Progress FUSE™

FUSE™ Message Broker

Security Guide

Version 5.3
Febuary 2009

PROGRESS SOFTWARE

Security Guide

Version 5.3

Publication date 23 Jul 2009

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. SSL/TLS Security

You can use SSL/TLS security to secure connections to brokers for a variety of different protocols: Openwire over TCP/IP, Openwire over HTTP, and Stomp.

Introduction to SSL/TLS

Overview

The Secure Sockets Layer (SSL) protocol was originally developed by Netscape Corporation to provide a mechanism for secure communication over the Internet. Subsequently, the protocol was adopted by the Internet Engineering Task Force (IETF) and renamed to Transport Layer Security (TLS). The latest specification of the TLS protocol is RFC 5246[1].

The SSL/TLS protocol sits between an application protocol layer and a reliable transport layer (such as TCP/IP). It is independent of the application protocol and can thus be layered underneath many different protocols, for example: HTTP, FTP, SMTP, and so on.

SSL/TLS security features

The SSL/TLS protocol supports the following security featues:

- *Privacy*—messages are encrypted using a secret symmetric key, making it impossible for eavesdroppers to read messages sent over the connection.
- *Message integrity*—messages are digitally signed, to ensure that they cannot be tampered with.
- *Authentication*—the identity of the target (server program) is authenticated and (optionally) the client as well.
- *Immunity to man-in-the-middle attacks*—because of the way authentication is performed in SSL/TLS, it is impossible for an attacker to interpose itself between a client and a target.

Cipher suites

To support all of the facets of SSL/TLS security, a number of different security algorithms must be used together. Moreover, for each of the security features (for example, message integrity), there are typically several different algorithms available. To manage these alternatives, the security algorithms are grouped together into *cipher suites*. Each cipher suite contains a complete collection of security algorithms for the SSL/TLS protocol. />.

Public key cryptography

Public key cryptography (also known as *asymmetric cryptography*) plays a critically important role in SSL/TLS security. With this form of cryptography, encryption and decryption is performed using a matching pair of keys: a *public*

[1] http://tools.ietf.org/html/rfc5246

key and a *private key*. A message encrypted by the public key can *only* be decrypted by the private key; and a message encrypted by the private key can *only* be decrypted by the public key. This basic mathematical property has some important consequences for cryptography:

- It becomes extremely easy to establish secure communications with people you have never previously had any contact with. Simply publish the public key in some accessible place. Anyone can now download the public key and use it to encrypt a message that *only you* can decrypt, using your private key.
- You can use your private key to digitally sign messages. Given a message to sign, simply generate a hash value from the message, encrypt that hash value using your private key, and append it to the message. Now, anyone can use the public key to decrypt the hash value and check that the message has not been tampered with.

Note

Actually, it is not compulsory to use public key cryptography with SSL/TLS. But the SSL/TLS protocol is practically useless (and very insecure) without it.

X.509 certificates

An X.509 certificate provides a way of binding an identity (in the form of an X.500 *distinguished name*) to a public key. X.509 is a standard specified by the IETF and the most recent specification is RFC 4158[2]. The X.509 certificate consists essentially of an identity concatenated with a public key, with the whole certificate being digitally signed in order to guarantee the association between the identity and the public key.

But who signs the certificate? It has to be someone (or some identity) that you trust. The certificate signer could be one of the following:

- *Self*—if the certificate signs itself, it is called a *self-signed certificate*. If you need to deploy a self-signed certificate, the certificate must be obtained from a secure channel. The only guarantee you have of the certificate's authenticity is that you obtained it from a trusted source.
- *CA certificate*—a more scalable solution is to sign certificates using a Certificate Authority (CA) certificate. In this case, you only need to be careful about deploying the original CA certificate (that is, obtaining it through a

[2] http://tools.ietf.org/html/rfc4158

secure channel). All of the certificates signed by this CA, on the other hand, can be distributed over insecure, public channels. The trusted CA can then be used to verify the signature on the certificates. In this case, the CA certificate is self-signed.

- *Chain of CA certificates*—an extension of the idea of signing with a CA certificate is to use a chain of CA certificates. For example, certificate X could be signed by CA foo, which is signed by CA bar. The last CA certificate in the chain (the *root certificate*) is self-signed.

For more details about managing X.509 certificates, see "Managing Certificates" on page 33.

Target-only authentication

The most common way to configure SSL/TLS is to associate an X.509 certificate with the target (server side) but not with the client. This implies that the client can verify the identity of the target, but the target cannot verify the identity of the client (at least, not through the SSL/TLS protocol). It might seem strange that we worry about protecting clients (by confirming the target identity) but not about protecting the target. Keep in mind, though, that SSL/TLS security was originally developed for the Internet, where protecting clients is a high priority. For example, if you are about to connect to your bank's Web site, you want to be very sure that the Web site is authentic. Also, it is typically easier to authenticate clients using other mechanisms (such as HTTP Basic Authentication), which do not incur the high maintenance overhead of generating and distributing X.509 certificates.

Secure Transport Protocols

Overview

FUSE Message Broker provides a common framework for adding SSL/TLS security to its transport protocols. All of the transport protocols discussed here are secured using the JSSE framework and most of their configuration settings are shared.

Transport protocols

Table 1.1 on page 15 shows the transport protocols that can be secured using SSL/TLS.

Table 1.1. Secure Transport Protocols

URL	Description
`ssl://Host:Port`	Endpoint URL for Openwire over TCP/IP, where the socket layer is secured using SSL or TLS.
`https://Host:Port`	Endpoint URL for Openwire over HTTP, where the socket layer is secured using SSL or TLS.
`stomp+ssl://Host:Port`	Endpoint URL for Stomp over TCP/IP, where the socket layer is secured using SSL or TLS.

Java Keystores

Overview

Java keystores provide a convenient mechanism for storing and deploying X.509 certificates and private keys. FUSE Message Broker uses Java keystore files as the standard format for deploying certificates

Prerequisites

The Java keystore is a feature of the *Java platform Standard Edition (SE)* from Sun. To perform the tasks described in this section, you will need to install a recent version of the Java Development Kit (JDK) and ensure that the JDK `bin` directory is on your path. See http://java.sun.com/javase/.

Default keystore provider

Sun's JDK provides a standard file-based implementation of the keystore. The instructions in this section presume you are using the standard keystore. If there is any doubt about the kind of keystore you are configured to use, check the following line in your `java.security` file (located either in *`JavaInstallDir`*`/lib/security` or *`JavaInstallDir`*`/jre/lib/security`):

```
keystore.type=jks
```

The `jks` (or `JKS`) keystore type represents the standard keystore.

Customizing the keystore provider

Java also allows you to provide a custom implementation of the keystore, by implementing the `java.security.KeystoreSpi` class. For details of how to do this see the following references:

- http://java.sun.com/j2se/1.5.0/docs/tooldocs/windows/keytool.html
- http://java.sun.com/j2se/1.5.0/docs/guide/security/HowToImplAProvider.html

If you use a custom keystore provider, you should consult the third-party provider documentation for details of how to manage certificates and private keys with this provider.

Store password

The keystore repository is protected by a *store password*, which is defined at the same time the keystore is created. Every time you attempt to access or modify the keystore, you must provide the store password.

Note

The store password can also be referred to as a *keystore password* or a *truststore password*, depending on what kind of entries are stored in the keystore file. The function of the password in both cases is the same: that is, to unlock the keystore file.

Keystore entries

The keystore provides two distinct kinds of entry for storing certificates and private keys, as follows:

- *Key entries*—each key entry contains the following components:
    - A private key.
    - An X.509 certificate (can be v1, v2, or v3) containing the public key that matches this entry's private key.
    - Optionally, one or more CA certificates that belong to the preceding certificate's trust chain.

  Note

  The CA certificates belonging to a certificate's trust chain can be stored either in its key entry or in trusted certificate entries.

  In addition, each key entry is tagged by an alias and protected by a key password. To access a particular key entry in the keystore, you must provide both the alias and the key password.

- *Trusted certificate entries*—each trusted certificate entry contains just a single X.509 certificate.

  Each trusted certificate entry is tagged by an alias. There is no need to protect the entry with a password, however, because the X.509 certificate contains only a public key.

Keystore utilities

The Java platform SE provides two keystore utilities: `keytool` and `jarsigner`. Only the `keytool` utility is needed here.

How to Use X.509 Certificates

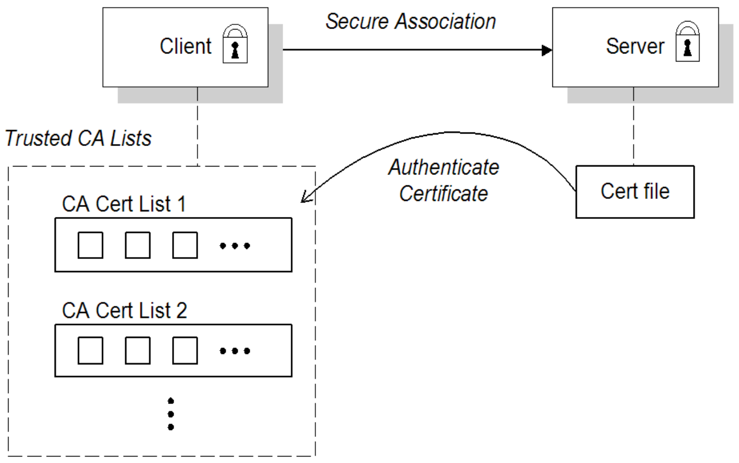
Overview

Before you can understand how to deploy X.509 certificates in a real system, you need to know about the different authentication scenarios supported by the SSL/TLS protocol. The way you deploy the certificates depends on what kind of authentication scenario you decide to adopt for your application.

Target-only authentication

In the target-only authentication scenario, as shown in Figure 1.1 on page 18, the target (in this case, the broker) presents its own certificate to the client during the SSL/TLS handshake, so that the client can verify the target's identity. In this scenario, therefore, the target is authentic to the client, but the client is not authentic to the target.

Figure 1.1. Target-Only Authentication Scenario

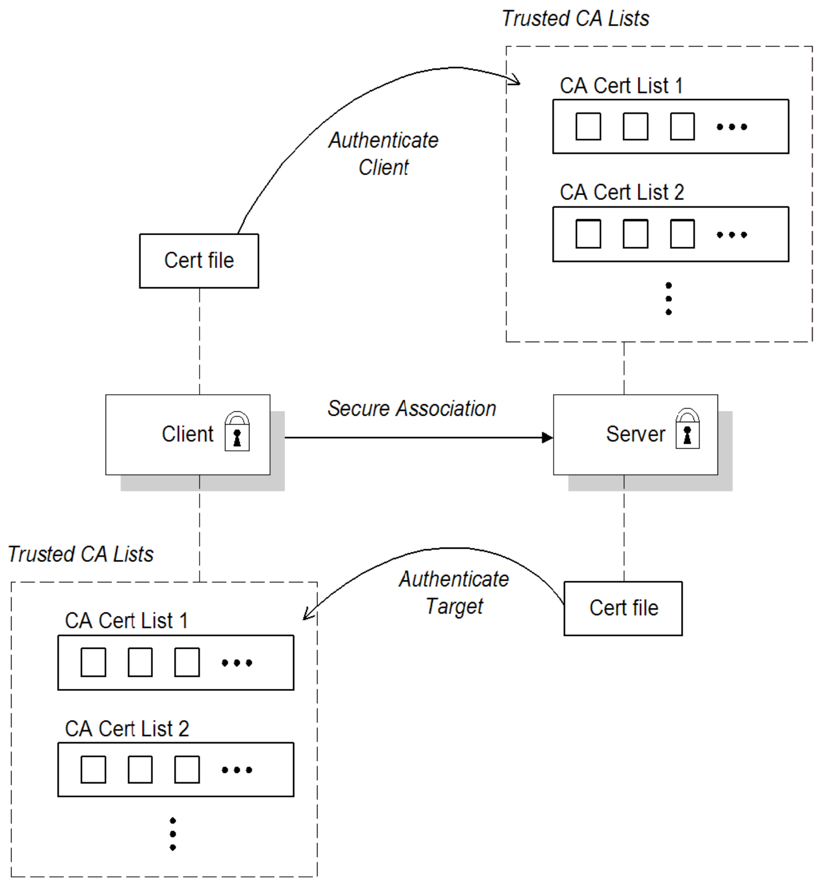


The broker is configured to have its own certificate and private key, which are both stored in the file, `broker.ks`. The client is configured to have a trust store, `client.ts`, that contains the certificate that originally signed the broker certificate. Normally, the trusted certificate is a Certificate Authority (CA) certificate.

