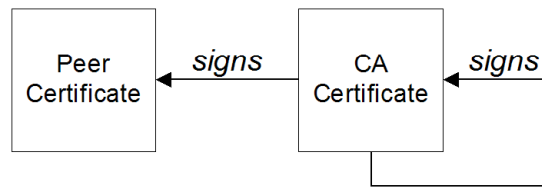
## A.3. CERTIFICATE CHAINING

### Certificate chain

A *certificate chain* is a sequence of certificates, where each certificate in the chain is signed by the subsequent certificate.

[Figure A.1, “A Certificate Chain of Depth 2”](#) shows an example of a simple certificate chain.



**Figure A.1. A Certificate Chain of Depth 2**

### Self-signed certificate

The last certificate in the chain is normally a *self-signed certificate*—a certificate that signs itself.

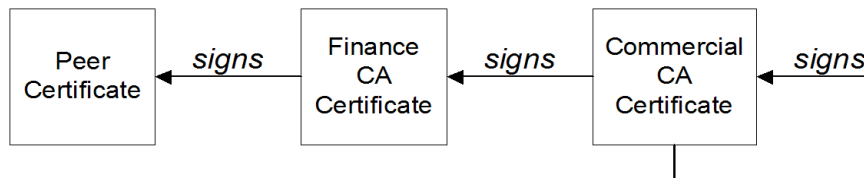
### Chain of trust

The purpose of a certificate chain is to establish a chain of trust from a peer certificate to a trusted CA certificate. The CA vouches for the identity in the peer certificate by signing it. If the CA is one that you trust (indicated by the presence of a copy of the CA certificate in your root certificate directory), this implies you can trust the signed peer certificate as well.

### Certificates signed by multiple CAs

A CA certificate can be signed by another CA. For example, an application certificate could be signed by the CA for the finance department of Progress Software, which in turn is signed by a self-signed commercial CA.

Figure A.2, “A Certificate Chain of Depth 3” shows what this certificate chain looks like.

**Figure A.2. A Certificate Chain of Depth 3**

### Trusted CAs

An application can accept a peer certificate, provided it trusts at least one of the CA certificates in the signing chain.

## A.4. SPECIAL REQUIREMENTS ON HTTPS CERTIFICATES

### Overview

The HTTPS specification mandates that HTTPS clients must be capable of verifying the identity of the server. This can potentially affect how you generate your X.509 certificates. The mechanism for verifying the server identity depends on the type of client. Some clients might verify the server identity by accepting only those server certificates signed by a particular trusted CA. In addition, clients can inspect the contents of a server certificate and accept only the certificates that satisfy specific constraints.

In the absence of an application-specific mechanism, the HTTPS specification defines a generic mechanism, known as the *HTTPS URL integrity check*, for verifying the server identity. This is the standard mechanism used by Web browsers.

## HTTPS URL integrity check

The basic idea of the URL integrity check is that the server certificate's identity must match the server host name. This integrity check has an important impact on how you generate X.509 certificates for HTTPS: *the certificate identity (usually the certificate subject DN's common name) must match the host name on which the HTTPS server is deployed.*

The URL integrity check is designed to prevent *man-in-the-middle* attacks.

## Reference

The HTTPS URL integrity check is specified by RFC 2818, published by the Internet Engineering Task Force (IETF) at <http://www.ietf.org/rfc/rfc2818.txt>.

## How to specify the certificate identity

The certificate identity used in the URL integrity check can be specified in one of the following ways:

- [Using commonName](#)
- [Using subjectAltName](#)

## Using commonName

The usual way to specify the certificate identity (for the purpose of the URL integrity check) is through the Common Name (CN) in the subject DN of the certificate.

For example, if a server supports secure TLS connections at the following URL:

```
https://www.redhat.com/secure
```

The corresponding server certificate would have the following subject DN:

```
C=IE,ST=Co. Dublin,L=Dublin,O=RedHat,  
OU=System,CN=www.redhat.com
```

Where the CN has been set to the host name, **www.redhat.com**.

For details of how to set the subject DN in a new certificate, see [the section called "Generate a certificate and private key pair"](#).