Mutual authentication

In the mutual authentication scenario, as shown in Figure 1.2 on page 19, the target presents its own certificate to the client and the client presents its own certificate to the target during the SSL/TLS handshake, so that both the client and the target can verify each other's identity. In this scenario, therefore, the target is authentic to the client and the client is authentic to the target.

Figure 1.2. Mutual Authentication Scenario



Because authentication is mutual in this scenario, both the client and the target must be equipped with a full set of certificates. The client is configured to have its own certificate and private key in the file, `client.ks`, and a trust store, `client.ts`, which contains the certificate that signed the target certificate. The target is configured to have its own certificate and private key in the file, `broker.ks`, and a trust store, `broker.ts`, which contains the certificate that signed the client certificate.

Selecting the authentication scenario

Various combinations of target and client authentication are theoretically supported by the SSL/TLS protocols. In general, SSL/TLS authentication scenarios are controlled by selecting a specific cipher suite (or cipher suites) and by setting flags in the SSL/TLS protocol layer (that is, the *WantClientAuth* or *NeedClientAuth* flags). The following list describes all of the possible authentication scenarios (some of which are *not* supported by FUSE Message Broker):

- *Target-only authentication—(supported)* this is the most important authentication scenario. If you want to authenticate the client as well, the most common approach is to let the client log on using username/password credentials, which can be sent securely through the encrypted channel established by the SSL/TLS session.
- *Target authentication and optional client authentication—(supported)* if you want to authenticate the client using an X.509 certificate, simply configure the client to have its own certificate. By default, the target will authenticate the client's certificate, if it receives one.
- *Target authentication and required client authentication—(not supported)* it is theoretically possible to configure a target to *require* client authentication by setting the *NeedClientAuth* flag on the SSL/TLS protocol layer. When this flag is set, the target would raise an error, if the client fails to send a certificate during the SSL/TLS handshake. Currently, this option is *not* supported by FUSE Message Broker. The *NeedClientAuth* flag is always set to false.
- *No authentication*—this scenario is potentially dangerous from a security perspective, because it is susceptible to a man-in-the-middle attack. *It is therefore recommended that you always avoid using this (non-)authentication scenario.*

  It is theoretically possible to get this scenario, if you select one of the anonymous Diffie-Hellman cipher suites for the SSL/TLS session. In practice, however, you normally do not need to worry about these cipher suites, because they have a low priority amongst the cipher suites supported by the `SunJSSE` security provider. Other, more secure cipher suites normally take precedence.

Demonstration certificates

FUSE Message Broker provides a collection of demonstration certificates, located in the `$ACTIVEMQ_HOME/conf` directory, that enable you to get started quickly and run some examples using the secure transport protocols. The following keystore files are provided (where, by convention, the `.ks` suffix denotes a keystore file with key entries and the `.ts` suffix denotes a keystore file with trusted certificate entries):

- `broker.ks`—broker keystore, contains the broker's self-signed X.509 certificate and its associated private key.
- `broker.ts`—broker trust store, contains the *client*'s self-signed X.509 certificate.
- `client.ks`—client keystore, contains the client's self-signed X.509 certificate and its associated private key.
- `client.ts`—client trust store, contains the *broker*'s self-signed X.509 certificate.

Warning

Do not deploy the demonstration certificates in a live production system! These certificate are provided for demonstration and testing purposes only. For a real system, create your own custom certificates.

Custom certificates

For a real deployment of a secure SSL/TLS application, you must first create a collection of custom X.509 certificates and private keys. For detailed instructions on how to go about creating and managing your X.509 certificates, see . "Managing Certificates" on page 33

Configuring JSSE System Properties

Overview

Java Secure Socket Extension (JSSE) provides the underlying framework for the SSL/TLS implementation in FUSE Message Broker. In this framework, you configure the SSL/TLS protocol and deploy X.509 certificates using a variety of JSSE system properties.

JSSE system properties

Table 1.2 on page 22 shows the JSSE system properties that can be used to configure SSL/TLS security for the SSL (Openwire over SSL), HTTPS (Openwire over HTTPS), and Stomp+SSL (Stomp over SSL) transport protocols.

Table 1.2. JSSE System Properties

System Property Name	Description
`javax.net.ssl.keyStore`	Location of the Java keystore file containing an application process's own certificate and private key. On Windows, the specified pathname must use forward slashes, `/`, in place of backslashes, `\`.
`javax.net.ssl.keyStorePassword`	Password to access the private key from the keystore file specified by `javax.net.ssl.keyStore`. This password is used twice: • To unlock the keystore file (store password), and • To decrypt the private key stored in the keystore (key password). In other words, the JSSE framework requires these passwords to be identical.
`javax.net.ssl.keyStoreType`	*(Optional)* For Java keystore file format, this property has the value `jks` (or `JKS`). You do not normally specify this property, because its default value is already `jks`.
`javax.net.ssl.trustStore`	Location of the Java keystore file containing the collection of CA certificates trusted by this application process (trust store). On Windows, the specified pathname must use forward slashes, `/`, in place of backslashes, `\`. If a trust store location is not specified using this property, the SunJSSE implementation searches for and uses a keystore file in the following locations (in order):

System Property Name	Description
	1. `$JAVA_HOME/lib/security/jssecacerts` 2. `$JAVA_HOME/lib/security/cacerts`
`javax.net.ssl.trustStorePassword`	Password to unlock the keystore file (store password) specified by `javax.net.ssl.trustStore`.
`javax.net.ssl.trustStoreType`	*(Optional)* For Java keystore file format, this property has the value `jks` (or `JKS`). You do not normally specify this property, because its default value is already `jks`.
`javax.net.debug`	To switch on logging for the SSL/TLS layer, set this property to `ssl`.

Warning

The default trust store locations (in the `jssecacerts` and the `cacerts` directories) present a potential security hazard. If you do not take care to manage the trust stores under the JDK installation or if you do not have control over which JDK installation is used, you might find that the effective trust store is too lax.

To be on the safe side, it is recommended that you *always* set the `javax.net.ssl.trustStore` property for a secure client or server, so that you have control over the CA certificates trusted by your application.

Setting properties at the command line

On the client side and in the broker, you can set the JSSE system properties on the Java command line using the standard syntax, `-D`*`Property`*`=`*`Value`*. For example, to specify JSSE system properties to a client program, `com.progress.Client`:

```
java -Djavax.net.ssl.trustStore=truststores/client.ts
com.progress.Client
```

To configure a broker to use the demonstration broker keystore and demonstration broker trust store, you can set the `SSL_OPTS` environment variable as follows, on Windows:

```
set SSL_OPTS=-Djavax.net.ssl.keyStore=C:/Programs/FUSE/fuse-
message-broker-5.3.0.0/conf/broker.ks
```

```
         -Djavax.net.ssl.keyStorePassword=password
         -Djavax.net.ssl.trustStore=C:/Programs/FUSE/fuse-
message-broker-5.3.0.0/conf/broker.ts
         -Djavax.net.ssl.trustStorePassword=password
```

Or on UNIX platforms (Bourne shell):

```
SSL_OPTS=-Djavax.net.ssl.keyStore=/local/FUSE/fuse-message-
broker-5.3.0.0/conf/broker.ks
         -Djavax.net.ssl.keyStorePassword=password
        -Djavax.net.ssl.trustStore=/local/FUSE/fuse-message-
broker-5.3.0.0/conf/broker.ts
         -Djavax.net.ssl.trustStorePassword=password
export SSL_OPTS
```

You can then launch the broker using the `bin/activemq[.bat|.sh]` script

Note

The `SSL_OPTS` environment variable is simply a convenient way of passing command-line properties to the `bin/activemq[.bat|.sh]` script. It is *not* accessed directly by the broker runtime or the JSSE package.

Setting properties by programming

You can also set JSSE system properties using the standard Java API, as long as you set the properties before the relevant transport protocol is initialized. For example:

```
// Java
import java.util.Properties;
...
Properties systemProps = System.getProperties();
systemProps.put(
    "javax.net.ssl.trustStore",
    "C:/Programs/FUSE/fuse-message-broker-5.3.0.0/conf/cli
ent.ts"
);
System.setProperties(systemProps);
```

Setting Security Context for the Openwire/SSL Protocol

Overview

Apart from configuration using JSSE system properties, the Openwire/SSL protocol (with schema, `ssl:`) also supports an option to set its SSL security context using the broker configuration file.

Note

The methods for setting the security context described in this section are available *exclusively* for the Openwire/SSL protocol. These features are *not* supported by the HTTPS protocol.

Setting security context in the broker configuration file

To configure the Openwire/SSL security context in the broker configuration file, edit the attributes in the `sslContext` element. For example, the default broker configuration file, `conf/activemq.xml`, includes the following entry:

```
<beans ...>
    ...
    <broker ...>
        <sslContext>
            <sslContext keyStore="file:${act
ivemq.base}/conf/broker.ks"
                        keyStorePassword="password"
                        trustStore="file:${act
ivemq.base}/conf/broker.ts"
                        trustStorePassword="password"/>
        </sslContext>
        ...
    </broker>
    ...
</beans>
```

Where the `activemq.base` property is defined in the `activemq[.bat|.sh]` script. You can specify any of the following `sslContext` attributes:

- `keyStore`—equivalent to setting `javax.net.ssl.keyStore`.
- `keyStorePassword`—equivalent to setting `javax.net.ssl.keyStorePassword`.
- `keyStoreType`—equivalent to setting `javax.net.ssl.keyStoreType`.

- `keyStoreAlgorithm`—
- `trustStore`—equivalent to setting `javax.net.ssl.trustStore`.
- `trustStorePassword`—equivalent to setting `javax.net.ssl.trustStorePassword`.
- `trustStoreType`—equivalent to setting `javax.net.ssl.trustStoreType`.

SSL/TLS Tutorial

Overview

This tutorial demonstrates how to connect to a broker through the SSL protocol (Openwire over SSL) and through the HTTPS protocol (Openwire over HTTPS). For simplicity, the tutorial uses the demonstration certificates (key stores and trust stores) provided with initial installation of FUSE Message Broker. *These demonstration certificates must not be used in a live production system, however.*

Prerequisites

Before you can build and run the sample clients, you must have installed the Apache Ant build tool, version 1.6 or later (see http://ant.apache.org/).

The OpenWire examples depend on the sample producer and consumer clients located in the following directory:

```
FUSEInstallDir/fuse-message-broker-Version/example
```

Sample consumer and producer clients

For the purposes of testing and experimentation, FUSE Message Broker provides a sample consumer client and a sample producer client in the `example` subdirectory. You can build and run these clients using the `consumer` and the `producer` Ant targets. In the following tutorial, these sample clients are used to demonstrate how to connect to secure endpoints in the broker.

Tutorial steps

To try out the secure SSL and HTTPS protocols, perform the following steps:

1. "Set the broker environment" on page 28.
2. "Configure the broker" on page 28.
3. "Configure the consumer and the producer clients" on page 29.
4. "Run the broker" on page 30
5. "Run the consumer with the SSL protocol" on page 30.
6. "Run the producer with the HTTPS protocol" on page 31.

7. "Enable SSL logging in the consumer" on page 31.

Set the broker environment

Create a script that sets the broker's JSSE system properties using the SSL_OPTS environment variable. On Windows, create a `setSslOpts.bat` script with the following contents:

```
set SSL_OPTS=-Djavax.net.ssl.keyStore=MessageBroker
Root/conf/broker.ks
         -Djavax.net.ssl.keyStorePassword=password
         -Djavax.net.ssl.trustStore=MessageBroker
Root/conf/broker.ts
         -Djavax.net.ssl.trustStorePassword=password
```

On UNIX, create a `setSslOpts.sh` script with the following contents:

```
SSL_OPTS=-Djavax.net.ssl.keyStore=MessageBroker
Root/conf/broker.ks
         -Djavax.net.ssl.keyStorePassword=password
         -Djavax.net.ssl.trustStore=MessageBroker
Root/conf/broker.ts
         -Djavax.net.ssl.trustStorePassword=password
export SSL_OPTS
```

Warning

The demonstration broker key store and broker trust sture 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 "Managing Certificates" on page 33.

Configure the broker

Add the `ssl` and `https` transport connectors to the default broker configuration file (`conf/activemq.xml`), as follows:

```
<beans ...>
    ...
    <broker ...>
        <sslContext>
            <sslContext keyStore="file:${act
ivemq.base}/conf/broker.ks"
                        keyStorePassword="password"
                        trustStore="file:${act
ivemq.base}/conf/broker.ts"
                        trustStorePassword="password"/>
        </sslContext>

        <transportConnectors>
             <transportConnector name="ssl" uri="ssl://local
host:61617"/>
           <transportConnector name="https" uri="https://loc
alhost:8443"/>
            ...
        </transportConnectors>
        ...
    </broker>
    ...
</beans>
```

Configure the consumer and the producer clients

Configure the consumer and the producer clients to pick up the client trust store. Edit the Ant build file, `example/build.xml`, and add the `javax.net.ssl.trustStore` and `javax.net.ssl.trustStorePassword` JSSE system properties to the consumer target and the producer target as shown in the following example:

```
<project ...>
    ...
     <target name="consumer" depends="compile" descrip
tion="Runs a simple consumer">
        ...
          <java classname="ConsumerTool" fork="yes"
maxmemory="100M">
               <classpath refid="javac.classpath" />
               <jvmarg value="-server" />
              <sysproperty key="activemq.home" value="${act
ivemq.home}"/>
              <sysproperty key="javax.net.ssl.trustStore"
                                 value="${act
ivemq.home}/conf/client.ts"/>
              <sysproperty key="javax.net.ssl.trustStorePass
word"
                                 value="password"/>
               <arg value="--url=${url}" />
            ...
          </java>
    </target>

     <target name="producer" depends="compile" descrip
tion="Runs a simple producer">
        ...
          <java classname="ProducerTool" fork="yes"
maxmemory="100M">
```

```
               <classpath refid="javac.classpath" />
               <jvmarg value="-server" />
              <sysproperty key="activemq.home" value="${act
ivemq.home}"/>
              <sysproperty key="javax.net.ssl.trustStore"
                                  value="${act
ivemq.home}/conf/client.ts"/>
              <sysproperty key="javax.net.ssl.trustStorePass
word"
                                  value="password"/>
               <arg value="--url=${url}" />
             ...
         </java>
    </target>
    ...
</project>
```

In the context of the Ant build tool, this is equivalent to adding the system properties to the command line.

Run the broker

Open a new command prompt and run the `setSslOpts.[bat|sh]` script to initialize the `SSL_OPTS` variable in the broker's environment. Now run the default broker by entering the following at a command line:

```
activemq
```

The default broker automatically takes its configuration from the default configuration file.

Note

The `activemq` script automatically sets the `ACTIVEMQ_HOME` and `ACTIVEMQ_BASE` environment variables to *`FUSEInstallDir`*`/fuse-message-broker-`*`Version`* by default. If you want the `activemq` script to pick up its configuration from a non-default `conf` directory, you can set `ACTIVEMQ_BASE` explicitly in your environment. The configuration files will then be taken from `$ACTIVEMQ_BASE/conf`.

Run the consumer with the SSL protocol

To connect the consumer tool to the `ssl://localhost:61617` endpoint (Openwire over SSL), change directory to `example` and enter the following command:

```
ant consumer -Durl=ssl://localhost:61617 -Dmax=100
```

You should see some output like the following:

```
Buildfile: build.xml
init:
compile:
consumer:
     [echo] Running consumer against server at $url =
ssl://localhost:61617 for subject $subject = TEST.FOO
     [java] Connecting to URL: ssl://localhost:61617
     [java] Consuming queue: TEST.FOO
     [java] Using a non-durable subscription
     [java] We are about to wait until we consume: 100 mes
sage(s) then we will shutdown
```

Run the producer with the HTTPS protocol

To connect the producer tool to the `https://localhost:8443` endpoint (Openwire over HTTPS), open a new command prompt, change directory to `example` and enter the following command:

```
ant producer -Durl=https://localhost:8443
```

In the window where the *consumer* tool is running, you should see some output like the following:

```
     [java] Received: Message: 0 sent at: Thu Feb 05 09:27:43
 GMT 2009  ...
     [java] Received: Message: 1 sent at: Thu Feb 05 09:27:43
 GMT 2009  ...
     [java] Received: Message: 2 sent at: Thu Feb 05 09:27:43
 GMT 2009  ...
     [java] Received: Message: 3 sent at: Thu Feb 05 09:27:43
 GMT 2009  ...
```

Enable SSL logging in the consumer

To enable SSL logging in the consumer, edit the Ant build file, `example/build.xml`, and set the `javax.net.debug` system property as follows:

```
<project ...>
    ...
     <target name="consumer" depends="compile" descrip
tion="Runs a simple consumer">
        ...
          <java classname="ConsumerTool" fork="yes"
maxmemory="100M">
            ...
            <sysproperty key="javax.net.debug" value="ssl"/>
            ...
          </java>
    </target>
    ...
</project>
```

Now run the consumer tool using the same command as before:

```
ant consumer -Durl=ssl://localhost:61617 -Dmax=100
```

You should see some output like the following:

```
...
     [java] setting up default SSLSocketFactory
     [java] use default SunJSSE impl class:
com.sun.net.ssl.internal.ssl.SSLSocketFactoryImpl
     [java] class com.sun.net.ssl.internal.ssl.SSLSocketFact
oryImpl is loaded
     [java] keyStore is : ../conf/client.ks
     [java] keyStore type is : jks
     [java] keyStore provider is :
     [java] init keystore
     [java] init keymanager of type SunX509
     [java] ***
     [java] found key for : client
     [java] chain [0] = [
     [java] [
     [java]   Version: V1
     [java]   Subject: CN=Unknown, OU=client, O=Unknown,
L=Unknown, ST=Unknown, C=Unknown
     [java]   Signature Algorithm: MD5withRSA, OID =
1.2.840.113549.1.1.4
...
```

Chapter 2. Managing Certificates

TLS authentication uses X.509 certificates—a common, secure and reliable method of authenticating your application objects. This chapter explains how to create X.509 certificates that identify your FUSE Message Broker applications.

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 demonstration certificates supplied with FUSE Message Broker 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. This chapter describes the set of certificates required by a FUSE Message Broker application and describes how to replace the default certificates.

The 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 A on page 101 for more details about DNs.

Certification Authorities

Choice of CAs

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 FUSE Message Broker 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:

- A *commercial CA* is a company that signs certificates for many systems.
- A *private CA* is a trusted node that you set up and use to sign certificates for your system only.

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 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?

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 (`<eay@cryptsoft.com>`). Complete license information can be found in Appendix B on page 107 . 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 "Creating Your Own Certificates" on page 44 .

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 FUSE Message Broker 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.

Certificate Chaining

Certificate chain

A *certificate chain* is a sequence of certificates, where each certificate in the chain is signed by the subsequent certificate.

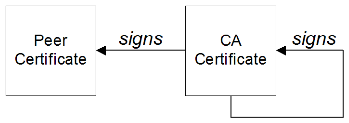
Self-signed certificate

The last certificate in the chain is normally a *self-signed certificate*—a certificate that signs itself.

Example

Figure 2.1 on page 40 shows an example of a simple certificate chain.

Figure 2.1. A Certificate Chain of Depth 2



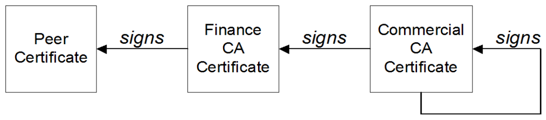
Chain of trust

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 2.2 on page 40 shows what this certificate chain looks like.

Figure 2.2. 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.

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.

Note

FUSE Message Broker does not implement the HTTPS URL integrity check. You can achieve a good degree of trust on the client side, however, by deploying clients with very restrictive CA certificates.

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 subectAltName

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.progress.com/secure
```

The corresponding server certificate would have the following subject DN:

```
C=IE,ST=Co. Dublin,L=Dublin,O=Progress,
OU=System,CN=www.progress.com
```

Where the CN has been set to the host name, `www.progress.com`.

For details of how to set the subject DN in a new certificate, see "Use the CA to Create Signed Certificates in a Java Keystore" on page 50 and "Use the CA to Create Signed Certificates in a Java Keystore" on page 50 .

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.progress.com
fusesource.com
```

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.progress.com,DNS:fusesource.com
```

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:*.progress.com
```

This certificate identity matches any three-component host name in the domain `progress.com`. For example, the wildcarded host name matches either `www.progress.com` or `fusesource.com`, but does not match `www.fusesource.com`.

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 `*progress.com`, your certificate could be used on *any* domain that ends in the letters `progress`.

Creating Your Own Certificates

Prerequisites

OpenSSL utilities

The steps described in this section are based on the OpenSSL command-line utilities from the OpenSSL project. Further documentation of the OpenSSL command-line utilities can be obtained at http://www.openssl.org/docs.

Sample CA directory structure

For the purposes of illustration, the CA database is assumed to have the following directory structure:

```
X509CA/ca

X509CA/certs

X509CA/newcerts

X509CA/crl
```

Where *X509CA* is the parent directory of the CA database.

Set Up Your Own CA

Substeps to perform

This section describes how to set up your own private CA. Before setting up a CA for a real deployment, read the additional notes in "Choosing a host for a private certification authority" on page 39 .

To set up your own CA, perform the following steps:

1. "Add the bin directory to your PATH"
2. "Create the CA directory hierarchy"
3. "Copy and edit the openssl.cnf file"
4. "Initialize the CA database"
5. "Create a self-signed CA certificate and private key"

Add the bin directory to your PATH

On the secure CA host, add the OpenSSL `bin` directory to your path:

Windows

```
> set PATH=OpenSSLDir\bin;%PATH%
```

UNIX

```
% PATH=OpenSSLDir/bin:$PATH; export PATH
```

This step makes the **openssl** utility available from the command line.

Create the CA directory hierarchy

Create a new directory, *X509CA*, to hold the new CA. This directory is used to hold all of the files associated with the CA. Under the *X509CA* directory, create the following hierarchy of directories:

```
X509CA/ca

X509CA/certs

X509CA/newcerts
```

X509CA/crl

Copy and edit the openssl.cnf file

Copy the sample `openssl.cnf` from your OpenSSL installation to the *X509CA* directory.

Edit the `openssl.cnf` to reflect the directory structure of the *X509CA* directory, and to identify the files used by the new CA.

Edit the `[CA_default]` section of the `openssl.cnf` file to look like the following:

```
#############################################################
[ 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
...
```

You might decide to edit other details of the OpenSSL configuration at this point—for more details, see the OpenSSL documentation.

Initialize the CA database

In the *X509CA* directory, initialize two files, `serial` and `index.txt`.

Windows

To initialize the `serial` file in Windows, enter the following command:

```
> echo 01 > serial
```

To create an empty file, `index.txt`, in Windows start Windows Notepad at the command line in the *X509CA* directory, as follows:

```
> notepad index.txt
```

In response to the dialog box with the text, `Cannot find the text.txt file. Do you want to create a new file?`, click **Yes**, and close Notepad.

UNIX

To initialize the `serial` file and the `index.txt` file in UNIX, enter the following command:

```
% echo "01" > serial
% touch index.txt
```

These files are used by the CA to maintain its database of certificate files.