## Using subjectAltName (multi-homed hosts)

Using the subject DN's Common Name for the certificate identity has the disadvantage that only *one* host name can be specified at a time. If you deploy a certificate on a multi-homed host, however, you might find it is practical to allow the certificate to be used with *any* of the multi-homed host names. In this case, it is necessary to define a certificate with multiple, alternative identities, and this is only possible using the **subjectAltName** certificate extension.

For example, if you have a multi-homed host that supports connections to either of the following host names:

```
www.redhat.com  
www.jboss.org
```

Then you can define a **subjectAltName** that explicitly lists both of these DNS host names. If you generate your certificates using the **openssl** utility, edit the relevant line of your **openssl.cnf** configuration file to specify the value of the **subjectAltName** extension, as follows:

```
subjectAltName=DNS:www.redhat.com,DNS:www.jboss.org
```

Where the HTTPS protocol matches the server host name against either of the DNS host names listed in the **subjectAltName** (the **subjectAltName** takes precedence over the Common Name).

The HTTPS protocol also supports the wildcard character, **\***, in host names. For example, you can define the **subjectAltName** as follows:

```
subjectAltName=DNS:*.jboss.org
```

This certificate identity matches any three-component host name in the domain **jboss.org**.



### WARNING

You must *never* use the wildcard character in the domain name (and you must take care never to do this accidentally by forgetting to type the dot, **.**, delimiter in front of the domain name). For example, if you specified **\*jboss.org**, your certificate could be used on *any* domain that ends in the letters **jboss**.

## A.5. CREATING YOUR OWN CERTIFICATES

### Overview

If you choose to use a private CA you will need to generate your own certificates for your applications to use. The OpenSSL project provides free command-line utilities for setting up a private CA, creating signed certificates, and adding the CA to your Java keystore.

### OpenSSL utilities

You can download the OpenSSL utilities from <http://openssl.org/>.

This section describes using the OpenSSL command-line utilities to create certificates. Further documentation of the OpenSSL command-line utilities can be obtained at <http://www.openssl.org/docs>.

### Procedure

To create your own CA and certificates:

1. Add the OpenSSL **bin** directory to your path.
2. Create your own private CA.
  - a. Create the directory structure for the CA.

The directory structure should be:

- **X509CA/ca**
- **X509CA/certs**
- **X509CA/newcerts**
- **X509CA/crl**

Where **X509CA** is the name of the CA's home directory.

- b. Copy the **openssl.cnf** file from your OpenSSL installation to your **X509CA** directory.
- c. Open your copy of **openssl.cnf** in a text editor.
- d. Edit the [**CA\_default**] section to look like [Example A.1, “OpenSSL Configuration”](#).

### Example A.1. OpenSSL Configuration

```
#####
[ CA_default ]

dir            = X509CA                # Where CA files are kept
certs         = $dir/certs             # Where issued certs are kept
crl_dir       = $dir/crl               # Where the issued crl are kept
database      = $dir/index.txt        # Database index file
new_certs_dir = $dir/newcerts         # Default place for new certs

certificate    = $dir/ca/new_ca.pem   # The CA certificate
serial        = $dir/serial            # The current serial number
crl            = $dir/crl.pem          # The current CRL
private_key    = $dir/ca/new_ca_pk.pem # The private key
RANDFILE      = $dir/ca/.rand         # Private random number file

x509_extensions = usr_cert           # The extensions to add to the cert
...
```



#### NOTE

You might decide to edit other details of the OpenSSL configuration at this point. For more details, see the [OpenSSL documentation](#).

- e. Initialize the CA database as described in [the section called “CA database files”](#).
- f. Create a new self-signed CA certificate and private key with the command:

```
openssl req -x509 -new -config X509CA/openssl.cnf -days 365 -out
X509CA/ca/new_ca.pem -keyout X509CA/ca/new_ca_pk.pem
```

You are prompted for a pass phrase for the CA private key and details of the CA distinguished name as shown in [Example A.2, “Creating a CA Certificate”](#).