Note

The `index.txt` file must initially be completely empty, not even containing white space.

Create a self-signed CA certificate and private key

Create a new self-signed CA certificate and private key with the following command:

```
openssl req -x509 -new -config X509CA/openssl.cnf -days 365 -out
X509CA/ca/new_ca.pem -keyout X509CA/ca/new_ca_pk.pem
```

The command prompts you for a pass phrase for the CA private key and details of the CA distinguished name. For example:

```
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) []:Progress
Organizational Unit Name (eg, section) []:Finance
Common Name (eg, YOUR name) []:Gordon Brown
Email Address []:gbrown@progress.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` (see the preceding step).

You are now ready to sign certificates with your CA.

Use the CA to Create Signed Certificates in a Java Keystore

Substeps to perform

To create and sign a certificate in a Java keystore (JKS), *CertName*.jks, perform the following substeps:

1. "Add the Java bin directory to your PATH"
2. "Generate a certificate and private key pair"
3. "Create a certificate signing request"
4. "Sign the CSR"
5. "Convert to PEM format"
6. "Concatenate the files"
7. "Update keystore with the full certificate chain"
8. "Repeat steps as required"

Add the Java bin directory to your PATH

If you have not already done so, add the Java bin directory to your path:

Windows

```
> set PATH=JAVA_HOME\bin;%PATH%
```

UNIX

```
% PATH=JAVA_HOME/bin:$PATH; export PATH
```

This step makes the **keytool** utility available from the command line.

Generate a certificate and private key pair

Open a command prompt and change directory to the directory where you store your keystore files, *KeystoreDir*. Enter the following command:

```
keytool -genkey -dname "CN=Alice, OU=Engineering, O=Progress,
 ST=Co. Dublin, C=IE" -validity 365 -alias CertAlias -keypass
 CertPassword -keystore CertName.jks -storepass CertPassword
```

This keytool command, invoked with the -genkey option, generates an X.509 certificate and a matching private key. The certificate and the key are both placed in a *key entry* in a newly created keystore, *CertName*.jks. Because the specified keystore, *CertName*.jks, did not exist prior to issuing the command, **keytool** implicitly creates a new keystore.

The -dname and -validity flags define the contents of the newly created X.509 certificate, specifying the subject DN and the days before expiration respectively. For more details about DN format, see Appendix A on page 101.

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 "Sign the CSR" on page 51).

Create a certificate signing request

Create a new certificate signing request (CSR) for the *CertName*.jks certificate, as follows:

```
keytool -certreq -alias CertAlias -file CertName_csr.pem -key
pass CertPassword -keystore CertName.jks -storepass CertPassword
```

This command exports a CSR to the file, *CertName*_csr.pem.

Sign the CSR

Sign the CSR using your CA, as follows:

```
openssl ca -config X509CA/openssl.cnf -days 365 -in Cert
Name_csr.pem -out CertName.pem
```

To sign the certificate successfully, you must enter the CA private key pass phrase (see "Set Up Your Own CA" on page 46).

Note

If you want to sign the CSR using a CA certificate *other* than the default CA, use the `-cert` and `-keyfile` options to specify the CA certificate and its private key file, respectively.

Convert to PEM format

Convert the signed certificate, *`CertName`*`.pem`, to PEM only format, as follows:

```
openssl x509 -in CertName.pem -out CertName.pem -outform PEM
```

Concatenate the files

Concatenate the CA certificate file and *`CertName`*`.pem` certificate file, as follows:

Windows

```
copy CertName.pem + X509CA\ca\new_ca.pem CertName.chain
```

UNIX

```
cat CertName.pem X509CA/ca/new_ca.pem > CertName.chain
```

Update keystore with the full certificate chain

Update the keystore, *`CertName`*`.jks`, by importing the full certificate chain for the certificate, as follows:

```
keytool -import -file CertName.chain -keypass CertPassword
-keystore CertName.jks -storepass CertPassword
```

Repeat steps as required

Repeat steps 2 through 7, to create a complete set of certificates for your system.

Adding Trusted CAs to a Java Trust Store

CA certificate deployment in the FUSE Message Broker configuration file

To deploy one or more trusted root CAs using a Java keystore file, perform the following steps:

1. 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 (for details of how to generate your own CA certificates, see "Set Up Your Own CA" on page 46). 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.

2. Given a CA certificate, `cacert.pem`, in PEM format, you can add the certificate to a JKS truststore (or create a new truststore) by entering the following command:

   ```
   keytool -import -file cacert.pem -alias CAAlias -keystore
    truststore.ts -storepass StorePass
   ```

   Where *`CAAlias`* is a convenient tag that enables you to access this particular CA certificate using the `keytool` utility. The file, `truststore.ts`, is a keystore file containing CA certificates—if this file does not already exist, the `keytool` utility creates one. The *`StorePass`* password provides access to the keystore file, `truststore.ts`.

3. Repeat step 2 as necessary, to add all of the CA certificates to the truststore file, `truststore.ts`.

Chapter 3. Authentication and Authorization

FUSE Message Broker has a flexible authentication model, which includes support for several different JAAS authentication plug-ins. In addition, you can optionally enable an authorization feature which implements group-based access control and allows you to control access at the granularity level of destinations or of individual messages.

Programming Client Credentials

Overview

Currently, for Java clients of the FUSE Message Broker, you must set the username/password credentials by programming. The ActiveMQConnectionFactory provides several alternative methods for specifying the username and password, as follows:

```
ActiveMQConnectionFactory(String userName, String password,
String brokerURL);
ActiveMQConnectionFactory(String userName, String password,
URI brokerURL);
Connection createConnection(String userName, String password);
QueueConnection createQueueConnection(String userName, String
 password);
TopicConnection createTopicConnection(String userName, String
 password);
```

Of these methods, `createConnection(String userName, String password)` is the most flexible, since it enables you to specify credentials on a connection-by-connection basis.

Setting login credentials for the Openwire protocol

To specify the login credentials on the client side, pass the username/password credentials as arguments to the `ActiveMQConnectionFactory.createConnection()` method, as shown in the following example:

```
// Java
...
public void run() {
  ...
   user = "jdoe";
   password = "secret";
  ActiveMQConnectionFactory connectionFactory = new ActiveMQ
ConnectionFactory(url);
  Connection connection = connectionFactory.createConnec
tion(user, password);
  ...
  }
```

Configuring Credentials for Broker Components

Overview

Once authentication is enabled in the broker, every application component that opens a connection to the broker must be configured with credentials. This includes some standard broker components, which are normally configured using Spring XML. To enable you to set credentials on these components, the XML schemas for these components have been extended as described in this section.

Default credentials for broker components

For convenience, you can configure default credentials for the broker components by setting the `activemq.username` property and the `activemq.password` property in the `conf/credentials.properties` file. By default, this file has the following contents:

```
activemq.username=system
activemq.password=manager
```

Command agent

You can configure the command agent with credentials by setting the `username` attribute and the `password` attribute on the `commandAgent` element in the broker configuration file. By default, the command agent is configured to pick up its credentials from the `activemq.username` property and the `activemq.password` property as shown in the following example:

```
<beans>
  ...
  <commandAgent xmlns="http://activemq.apache.org/schema/core"

                brokerUrl="vm://localhost"
                username="${activemq.username}"
                password="${activemq.password}" />

  ...
</beans>
```

FUSE Mediation Router

The default broker configuration file contains an example of a FUSE Mediation Router route that is integrated with the broker. This sample route is defined as follows:

```
<beans>
  ...
  <camelContext id="camel" xmlns="http://act
```

```
ivemq.apache.org/camel/schema/spring">
    <package>org.foo.bar</package>
    <route>
      <from uri="activemq:example.A"/>
      <to uri="activemq:example.B"/>
    </route>
  </camelContext>
  ...
</beans>
```

The preceding route integrates with the broker using endpoint URIs that have the component prefix, `activemq:`. For example, the URI, `activemq:example.A`, represents a queue named `example.A` and the endpoint URI, `activemq:example.B`, represents a queue named `example.B`.

The integration with the broker is implemented by the Camel component with bean ID equal to `activemq`. When the broker has authentication enabled, it is necessary to configure this component with a `userName` property and a `password` property, as follows:

```
<beans>
  ...
  <bean id="activemq" class="org.apache.activemq.camel.compon
ent.ActiveMQComponent" >
    <property name="connectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFact
ory">
        <property name="brokerURL" value="vm://localhost?cre
ate=false&amp;waitForStart=10000" />
        <property name="userName" value="${activemq.user
name}"/>
        <property name="password" value="${activemq.pass
word}"/>
      </bean>
    </property>
  </bean>
  ...
</beans>
```

Simple Authentication Plug-In

Overview

The simple authentication plug-in provides the quickest way to enable authentication in a broker. With this approach, all of the user data is embedded in the broker configuration file. It is useful for testing purposes and for small-scale systems with relatively few users, but it does not scale well for large systems.

Broker configuration for simple authentication

Example 3.1 on page 59 shows how to configure simple authentication by adding a `simpleAuthenticationPlugin` element to the list of plug-ins in the broker configuration.

Example 3.1. Simple Authentication Configuration

```
<beans>
  <broker ...>
    ...
    <plugins>
        <simpleAuthenticationPlugin>
           <users>
              <authenticationUser username="system"
                              password="manager"
                                   groups="users,admins"/>
              <authenticationUser username="user"
                              password="password"
                                   groups="users"/>
              <authenticationUser username="guest"
                              password="password"
                              groups="guests"/>
           </users>
        </simpleAuthenticationPlugin>
    </plugins>
    ...
  </broker>

</beans>
```

For each user, add an `authenticationUser` element as shown, setting the `username`, `password`, and `groups` attributes. In order to authenticate a user successfully, the username/password credentials received from a client must match the corresponding attributes in one of the `authenticationUser` elements. The `groups` attribute assigns a user to one or more groups (formatted as a comma-separated list). If authorization is enabled, the assigned groups are used to check whether a user has permission to invoke certain operations. If authorization is not enabled, the groups are ignored.

JAAS Authentication

Introduction to JAAS

Overview

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. In particular, JAAS defines a general configuration file format that can be used to configure any custom login modules.

For background information about JAAS, see the JAAS Reference Guide[1].

JAAS login configuration

The JAAS login configuration file has the general format shown in Example 3.2 on page 62.

Example 3.2. JAAS Login Configuration File Format

```
/* JAAS Login Configuration */

LoginEntry {
    ModuleClass Flag Option="Value" Option="Value" ... ;
    ModuleClass Flag Option="Value" Option="Value" ... ;
    ...
};
LoginEntry {
    ModuleClass Flag Option="Value" Option="Value" ... ;
    ModuleClass Flag Option="Value" Option="Value" ... ;
    ...
};
...
```

Where the file format can be explained as follows:

- *`LoginEntry`* labels a single entry in the login configuration. An application is typically configured to search for a particular *`LoginEntry`* label (for example, in FUSE Message Broker the *`LoginEntry`* label to use is specifed in the broker configuration file). Each login entry contains a list of login modules that are invoked in order.
- *`ModuleClass`* is the fully-qualified class name of a JAAS login module. For example, `org.apache.activemq.jaas.PropertiesLoginModule` is the

[1] http://java.sun.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html

class name of FUSE Message Broker's JAAS simple authentication login module.

- `Flag` determines how to react when the current login module reports an authentication failure. The `Flag` can have one of the following values:
  - `required`—authentication of this login module must succeed. Always proceed to the next login module in this entry, irrespective of success or failure.
  - `requisite`—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.
  - `sufficient`—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.
  - `optional`—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.
- `Option="Value"`—after the `Flag`, you can pass zero or more option settings to the login module. The options are specified in the form of a space-separated list, where each option has the form `Option="Value"`. The login module line is terminated by a semicolon, `;`.

Location of the login configuration file

There are two general approaches to specifying the location of the JAAS login configuration file, as follows:

- *Set a system property*—set the value of the system property, `java.security.auth.login.config`, to the location of the login configuration file. For example, you could set this system property on the command line, as follows:

```
java -Djava.security.auth.login.config=/var/activemq/con
fig/login.config ...
```

- *Configure the JDK*—if the relevant system property is not set, JAAS checks the `$JAVA_HOME/jre/lib/security/java.security` security properties file, looking for entries of the form:

  ```
  login.config.url.1=file:C:/activemq/config/login.config
  ```

  If there is more than one such entry, `login.config.url.`*`n`*, the entries must be consecutively numbered. The contents of the login files listed in `java.security` are merged into a single configuration.

In addition to these general approaches, FUSE Message Broker defines a custom approach to locating the JAAS login configuration. If the system property is not specified, the broker searches the CLASSPATH for a file named, `login.config`.

JAAS Simple Authentication Plug-In

Overview

The JAAS simple authentication plug-in provides a light-weight authentication implementation, where the relevant user security data is stored in a pair of flat files. This is convenient for demonstrations and testing, but for an enterprise system, the integration with LDAP is preferable (see "JAAS LDAP Authentication Plug-In" on page 72).

Specifying the login.config file location

The simplest way to make the login configuration available to JAAS is to add the directory containing the file, `login.config`, to your CLASSPATH.

Alternatively, you can set the `java.security.auth.login.config` system property at the command line, setting it to the pathname of the login configuration file (for example, edit the `bin/activemq script`, adding an option of the form, `-Djava.security.auth.login.config=`*`Value`* to the Java command line). If you are working on the Windows platform, note that the pathname of the login configuration file must use forward slashes, `/`, in place of backslashes, `\`.

login.config file

The following `PropertiesLogin` login entry shows how to configure JAAS simple authentication in the `login.config` file:

Example 3.3. JAAS Login Entry for Simple Authentication

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

JAAS simple authentication is configured by the `org.apache.activemq.jaas.PropertiesLoginModule` login module. The options supported by this login module are as follows:

- `debug`—boolean debugging flag. If `true`, enable debugging. This is used only for testing or debugging. Normally, it should be set to `false`, or omitted.

- `org.apache.activemq.jaas.properties.user`—specifies the location of the user properties file (relative to the directory containing the login configuration file).
- `org.apache.activemq.jaas.properties.group`—specifies the location of the group properties file (relative to the directory containing the login configuration file).

users.properties file

In the context of the simple authentication plug-in, the `users.properties` file consists of a list of properties of the form, *`UserName`*`=`*`Password`*. For example, to define the users, `system`, `user`, and `guest`, you could create a file like the following:

```
system=manager
user=password
guest=password
```

groups.properties file

The `groups.properties` file consists of a list of properties of the form, *`Group`*`=`*`UserList`*, where *`UserList`* is a comma-separated list of users. For example, to define the groups, `admins`, `users`, and `guests`, you could create a file like the following:

```
admins=system
users=system,user
guests=guest
```

Enable the JAAS simple authentication plug-in

To enable the JAAS simple authentication plug-in, add the `jaasAuthenticationPlugin` element to the list of plug-ins in the broker configuration file, as shown:

```
<beans>
  <broker ...>
    ...
    <plugins>
     <jaasAuthenticationPlugin configuration="PropertiesLogin"
 />
    </plugins>
    ...
  </broker>
</beans>
```

The `configuration` attribute specifies the label of a login entry from the login configuration file (for example, see Example 3.3 on page 65). In the preceding example, the `PropertiesLogin` login entry is selected.

JAAS Certificate Authentication Plug-In

Overview

The JAAS certificate authentication plug-in must be used in combination with an SSL/TLS protocol (for example, `ssl:` or `https:`) and the clients must be configured with their own certificate. In this scenario, authentication is actually performed during the SSL/TLS handshake, *not* directly by the JAAS certificate authentication plug-in. The role of the plug-in is as follows:

- To further constrain the set of acceptable users, because only the user DNs explicitly listed in the relevant properties file are eligible to be authenticated.
- To associate a list of groups with the received user identity, facilitating integration with the authorization feature.
- To *require* the presence of an incoming certificate (by default, the SSL/TLS layer is configured to treat the presence of a client certificate as optional).

Specifying the login.config file location

The simplest way to make the login configuration available to JAAS is to add the directory containing the file, `login.config`, to your CLASSPATH.

Alternatively, you can set the `java.security.auth.login.config` system property at the command line, setting it to the pathname of the login configuration file (for example, edit the `bin/activemq script`, adding an option of the form, `-Djava.security.auth.login.config=`*`Value`* to the Java command line). If you are working on the Windows platform, note that the pathname of the login configuration file must use forward slashes, `/`, in place of backslashes, `\`.

login.config file

The following `CertLogin` login entry shows how to configure JAAS certificate authentication in the `login.config` file:

Example 3.4. JAAS Login Entry for Certificate Authentication

```
CertLogin {
    org.apache.activemq.jaas.TextFileCertificateLoginModule
required
        debug=true
        org.apache.activemq.jaas.textfiledn.user="users.prop
erties"
       org.apache.activemq.jaas.textfiledn.group="groups.prop
erties";
};
```

JAAS simple authentication is configured by the `org.apache.activemq.jaas.TextFileCertificateLoginModule` login module. The options supported by this login module are as follows:

- `debug`—boolean debugging flag. If `true`, enable debugging. This is used only for testing or debugging. Normally, it should be set to `false`, or omitted.
- `org.apache.activemq.jaas.textfiledn.user`—specifies the location of the user properties file (relative to the directory containing the login configuration file).
- `org.apache.activemq.jaas.textfiledn.group`—specifies the location of the group properties file (relative to the directory containing the login configuration file).

users.properties file

In the context of the certificate authentication plug-in, the `users.properties` file consists of a list of properties of the form, *`UserName=StringifiedSubjectDN`*. For example, to define the users, `system`, `user`, and `guest`, you could create a file like the following:

```
system=CN=system,O=Progress,C=US
user=CN=humble user,O=Progress,C=US
guest=CN=anon,O=Progress,C=DE
```

Each username is mapped to a subject DN, encoded as a string (where the string encoding is specified by RFC 2253[2]). For example, the `system` username is mapped to the `CN=system,O=Progress,C=US` subject DN. When performing authentication, the plug-in extracts the subject DN from the received certificate, converts it to the standard string format, and compares it with the subject DNs in the `users.properties` file by testing for *string equality*. Consequently, you must be careful to ensure that the subject DNs appearing in the `users.properties` file are an *exact* match for the subject DNs extracted from the user certificates.

Note

Technically, there is some residual ambiguity in the DN string format. For example, the `domainComponent` attribute could be represented in a string either as the string, `DC`, or as the OID,

[2] http://www.ietf.org/rfc/rfc2253.txt

0.9.2342.19200300.100.1.25. Normally, you do not need to worry about this ambiguity. But it could potentially be a problem, if you changed the underlying implementation of the Java security layer.

Obtaining the subject DNs

The easiest way to obtain the subject DNs from the user certificates is by invoking the keytool utility to print the certificate contents. To print the contents of a certificate in a keystore, perform the following steps:

1. Export the certificate from the keystore file into a temporary file. For example, to export the certificate with alias broker-localhost from the broker.ks keystore file, enter the following command:

   ```
   keytool -export -file broker.export -alias broker-localhost
    -keystore broker.ks -storepass password
   ```

   After running this command, the exported certificate is in the file, broker.export.

2. Print out the contents of the exported certificate. For example, to print out the contents of broker.export, enter the following command:

   ```
   keytool -printcert -file broker.export
   ```

   Which should produce output like the following

   ```
   Owner: CN=localhost, OU=broker, O=Unknown, L=Unknown,
   ST=Unknown, C=Unknown
   Issuer: CN=localhost, OU=broker, O=Unknown, L=Unknown,
   ST=Unknown, C=Unknown
   Serial number: 4537c82e
   Valid from: Thu Oct 19 19:47:10 BST 2006 until: Wed Jan 17
    18:47:10 GMT 2007
   Certificate fingerprints:
            MD5:
   3F:6C:0C:89:A8:80:29:CC:F5:2D:DA:5C:D7:3F:AB:37
            SHA1:
   F0:79:0D:04:38:5A:46:CE:86:E1:8A:20:1F:7B:AB:3A:46:E4:34:5C
   ```

   The string following Owner: gives the subject DN, but you must remove the spaces appearing after each of the commas. For example, the preceding output represents a certificate with subject DN equal to `CN=localhost,OU=broker,O=Unknown,L=Unknown,ST=Unknown,C=Unknown`.

groups.properties file

The `groups.properties` file consists of a list of properties of the form, *`Group=UserList`*, where *`UserList`* is a comma-separated list of users. For example, to define the groups, `admins`, `users`, and `guests`, you could create a file like the following:

```
admins=system
users=system,user
guests=guest
```

Enable the JAAS certificate authentication plug-in

To enable the JAAS certificate authentication plug-in, add the `jaasCertificateAuthenticationPlugin` element to the list of plug-ins in the broker configuration file, as shown:

```
<beans>
  <broker ...>
    ...
    <plugins>
      <jaasCertificateAuthenticationPlugin configura
tion="CertLogin" />
    </plugins>
    ...
  </broker>
</beans>
```

The `configuration` attribute specifies the label of a login entry from the login configuration file (for example, see Example 3.4 on page 68). In the preceding example, the `CertLogin` login entry is selected.

JAAS LDAP Authentication Plug-In

Overview

The LDAP authentication plug-in enables you to perform authentication by checking the incoming credentials against user data stored in a central X.500 directory server. For systems that already have an X.500 directory server in place, this means that you can rapidly integrate FUSE Message Broker with the existing security database and user accounts can be managed using the X.500 system.

Specifying the login.config file location

The simplest way to make the login configuration available to JAAS is to add the directory containing the file, `login.config`, to your CLASSPATH.

Alternatively, you can set the `java.security.auth.login.config` system property at the command line, setting it to the pathname of the login configuration file (for example, edit the `bin/activemq script`, adding an option of the form, `-Djava.security.auth.login.config=`*`Value`* to the Java command line). If you are working on the Windows platform, note that the pathname of the login configuration file must use forward slashes, `/`, in place of backslashes, `\`.

login.config file

Example 3.5 on page 72 shows an example of a login entry for the LDAP authentication plug-in, connecting to a directory repository with the URL, `ldap://localhost:10389`.

Example 3.5. LDAP Login Entry