**Example A.2. Creating a CA Certificate**

```
Using configuration from X509CA/openssl.cnf Generating a 512
bit RSA private key .....+++++ .+++++ writing new private key to
'new_ca_pk.pem' Enter PEM pass phrase: Verifying password -
Enter PEM pass phrase: ----- You are about to be asked to
enter information that will be incorporated into your
certificate request. What you are about to enter is what is
called a Distinguished Name or a DN. There are quite a few
fields but you can leave some blank. For some fields there will
be a default value, If you enter '.', the field will be left
blank. ----- Country Name (2 letter code) []:IE
State or Province Name (full name) []:Co. Dublin
Locality Name (eg, city) []:Dublin
Organization Name (eg, company) []:Red Hat
Organizational Unit Name (eg, section) []:Content Services
Common Name (eg, YOUR name) []:Dita Pressgang
Email Address []:nobody@redhat.com
```

**NOTE**

The security of the CA depends on the security of the private key file and the private key pass phrase used in this step.

You must ensure that the file names and location of the CA certificate and private key, **new\_ca.pem** and **new\_ca\_pk.pem**, are the same as the values specified in **openssl.cnf** during [Step 2.d](#).

3. Create signed certificates in a Java keystore.
  - a. Generate a certificate and private key pair using the **keytool -genkeypair** command.

For details on the options to use when using **keytool -genkeypair** see [the section called "Generate a certificate and private key pair"](#).

- b. Create a certificate signing request using the **keytool -certreq** command.

[Example A.3, "Creating a CSR"](#) creates a new certificate signing request for the **fusesample.jks** certificate and exports it to the **fusesample\_csr.pem** file.

**Example A.3. Creating a CSR**

```
keytool -certreq -alias fuse -file fusesample_csr.pem -keypass
fusepass -keystore fusesample.jks -storepass fusestorepass
```

- c. Sign the CSR using the **openssl ca** command.

You will be prompted to enter the CA private key pass phrase you used when creating the CA in [Step 2.f](#)).

See [the section called “Signing a CSR”](#) for details on the options to use when signing the CSR.

- d. Convert the signed certificate to PEM only format using the **openssl x509** command with the **-outform** option set to **PEM**.

[Example A.4, “Converting a Signed Certificate to PEM”](#) converts the signed certificate **fusesigned.pem**.

#### Example A.4. Converting a Signed Certificate to PEM

```
openssl x509 -in fusesigned.pem -out fusesigned.pem -outform PEM
```

- e. Concatenate the CA certificate file and the converted, signed certificate file to form a certificate chain.

The CA certificate file is stored in the CA's **ca** directory. For example, the certificate file for the CA created in [Step 2.f](#) would be **ca/new\_ca.pem**.

- f. Import the new certificate's full certificate chain into the Java keystore using the **keytool -import** command.

[Example A.5, “Importing a Certificate Chain”](#) imports the chain **fusesample.chain** into the **fusesample.jks** keystore.

#### Example A.5. Importing a Certificate Chain

```
keytool -import -file fusesample.chain -keystore fusesample.jks -storepass fusestorepass
```

4. Repeat [Step 3](#) to create a full set of certificates for your system.

5. Add trusted CAs to your Java trust store.

- a. Assemble the collection of trusted CA certificates that you want to deploy.

The trusted CA certificates can be obtained from public CAs or private CAs. The trusted CA certificates can be in any format that is compatible with the Java **keystore** utility; for example, PEM format. All you need are the certificates themselves—the private keys and passwords are *not* required.

- b. Add a CA certificate to the trust store using the **keytool -import** command.

[Example A.6, “Adding a CA to the Trust Store”](#) adds the CA certificate **cacert.pem**, in PEM format, to a JKS trust store.

#### Example A.6. Adding a CA to the Trust Store

```
keytool -import -file cacert.pem -alias CAAlias -keystore truststore.ts -storepass StorePass
```

■

**truststore.ts** is a keystore file containing CA certificates. If this file does not already exist, the **keytool** command creates one. **StorePass** is the password required to access the keystore file.

c. Repeat [Step 5.b](#) to add all of the CA certificates to the trust store.

## CA database files

The CA uses two files, **serial** and **index.txt** to maintain its database of certificate files. Both files must be stored in the **X509CA** directory.

When you first create your CA the OpenSSL tools require that they have very specific initial contents:

- **serial**

The initial contents of this file must be **01**.

- **index.txt**

Initially this file *must* be completely empty. It cannot even contain white space.

## Generate a certificate and private key pair

To generate a certificate and private key pair you use the **keytool -genkeypair** command. For example, [Example A.7, “Creating a Certificate and Private Key using Keytool”](#) creates a certificate and key pair that are valid for 365 days and is stored in the keystore file **fusesample.jks**. The generated key store entry will use the alias **fuse** and the password **fusepass**.