```
LDAPLogin {
    org.apache.activemq.jaas.LDAPLoginModule required
        debug=true
       initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory

        connectionURL="ldap://localhost:10389"
        connectionUsername="uid=admin,ou=system"
        connectionPassword=secret
        connectionProtocol=""
        authentication=simple
        userBase="ou=users,ou=system"
        userSearchMatching="(uid={0})"
        userSearchSubtree=false
        roleSearchMatching="(uid={1})"
        ;
};
```

The preceding login entry is configured to search for users under the `ou=users,ou=system` level in the Directory Information Tree (DIT). For example, an incoming username, `jdoe`, would match the entry whose DN is `uid=jdoe,ou=users,ou=system`.

LDAP login entry options

The LDAP login entry supports the following options:

- `debug`—boolean debugging flag. If `true`, enable debugging. This is used only for testing or debugging. Normally, it should be set to `false`, or omitted.
- `initialContextFactory`—*(mandatory)* must always be set to `com.sun.jndi.ldap.LdapCtxFactory`.
- `connectionURL`—*(mandatory)* specify the location of the directory server using an ldap URL, `ldap://`*`Host`*`:`*`Port`*. You can optionally qualify this URL, by adding a forward slash, `/`, followed by the DN of a particular node in the directory tree. For example, `ldap://ldapserver:10389/ou=system`.
- `connectionUsername`—*(optional)*the DN of the user that opens the connection to the directory server. For example, `uid=admin,ou=system`.

  Directory servers generally require clients to present username/password credentials in order to open a connection.
- `connectionPassword`—*(optional)*the password that matches the DN from `connectionUsername`. In the directory server, in the DIT, the password is normally stored as a `userPassword` attribute in the corresponding directory entry.
- `connectionProtocol`—*(mandatory)*currently, the only supported value is a blank string. In future, this option will allow you to select the Secure Socket Layer (SSL) for the connection to the directory server.

  Note

  This option *must* be set explicitly to an empty string, because it has no default value.

- `authentication`—*(mandatory)*can take either of the values, `simple` or `none`.
- `userBase`—*(mandatory)*selects a particular subtree of the DIT to search for user entries. The subtree is specified by a DN, which specifes the base node of the subtree. For example, by setting this option to `ou=users,ou=system`, the search for user entries is restricted to the subtree beneath the `ou=users,ou=system` node.
- `userSearchMatching`—*(mandatory)*specifies an LDAP search filter, which is applied to the subtree selected by `userBase`. Before passing to the LDAP search operation, the string value you provide here is subjected to *string substitution*, as implemented by the `java.text.MessageFormat` class. Essentially, this means that the special string, `{0}`, is substituted by the username, as extracted from the incoming client credentials.

  After substitution, the string is interpreted as an LDAP search filter, where the LDAP search filter syntax is defined by the IETF standard, RFC 2254[3]. A short introduction to the search filter syntax is available from Sun's JNDI tutorial, Search Filters[4].

  For example, if this option is set to `(uid={0})` and the received username is `jdoe`, the search filter becomes `(uid=jdoe)` after string substitution. If the resulting search filter is applied to the subtree selected by the user base, `ou=users,ou=system`, it would match the entry, `uid=jdoe,ou=users,ou=system` (and possibly more deeply nested entries, depending on the specified search depth—see the `userSearchSubtree` option).
- `userSearchSubtree`—*(optional)*specify the search depth for user entries, relative to the node specified by `userBase`. This option can take boolean values, as follows:
  - `false`—*(default)* try to match one of the child entries of the `userBase` node (maps to `javax.naming.directory.SearchControls.ONELEVEL_SCOPE`).

[3] http://www.ietf.org/rfc/rfc2254.txt
[4] http://java.sun.com/products/jndi/tutorial/basics/directory/filter.html

  - `true`—try to match *any* entry belonging to the subtree of the `userBase` node (maps to `javax.naming.directory.SearchControls.SUBTREE_SCOPE`).

- `userRoleName`—*(optional)*specifies the name of the multi-valued attribute of the user entry that contains a list of role names for the user (where the role names are interpreted as group names by the broker's authorization plug-in). If you omit this option, no role names are extracted from the user entry.

- `roleBase`—if you want to store role data directly in the directory server, you can use a combination of role options (`roleBase`, `roleSearchMatching`, `roleSearchSubtree`, and `roleName`) as an alternative to (or in addition to) specifying the `userRoleName` option.

  This option selects a particular subtree of the DIT to search for role entries. The subtree is specified by a DN, which specifes the base node of the subtree. For example, by setting this option to `ou=roles,ou=system`, the search for user entries is restricted to the subtree beneath the `ou=roles,ou=system` node.

- `roleSearchMatching`—*(mandatory)*specifies an LDAP search filter, which is applied to the subtree selected by `roleBase`. This works in a similar manner to the `userSearchMatching` option, except that it supports *two* substitution strings, as follows:

  - `{0}` substitutes the full DN of the matched user entry (that is, the result of the user search). For example, for the user, `jdoe`, the substituted string could be `uid=jdoe,ou=users,ou=system`.

  - `{1}` substitutes the received username. For example, `jdoe`.

  For example, if this option is set to `(uid={1})` and the received username is `jdoe`, the search filter becomes `(uid=jdoe)` after string substitution. If the resulting search filter is applied to the subtree selected by the role base, `ou=roles,ou=system`, it would match the entry, `uid=jdoe,ou=roles,ou=system` (and possibly more deeply nested entries, depending on the specified search depth—see the `roleSearchSubtree` option).

Note

This option must always be set, *even if role searching is disabled*, because it has no default value.

- `roleSearchSubtree`—*(optional)*specify the search depth for role entries, relative to the node specified by `roleBase`. This option can take boolean values, as follows:
  - `false`—*(default)* try to match one of the child entries of the `roleBase` node (maps to `javax.naming.directory.SearchControls.ONELEVEL_SCOPE`).
  - `true`—try to match *any* entry belonging to the subtree of the `roleBase` node (maps to `javax.naming.directory.SearchControls.SUBTREE_SCOPE`).
- `roleName`—*(optional)*specifies the name of the multi-valued attribute of the role entry that contains a list of role names for the current user. If you omit this option, the role search feature is effectively disabled.

Creating a user entry in the directory

Add user entries under the node specified by the `userBase` option. When creating a new user entry in the directory, choose an object class that supports the `userPassword` attribute (for example, the `person` or `inetOrgPerson` object classes are typically suitable). After creating the user entry, add the `userPassword` attribute, to hold the user's password.

Adding roles to the user entry

If you want to add roles to user entries, you will probably need to customize the directory schema, by adding a suitable attribute type to the user entry's object class. The chosen attribute type must be capable of handling multiple values.

Enable the JAAS LDAP authentication plug-in

To enable the JAAS LDAP authentication plug-in, add the `jaasAuthenticationPlugin` element to the list of plug-ins in the broker configuration file, as shown:

```
<beans>
  <broker ...>
```

```
    ...
    <plugins>
      <jaasAuthenticationPlugin configuration="LDAPLogin" />
    </plugins>
    ...
  </broker>
</beans>
```

The `configuration` attribute specifies the label of a login entry from the login configuration file (for example, see Example 3.5 on page 72). In the preceding example, the `LDAPLogin` login entry is selected.

Broker-to-Broker Authentication

Overview

If you are deploying your brokers in a cluster configuration, and one or more of the brokers is configured to require authentication, then it is necessary to equip *all* of the brokers in the cluster with the appropriate credentials, so that they can all talk to each other.

Configuring the network connector

Given two brokers, Broker A and Broker B, where Broker A is configured to perform authentication, you can configure Broker B to log on to Broker A by setting the `userName` attribute and the `password` attribute in the `networkConnector` element, as follows:

```
<beans ...>
    <broker ...>
        ...
        <networkConnectors>
            <networkConnector name="BrokerABridge"
                              userName="user"
                              password="password"
                              uri="stat
ic://(ssl://brokerA:61616)"/>
            ...
        </networkConnectors>

        ...
    </broker>
</beans>
```

If Broker A is configured to connect to Broker B, Broker A's `networkConnector` element must also be configured with username/password credentials, even though Broker B is not configured to perform authentication. Broker A's authentication plug-in checks for Broker A's username. For example, if Broker A has its authentication configured by a `simpleAuthenticationPlugin` element, Broker A's username must appears in this element.

Authorization Plug-In

Overview

In a security system without authorization, every successfully authenticated user would have unrestricted access to every queue and every topic in the broker. Using the authorization plug-in, on the other hand, you can restrict access to specific destinations based on a user's group membership.

Configuring the authorization plug-in

To configure the authorization plug-in, add an `authorizationPlugin` element to the list of plug-ins in the broker configuration, as shown in Example 3.6 on page 79.

Example 3.6. Authorization Plug-In Configuration

```
<beans>
  <broker ...>
    ...
    <plugins>
      ...
      <authorizationPlugin>
        <map>
          <authorizationMap>
            <authorizationEntries>
              <authorizationEntry queue=">"
                                  read="admins"
                                  write="admins"
                                  admin="admins" />
              <authorizationEntry queue="USERS.>"
                                  read="users"
                                  write="users"
                                  admin="users" />
              <authorizationEntry queue="GUEST.>"
                                  read="guests"
                                  write="guests,users"
                                  admin="guests,users" />
              <authorizationEntry topic=">"
                                  read="admins"
                                  write="admins"
                                  admin="admins" />
              <authorizationEntry topic="USERS.>"
                                  read="users"
                                  write="users"
                                  admin="users" />
              <authorizationEntry topic="GUEST.>"
                                  read="guests"
                                  write="guests,users"
```

```
                                 admin="guests,users" />
            </authorizationEntries>
            <tempDestinationAuthorizationEntry>
                <tempDestinationAuthorizationEntry
                                 read="admins"
                                 write="admins"
                                 admin="admins"/>
            </tempDestinationAuthorizationEntry>
          </authorizationMap>
        </map>
      </authorizationPlugin>
    </plugins>
    ...
  </broker>

</beans>
```

The authorization plug-in contains two different kinds of entry, as follows:

- "Authorization entries for named destinations" on page 80.
- "Authorization entries for temporary destinations" on page 81.

Authorization entries for named destinations

A named destination is just an ordinary JMS queue or topic (these destinations are named, in contrast to temporary destinations which have no permanent identity).The authorization entries for ordinary destinations are defined by the `authorizationEntry` element, which supports the following attributes:

- `queue` or `topic`—you can specify *either* a `queue` or a `topic` attribute, but not both in the same element. To apply authorization settings to a particular queue or topic, simply set the relevant attribute equal to the queue or topic name. The greater-than symbol, `>`, acts as a wildcard. For example, an entry with, `queue="USERS.>"`, would match any queue name beginning with the `USERS.` string.
- `read`—specifies a comma-separated list of groups that have permission to *consume* messages from the matching destinations.
- `write`—specifies a comma-separated list of groups that have permission to *publish* messages to the matching destinations.

- `admin`—specifies a comma-separated list of groups that have permission to create destinations in the destination subtree.

Authorization entries for temporary destinations

A temporary destination is a special feature of JMS that enables you to create a queue for a particular network connection. The temporary destination exists only as long as the network connection remains open and, as soon as the connection is closed, the temporary destination is deleted on the server side. The original motivation for defining temporary destinations was to facilitate request-reply semantics on a destination, without having to define a dedicated reply destination.

Because temporary destinations have no name, the `tempDestinationAuthorizationEntry` element does not support any `queue` or `topic` attributes. The attributes supported by the `tempDestinationAuthorizationEntry` element are as follows:

- `read`—specifies a comma-separated list of groups that have permission to *consume* messages from all temporary destinations.
- `write`—specifies a comma-separated list of groups that have permission to *publish* messages to all temporary destinations.
- `admin`—specifies a comma-separated list of groups that have permission to create temporary destinations.

Programming Message-Level Authorization

Overview

In the preceding examples, the authorization step is performed at the time of connection creation and access is applied at the *destination* level of granularity. That is, the authorization step grants or denies access to particular queues or topics. It is conceivable, though, that in some systems you might want to grant or deny access at the level of individual *messages*, rather than at the level of destinations. For example, you might want to grant permission to all users to read from a certain queue, but some messages published to this queue should be accessible to administrators only.

You can achieve message-level authorization by configuring a *message authorization policy* in the broker configuration file. To implement this policy, you need to write some Java code.

Implement the MessageAuthorizationPolicy interface

Example 3.7 on page 82 shows an example of a message authorization policy that allows messages from the `WebServer` application to reach only the `admin` user, with all other users blocked from reading these messages. This example presupposes that the `WebServer` application is configured to set the `JMSXAppID` property in the message's JMS header.

Example 3.7. Implementation of MessageAuthorizationPolicy

```
// Java
package com.acme;
...

public class MsgAuthzPolicy implements MessageAuthorization
Policy {

  public boolean isAllowedToConsume(ConnectionContext context,
 Message message)
  {
    if (message.getProperty("JMSXAppID").equals("WebServer"))
 {
      if (context.getUserName().equals("admin")) {
        return true;
      }
      else {
        return false;
      }
    }
    return true;
  }
}
```

The `org.apache.activemq.broker.ConnectionContext` class stores details of the current client connection and the `org.apache.activemq.command.Message` class is essentially an implementation of the standard `javax.jms.Message` interface.

To install the message authorization policy, compile the preceding code, package it as a JAR file, and drop the JAR file into the `$ACTIVEMQ_HOME/lib` directory.

Configure the messageAuthorizationPolicy element

To configure the broker to install the message authorization policy from Example 3.7 on page 82, add the following lines to the broker configuration file, `conf/activemq.xml`, inside the `broker` element:

```
<broker>
  ...
  <messageAuthorizationPolicy>
    <bean class="com.acme.MsgAuthzPolicy"
         xmlns="http://www.springframework.org/schema/beans"/>