### Example A.7. Creating a Certificate and Private Key using Keytool

```
keytool -genkeypair -dname "CN=Alice, OU=Engineering, O=Progress, ST=Co.
Dublin, C=IE" -validity 365 -alias fuse -keypass fusepass -keystore
fusesample.jks -storepass fusestorepass
```

Because the specified keystore, **fusesample.jks**, did not exist prior to issuing the command implicitly creates a new keystore and sets its password to **fusestorepass**.

The **-dname** and **-validity** flags define the contents of the newly created X.509 certificate.

The **-dname** flag specifies the subject DN. For more details about DN format, see [Appendix B, ASN.1 and Distinguished Names](#). Some parts of the subject DN must match the values in the CA certificate (specified in the CA Policy section of the **openssl.cnf** file). The default **openssl.cnf** file requires the following entries to match:

- Country Name (C)
- State or Province Name (ST)
- Organization Name (O)

**NOTE**

If you do not observe the constraints, the OpenSSL CA will refuse to sign the certificate (see [Step 2.f](#)).

The **-validity** flag specifies the number of days for which the certificate is valid.

## Signing a CSR

To sign a CSR using your CA, you use the **openssl ca** command. At a minimum you will need to specify the following options:

- **-config**—the path to the CA's **openssl.cnf** file
- **-in**—the path to certificate to be signed
- **-out**—the path to the signed certificates

[Example A.8, “Signing a CSR”](#) signs the **fusesample\_csr.pem** certificate using the CA stored at **/etc/fuseCA**.

### Example A.8. Signing a CSR

```
openssl ca -config /etc/fuse/openssl.cnf -days 365 -in
fusesample_csr.pem -out fusesigned.pem
```

For more details on the **openssl ca** command see <http://www.openssl.org/docs/apps/ca.html#>.



## APPENDIX B. ASN.1 AND DISTINGUISHED NAMES

### Abstract

The OSI Abstract Syntax Notation One (ASN.1) and X.500 Distinguished Names play an important role in the security standards that define X.509 certificates and LDAP directories.

### B.1. ASN.1

#### Overview

The *Abstract Syntax Notation One* (ASN.1) was defined by the OSI standards body in the early 1980s to provide a way of defining data types and structures that are independent of any particular machine hardware or programming language. In many ways, ASN.1 can be considered a forerunner of modern interface definition languages, such as the OMG's IDL and WSDL, which are concerned with defining platform-independent data types.

ASN.1 is important, because it is widely used in the definition of standards (for example, SNMP, X.509, and LDAP). In particular, ASN.1 is ubiquitous in the field of security standards—the formal definitions of X.509 certificates and distinguished names are described using ASN.1 syntax. You do not require detailed knowledge of ASN.1 syntax to use these security standards, but you need to be aware that ASN.1 is used for the basic definitions of most security-related data types.

#### BER

The OSI's Basic Encoding Rules (BER) define how to translate an ASN.1 data type into a sequence of octets (binary representation). The role played by BER with respect to ASN.1 is, therefore, similar to the role played by GIOP with respect to the OMG IDL.

#### DER

The OSI's Distinguished Encoding Rules (DER) are a specialization of the BER. The DER consists of the BER plus some additional rules to ensure that the encoding is unique (BER encodings are not).

#### References

You can read more about ASN.1 in the following standards documents:

- ASN.1 is defined in X.208.
- BER is defined in X.209.

### B.2. DISTINGUISHED NAMES

#### Overview

Historically, distinguished names (DN) are defined as the primary keys in an X.500 directory structure. However, DNs have come to be used in many other contexts as general purpose identifiers. In Apache CXF, DNs occur in the following contexts:

- X.509 certificates—for example, one of the DNs in a certificate identifies the owner of the certificate (the security principal).
- LDAP—DNs are used to locate objects in an LDAP directory tree.

## String representation of DN

Although a DN is formally defined in ASN.1, there is also an LDAP standard that defines a UTF-8 string representation of a DN (see **RFC 2253**). The string representation provides a convenient basis for describing the structure of a DN.



### NOTE

The string representation of a DN does *not* provide a unique representation of DER-encoded DN. Hence, a DN that is converted from string format back to DER format does not always recover the original DER encoding.

## DN string example

The following string is a typical example of a DN:

```
C=US,O=IONA Technologies,OU=Engineering,CN=A. N. Other
```

## Structure of a DN string

A DN string is built up from the following basic elements:

- [OID](#) .
- [Attribute Types](#) .
- [AVA](#) .
- [RDN](#) .

## OID

An OBJECT IDENTIFIER (OID) is a sequence of bytes that uniquely identifies a grammatical construct in ASN.1.

## Attribute types

The variety of attribute types that can appear in a DN is theoretically open-ended, but in practice only a small subset of attribute types are used. [Table B.1, “Commonly Used Attribute Types”](#) shows a selection of the attribute types that you are most likely to encounter:

**Table B.1. Commonly Used Attribute Types**

String Representation	X.500 Attribute Type	Size of Data	Equivalent OID
C	countryName	2	2.5.4.6
O	organizationName	1...64	2.5.4.10

String Representation	X.500 Attribute Type	Size of Data	Equivalent OID
OU	organizationalUnitName	1..64	2.5.4.11
CN	commonName	1..64	2.5.4.3
ST	stateOrProvinceName	1..64	2.5.4.8
L	localityName	1..64	2.5.4.7
STREET	streetAddress		
DC	domainComponent		
UID	userid		

## AVA

An *attribute value assertion* (AVA) assigns an attribute value to an attribute type. In the string representation, it has the following syntax:

```
<attr-type>=<attr-value>
```

For example:

```
CN=A. N. Other
```

Alternatively, you can use the equivalent OID to identify the attribute type in the string representation (see [Table B.1, “Commonly Used Attribute Types”](#)). For example:

```
2.5.4.3=A. N. Other
```

## RDN

A *relative distinguished name* (RDN) represents a single node of a DN (the bit that appears between the commas in the string representation). Technically, an RDN might contain more than one AVA (it is formally defined as a set of AVAs). However, this almost never occurs in practice. In the string representation, an RDN has the following syntax:

```
<attr-type>=<attr-value>[+<attr-type>=<attr-value> ...]
```

Here is an example of a (very unlikely) multiple-value RDN:

```
OU=Eng1+OU=Eng2+OU=Eng3
```

Here is an example of a single-value RDN:

OU=Engineering

## INDEX

### A

**Abstract Syntax Notation One** (see [ASN.1](#))

**administration**

OpenSSL command-line utilities, [OpenSSL utilities](#)

**Aries**

namespaces, [Namespaces](#)

placeholder extension, [Placeholder extension](#)

**ASN.1**, [Contents of an X.509 certificate](#), [ASN.1 and Distinguished Names](#)

attribute types, [Attribute types](#)

AVA, [AVA](#)

OID, [OID](#)

RDN, [RDN](#)

**attribute value assertion** (see [AVA](#))

**authentication**, [LDAP properties](#)

**AVA**, [AVA](#)

### B

**Basic Encoding Rules** (see [BER](#))

**BER**, [BER](#)

### C

**CA**, [Integrity of the public key](#)

choosing a host, [Choosing a host for a private certification authority](#)

commercial CAs, [Commercial Certification Authorities](#)

index file, [CA database files](#)

list of trusted, [Trusted CAs](#)

multiple CAs, [Certificates signed by multiple CAs](#)

private CAs, [Private Certification Authorities](#)

private key, creating, [Procedure](#)

security precautions, [Security precautions](#)

self-signed, [Procedure](#)

---

serial file, [CA database files](#)

setting up, [Procedure](#)

certificate signing request, [Procedure](#)

signing, [Procedure](#)

certificates

chaining, [Certificate chain](#)

peer, [Chain of trust](#)

public key, [Contents of an X.509 certificate](#)

self-signed, [Self-signed certificate](#), [Procedure](#)

signing, [Integrity of the public key](#), [Procedure](#)

signing request, [Procedure](#)

X.509, [Role of certificates](#)

chaining of certificates, [Certificate chain](#)

connection.password, [LDAP properties](#)

connection.url, [LDAP properties](#)

connection.username, [LDAP properties](#)

CSR, [Procedure](#)

**D**

DER, [DER](#)