  </messageAuthorizationPolicy>
  ...
</broker>
```

Chapter 4. LDAP Authentication Tutorial

This chapter explains how to set up an X.500 directory server and configure the broker to use LDAP authentication.

Tutorial Overview

Overview

This tutorial is aimed at users who are unfamiliar with LDAP and the X.500 directory services. It covers all of the steps required to set up an X.500 directory service and use it as a repository of security data for performing authentication in a FUSE Message Broker application.

Tutorial stages

The tutorial consists of the following stages:

1. "Tutorial: Install a Directory Server and Browser" on page 87.
2. "Tutorial: Add User Entries to the Directory Server" on page 89.
3. "Tutorial: Enable LDAP Authentication in the Broker and its Clients" on page 97.

Tutorial: Install a Directory Server and Browser

Overview

This section describes how to install an X.500 directory server and browser client, which you can then use to test the LDAP authentication feature of FUSE Message Broker. For the purpose of this tutorial, we recommend using the relevant applications from the *Apache Directory* project.

Install Apache Directory Server

Apache Directory Server (ApacheDS) is an open-source implementation of an X.500 directory server. You can use this directory server as a store of security data for the LDAP authentication feature of FUSE Message Broker.

To install Apache Directory Server, download ApacheDS 1.5 from http://directory.apache.org/apacheds/1.5/downloads.html and run the installer. During the installation process, you will be asked whether or not to install a *default instance* of the directory server. Choose the default instance.

If you install on the Windows platform, 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[1] for starting and stopping the directory server.

Note

This tutorial was tested with version 1.5.4 of Apache Directory Studio.

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:

[1] http://directory.apache.org/apacheds/1.5/13-installing-and-starting-the-server.html

- *Standalone application*—download the standalone distribution from the Directory Studio downloads[2] page and follow the installation instructions from the Apache Directory Studio User Guide[3].
- *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[4].

[2] http://directory.apache.org/studio/downloads.html
[3] http://directory.apache.org/studio/static/users_guide/apache_directory_studio/download_install.html
[4] http://directory.apache.org/studio/static/users_guide/ldap_browser/gettingstarted_download_install.html

Tutorial: Add User Entries to the Directory Server

Overview

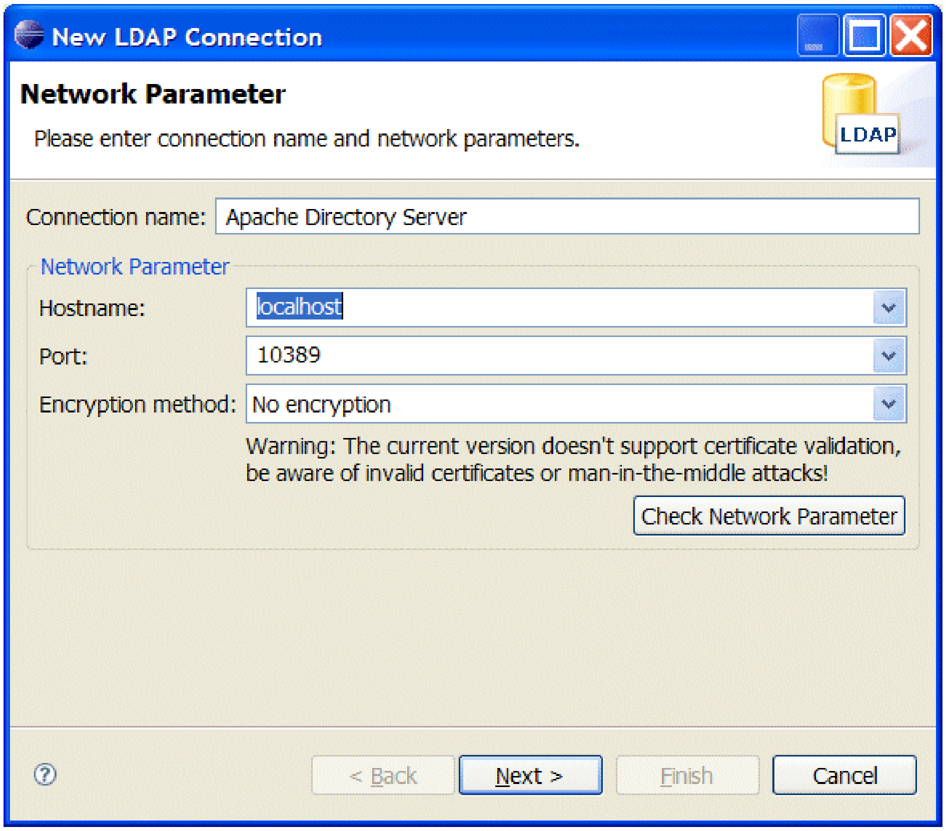
The basic prerequisite for using LDAP authentication in the broker is to have an X.500 directory server running and configured with a collection of user entries. For users who are unfamiliar with X.500 directory servers, this section briefly describes how to create user entries using the Apache Directory Studio as an administrative tool.

Steps to add a user entry

Perform the following steps to add a user entry to the directory server:

1. Ensure that the X.500 directory server is running (see "Install Apache Directory Server" on page 87).
2. Start the *LDAP Browser*, as follows:
   - If you installed the standalone version of Apache Directory Studio, double-click the relevant icon to launch the application.
   - If you installed the *LDAP Browser* plug-in into an existing Eclipse IDE, start Eclipse and open the LDAP perspective. To open the LDAP perspective, select **Window|Open Perspective|Other** and in the **Open Perspective** dialog, select LDAP and click **OK**.
3. Open a connection to the directory server. Right-click inside the **Connections** view in the lower left corner and select **New Connection**. The **New LDAP Connection** wizard opens.
4. Specify the network parameters for the new connection. In the **Connection name** field, enter `Apache Directory Server`. In the **Hostname** field enter the name of the host where the Apache Directory Server is running. In the Port field, enter the IP port of the directory server (for the default instance of the Apache directory server, this is 10389). Click **Next**.

Figure 4.1. New LDAP Connection Wizard



Please enter connection name and network parameters.

Connection name: Apache Directory Server

Hostname: localhost

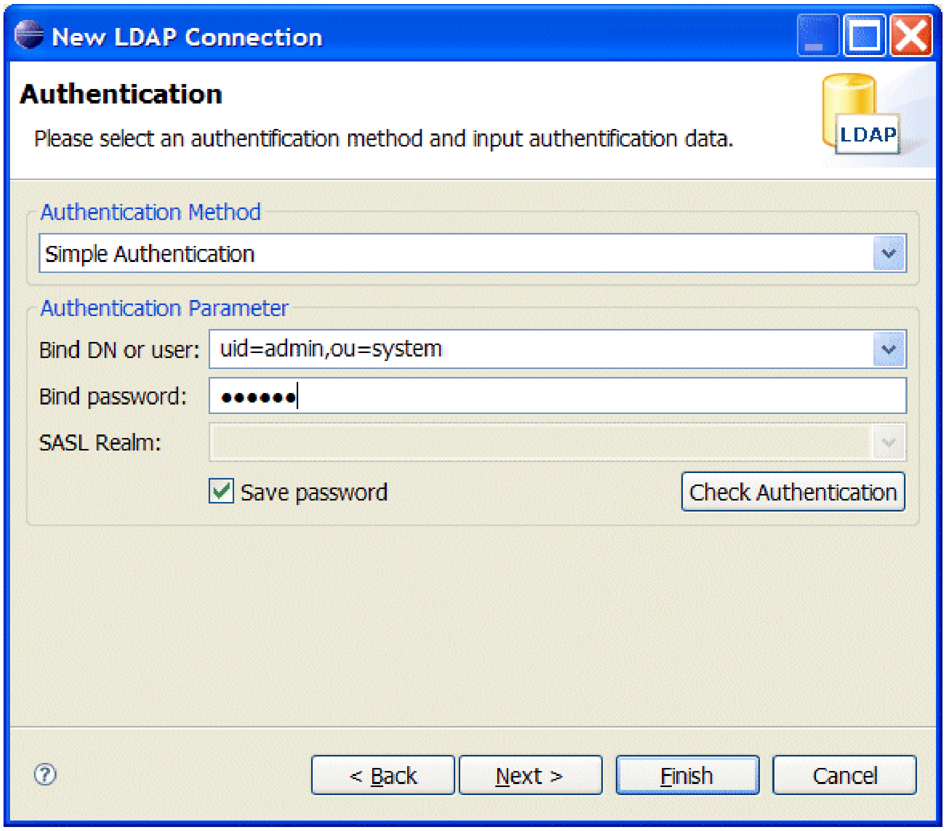
Port: 10389

Encryption method: No encryption

Warning: The current version doesn't support certificate validation, be aware of invalid certificates or man-in-the-middle attacks!

5. Enter the parameters for simple authentication. In the **Bind DN or user** field, enter the DN of the administrator's account on the directory server (for the default instance of the Apache directory server, this is `uid=admin,ou=system`). In the **Bind password** field, enter the administrator's password (for the default instance of the Apache directory server, the administrator's password is `secret`). Click **Finish**.

Figure 4.2. Authentication Step of New LDAP Connection



Authentication

Please select an authentification method and input authentification data.

Authentication Method

Simple Authentication

Authentication Parameter

Bind DN or user: uid=admin,ou=system

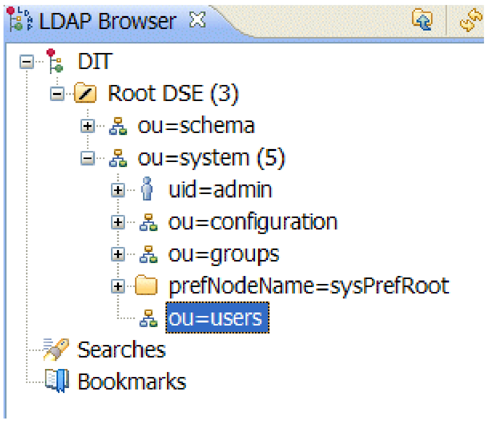
Bind password: ●●●●●●

SASL Realm:

☑ Save password

Check Authentication

6. If the connection is successfully established, you should see an outline of the Directory Information Tree (DIT) in the **LDAP Browser** view. In the **LDAP Browser** view, drill down to the `ou=users` node, as shown.

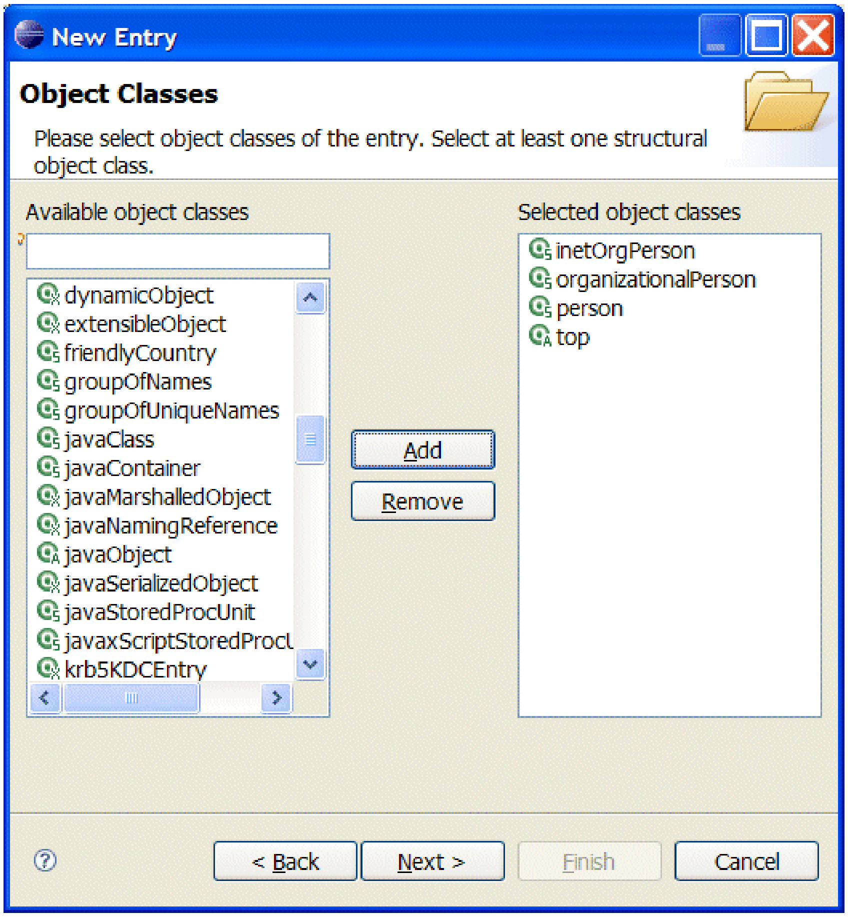


7. Right-click on the `ou=users` node and select **New Entry**. The **New Entry** wizard appears.

8. In the **Entry Creation Method** pane, you do not need to change any settings. Click **Next**.

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

Figure 4.3. New Entry Wizard


New Entry

Object Classes

Please select object classes of the entry. Select at least one structural object class.

Available object classes

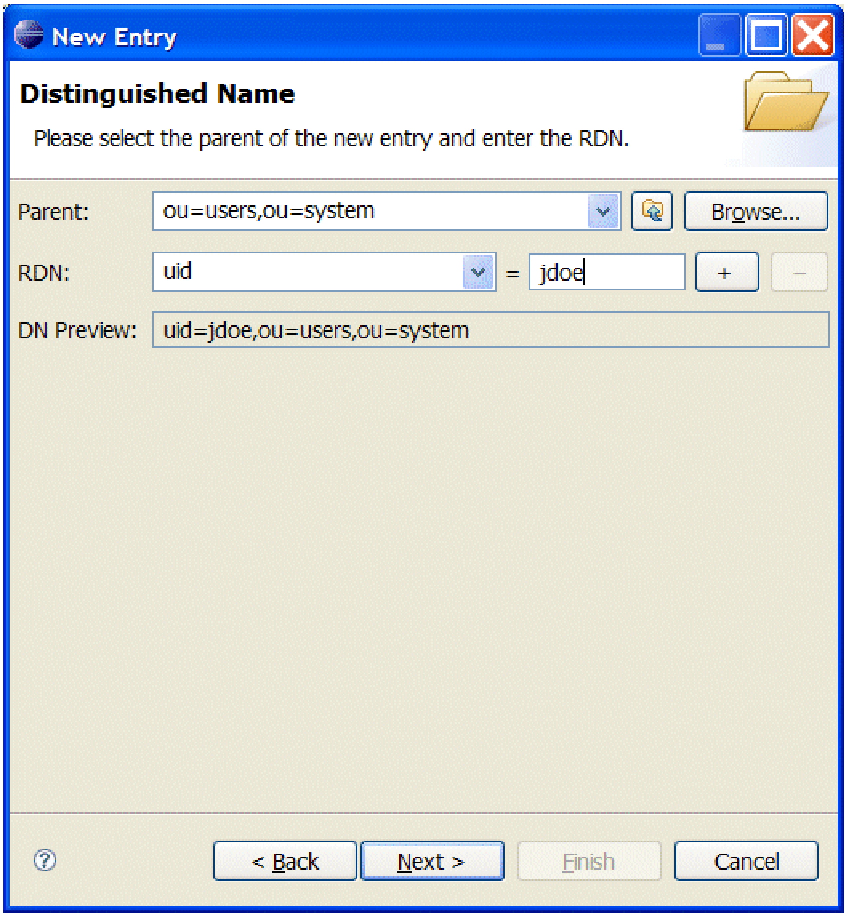
- dynamicObject
- extensibleObject
- friendlyCountry
- groupOfNames
- groupOfUniqueNames
- javaClass
- javaContainer
- javaMarshalledObject
- javaNamingReference
- javaObject
- javaSerializedObject
- javaStoredProcUnit
- javaxScriptStoredProcU
- krb5KDCEntry

Selected object classes

- inetOrgPerson
- organizationalPerson
- person
- top

10. In the **Distinguished Name** pane, complete the **RDN** field, putting `uid` in front and `jdoe` after the equals sign. Click **Next**.

Figure 4.4. Distinguished Name Step of New Entry Wizard



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

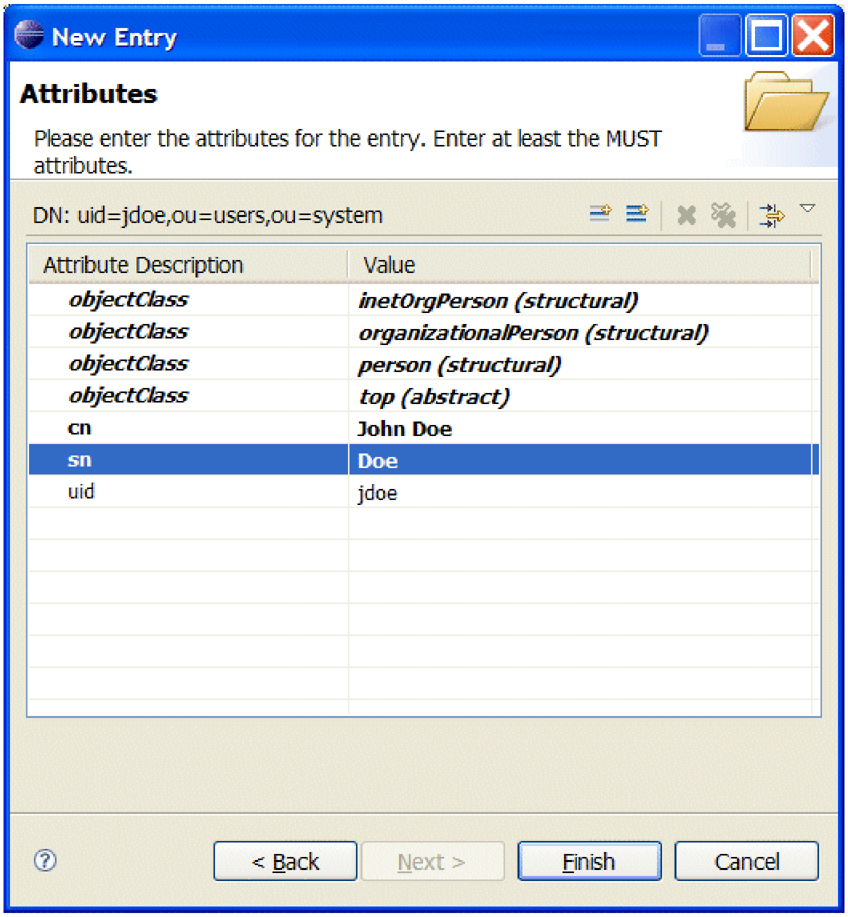
Parent: ou=users,ou=system

RDN: uid = jdoe

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

11. Now fill in the remaining mandatory attributes in the **Attributes** pane. Set the **cn** (common name) attribute to `John Doe` and the **sn** (surname) attribute to `Doe`. Click **Finish**.

Figure 4.5. Attributes Step of New Entry Wizard



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

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

Attribute Description	Value
objectClass	*inetOrgPerson (structural)*
objectClass	*organizationalPerson (structural)*
objectClass	*person (structural)*
objectClass	*top (abstract)*
cn	**John Doe**
sn	**Doe**
uid	jdoe

12. Add a `userPassword` attribute to the user entry. In the **LDAP Browser** view, you should now be able to see a new node, `uid=jdoe`. Select the `uid=jdoe` node. Now, right-click in the **Entry Editor** view and select **New Attribute**. The **New Attribute** wizard appears.
13. From the **Attribute type** drop-down list, select `userPassword`. Click **Finish**.

14. The **Password Editor** dialog appears. In the **Enter New Password** field, enter the password, `secret`. Click **Ok**.

15. To add more users, repeat steps 7 to 14.

Tutorial: Enable LDAP Authentication in the Broker and its Clients

Overview

This section describes how to configure LDAP authentication in the broker, so that it can authenticate incoming credentials based on user entries stored in the X.500 directory server. The tutorial concludes by showing how to program credentials in Java clients and by running an end-to-end demonstration using the consumer and producer tools.

Steps to enable LDAP authentication

Perform the following steps to enable LDAP authentication:

1. Create the login configuration file. Using a text editor, create the file, `login.conf` under the directory, `$ACTIVEMQ_HOME/conf`. Paste the following text into the `login.conf` file:

   ```
   LDAPLogin {
       org.apache.activemq.jaas.LDAPLoginModule required
           debug=true
          initialContextFactory=com.sun.jndi.ldap.LdapCtxFact
   ory
           connectionURL="ldap://localhost:10389"
           connectionUsername="uid=admin,ou=system"
           connectionPassword=secret
           connectionProtocol=""
           authentication=simple
           userBase="ou=users,ou=system"
           userSearchMatching="(uid={0})"
           userSearchSubtree=false
           roleSearchMatching="(uid={1})"
           ;
   };
   ```

   Where these settings assume that the broker connects to a default instance of the Apache Directory Server running on the local host. The account with username, `uid=admin,ou=system`, and password, `secret`, is the default administration account created by the Apache server.

2. Add the LDAP authentication plug-in to the broker configuration. Open the broker configuration file, `$ACTIVEMQ_HOME/conf/activemq.xml`, with a text editor and add the `jaasAuthenticationPlugin` element, as follows:

```
<beans>
  <broker ...>
    ...
    <plugins>
      <jaasAuthenticationPlugin configuration="LDAPLogin"
/>
    </plugins>
    ...
  </broker>
</beans>
```

The value of the configuration attribute, `LDAPLogin`, references the login entry from the `login.conf` file.

3. Comment out the mediation router elements in the broker configuration. Open the broker configuration file and comment out the `camelContext` element as follows:

```
<beans>
  <broker ...>
    ...
  </broker>

  <!--
  <camelContext>
    ...
  </camelContext>
  -->
  ...
</beans>
```

The Camel route is *not* used in the current tutorial. If you left it enabled, you would have to supply it with appropriate username/password credentials, because it acts as a broker client.

4. Add username/password credentials to the consumer tool. Edit the file, `example/src/ConsumerTool.java`, search for the line that creates a new `ActiveMQConnectionFactory` instance, and just before this line, set the credentials, `user` and `password`, as shown:

```
// Java
...
public void run() {
  ...
   user = "jdoe";
   password = "secret";
  ActiveMQConnectionFactory connectionFactory = new ActiveM
QConnectionFactory(user, password, url);
  ...
}
```

5. Add username/password credentials to the producer tool. Edit the file, `example/src/ProducerTool.java`, search for the line that creates a new `ActiveMQConnectionFactory` instance, and just before this line, set the credentials, `user` and `password`, just as you did for the consumer tool.
6. Ensure that the X.500 directory server is running. If necessary, manually restart the X.500 directory server. If the server is not running, all broker connections will fail.
7. Run the broker. Open a new command prompt and start the broker by entering the following command:

   ```
   activemq
   ```

8. Run the consumer client. Open a new command prompt, change directory to `example` and enter the following Ant command:

   ```
   ant consumer -Durl=tcp://localhost:61616 -Dmax=100
   ```

9. Run the producer client. Open a new command prompt, change directory to `example` and enter the following Ant command:

   ```
   ant producer -Durl=tcp://localhost:61616
   ```

10. Perform a negative test. Edit one of the client source files (for example, `ConsumerTool.java`) and change the credentials (username and password) to some invalid values. Now, if you re-run the client, you will get an authentication error.

Appendix A. ASN.1 and Distinguished Names

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.

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.

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 FUSE Services Framework, 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 A.1 on page 104 shows a selection of the attribute types that you are most likely to encounter:

Table A.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
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 A.1 on page 104). 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
```

Appendix B. Licenses

This appendix contains the text of licenses that are relevant to FUSE Services Framework.

OpenSSL License

The licence agreement for using the OpenSSL command line utility shipped with FUSE Services Framework SSL/TLS is as follows:

Index