Distinguished Encoding Rules (see DER)

distinguished names

definition, [Overview](#)

**DN**

definition, [Overview](#)

string representation, [String representation of DN](#)

**E**

encryptor, [Jasypt configuration](#)

**F**

features:install, [Installing the Jasypt features](#)

**I**

index file, [CA database files](#)

initial.context.factory, [LDAP properties](#)

## J

### JAAS

configuration syntax, [Configuring a JAAS realm](#)

converting to blueprint, [Converting standard JAAS login properties to XML](#)

namespace, [Namespace](#)

jaas:config, [Configuring a JAAS realm](#)

jaas:module, [Configuring a JAAS realm](#)

### Jasypt

configuration, [Jasypt configuration](#)

libraries, [Installing the Jasypt features](#)

namespaces, [Namespaces](#)

jasypt-encryption, [Installing the Jasypt features](#)

### JMX

roles, [Changing the JMX role](#)

JMX SSL connection, enabling, [Enabling Remote JMX SSL](#)

## L

### LDAP

authentication, [LDAP properties](#)

configuration, [LDAP properties](#)

connection.password, [LDAP properties](#)

connection.url, [LDAP properties](#)

connection.username, [LDAP properties](#)

enabling, [Enabling LDAP Authentication](#)

initial.context.factory, [LDAP properties](#)

properties, [LDAP properties](#)

role.base.dn, [LDAP properties](#)

role.filter, [LDAP properties](#)

role.name.attribute, [LDAP properties](#)

role.search.subtree, [LDAP properties](#)

ssl, [LDAP properties](#)

---

- ssl.algorithm, [LDAP properties](#)
- ssl.keyalias, [LDAP properties](#)
- ssl.keystore, [LDAP properties](#)
- ssl.protocol, [LDAP properties](#)
- ssl.provider, [LDAP properties](#)
- ssl.truststore, [LDAP properties](#)
- user.base.dn, [LDAP properties](#)
- user.filter, [LDAP properties](#)
- user.search.subtree, [LDAP properties](#)

LDAPLoginModule, [Enabling LDAP Authentication](#)

## M

multiple CAs, [Certificates signed by multiple CAs](#)

## N

namespaces

- Aries, [Namespaces](#)

- Jasypt, [Namespaces](#)

## O

OpenSSL, [OpenSSL software package](#)

OpenSSL command-line utilities, [OpenSSL utilities](#)

## P

peer certificate, [Chain of trust](#)

private key, [Procedure](#)

properties

- Apache Karaf placeholder extension, [Jasypt configuration](#)

- Aries placeholder extension, [Placeholder extension](#)

- encrypted, [Encrypted properties](#)

- LDAP, [LDAP properties](#)

- placeholder, [Placeholders](#)

property-placeholder, [Placeholder extension](#), [Jasypt configuration](#)

public keys, [Contents of an X.509 certificate](#)

## R

**RDN**, [RDN](#)

relative distinguished name (see [RDN](#))

remote console

roles, [Changing the remote console's role](#)

role.base.dn, [LDAP properties](#)

role.filter, [LDAP properties](#)

role.name.attribute, [LDAP properties](#)

role.search.subtree, [LDAP properties](#)

roles

default, [Default role](#)

JMX, [Changing the JMX role](#)

LDAP configuration, [LDAP properties](#)

remote console, [Changing the remote console's role](#)

root certificate directory, [Trusted CAs](#)

## S

self-signed CA, [Procedure](#)

self-signed certificate, [Self-signed certificate](#)

serial file, [CA database files](#)

signing certificates, [Integrity of the public key](#)

ssl, [LDAP properties](#)

ssl.algorithm, [LDAP properties](#)

ssl.keyalias, [LDAP properties](#)

ssl.keystore, [LDAP properties](#)

ssl.protocol, [LDAP properties](#)

ssl.provider, [LDAP properties](#)

ssl.truststore, [LDAP properties](#)

SSLey, [OpenSSL software package](#)

## T

trusted CAs, [Trusted CAs](#)

## U



**user.base.dn, [LDAP properties](#)**

**user.filter, [LDAP properties](#)**

**user.search.subtree, [LDAP properties](#)**

## **X**

**X.500, [ASN.1 and Distinguished Names](#)**

**X.509 certificate**

**definition, [Role of certificates](#)**