Fuse ESB

ActiveMQ Security Guide

Version 4.4.1
Sept. 2011

ActiveMQ Security Guide

Version 4.4.1

Updated: 06 Jun 2013

Trademark Disclaimer

Third Party Acknowledgements

One or more products in the Red Hat JBoss Fuse release includes third party components covered by licenses that require that the following documentation notices be provided:

- JLine (http://jline.sourceforge.net) jline:jline:jar:1.0

  License: BSD (LICENSE.txt) - Copyright (c) 2002-2006, Marc Prud'hommeaux `<mwp1@cornell.edu>`

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JLine nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR

SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- Stax2 API (http://woodstox.codehaus.org/StAX2) org.codehaus.woodstox:stax2-api:jar:3.1.1

  License: The BSD License (http://www.opensource.org/licenses/bsd-license.php)

  Copyright (c) <YEAR>, <OWNER> All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- jibx-run - JiBX runtime (http://www.jibx.org/main-reactor/jibx-run) org.jibx:jibx-run:bundle:1.2.3

  License: BSD (http://jibx.sourceforge.net/jibx-license.html) Copyright (c) 2003-2010, Dennis M. Sosnoski.

  All rights reserved.

  Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

  - Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
  - Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
  - Neither the name of JiBX nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON

ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

- JavaAssist (http://www.jboss.org/javassist) org.jboss.javassist:com.springsource.javassist:jar:3.9.0.GA:compile

  License: MPL (http://www.mozilla.org/MPL/MPL-1.1.html)

- HAPI-OSGI-Base Module (http://hl7api.sourceforge.net/hapi-osgi-base/) ca.uhn.hapi:hapi-osgi-base:bundle:1.2

  License: Mozilla Public License 1.1 (http://www.mozilla.org/MPL/MPL-1.1.txt)

Table of Contents

1. SSL/TLS Security 13
- Introduction to SSL/TLS 14
- Secure Transport Protocols 17
- Java Keystores 18
- How to Use X.509 Certificates 20
- Configuring JSSE System Properties 24
- Setting Security Context for the Openwire/SSL Protocol 27
- Securing Java Clients 29
- SSL/TLS Tutorial 31

2. Managing Certificates 37
- What is an X.509 Certificate? 38
- Certification Authorities 40
  - Commercial Certification Authorities 41
  - Private Certification Authorities 42
- Certificate Chaining 43
- Special Requirements on HTTPS Certificates 45
- Creating Your Own Certificates 48

3. Authentication 55
- Programming Client Credentials 56
- Configuring Credentials for Broker Components 57
- Simple Authentication Plug-In 59
- JAAS Authentication 61
  - Introduction to JAAS 62
  - JAAS Username/Password Authentication Plug-In 64
  - JAAS Certificate Authentication Plug-In 66
  - JAAS Dual Authentication Plug-In 70
  - JAAS Guest Login Module 72
  - JAAS LDAP Login Module 75
- Broker-to-Broker Authentication 80
- Web Console Security 81

4. Authorization 85
- Simple Authorization Plug-In 86
- LDAP Authorization Plug-In 89
- Programming Message-Level Authorization 94

5. JMX Security 97
- JMX Connectors 98
- Apache ActiveMQ JMX Connector 99
  - Authentication for the Apache ActiveMQ JMX Connector 100
  - SSL/TLS for the Apache ActiveMQ JMX Connector 106
- Platform JMX Connector 107
  - Authentication for the Platform JMX Connector 108

SSL/TLS for the Platform JMX Connector 114
6. LDAP Tutorial 117
Tutorial Overview 118
Tutorial: Install a Directory Server and Browser 119
Tutorial: Add User Entries and Group Entries 121
Tutorial: Enable LDAP Authentication in the Broker and its Clients 131
Tutorial: Add Authorization Entries 134
Tutorial: Enable LDAP Authorization in the Broker 139
A. ASN.1 and Distinguished Names 141
ASN.1 142
Distinguished Names 143
B. LDAP Entries as an LDIF File 147
Importing from LDIF 148
Index 155

List of Figures

1.1. Target-Only Authentication Scenario 20
1.2. Mutual Authentication Scenario 21
2.1. A Certificate Chain of Depth 2 43
2.2. A Certificate Chain of Depth 3 43
5.1. Alternative JMX Connectors 98
6.1. New LDAP Connection Wizard 122
6.2. Authentication Step of New LDAP Connection 123
6.3. New Entry Wizard 124
6.4. Distinguished Name Step of New Entry Wizard 125
6.5. DIT after Creating ActiveMQ, User, and Group Nodes 126
6.6. Attributes Step of New Entry Wizard 127
6.7. Attributes Step of New Entry Wizard 128
6.8. Attributes Step of New Entry Wizard 129
6.9. Complete Tree of User Entries and Group Entries 130
6.10. DIT after Creating Destination, Queue, and Topic Nodes 135
6.11. DIT after Creating Children of Queue and Topic Nodes 136
6.12. Attributes of the cn=admin Permission Node 137
6.13. DIT after Creating Children of Queue and Topic Nodes 138

List of Tables

1.1. Secure Transport Protocols 17
1.2. JSSE System Properties 24
A.1. Commonly Used Attribute Types 144

List of Examples

1.1. Java Client Using the ActiveMQSslConnectionFactory Class 30
2.1. OpenSSL Configuration 49
2.2. Creating a CA Certificate 49
2.3. Creating a CSR 50
2.4. Converting a Signed Certificate to PEM 51
2.5. Importing a Certificate Chain 51
2.6. Adding a CA to the Trust Store 52
2.7. Creating a Certificate and Private Key using Keytool 52
2.8. Signing a CSR 53
3.1. Simple Authentication Configuration 59
3.2. Enabling Anonymous Access 60
3.3. JAAS Login Configuration File Format 62
3.4. JAAS Login Entry for Simple Authentication 64
3.5. JAAS Login Entry for Certificate Authentication 66
3.6. JAAS Login Entries for Secure and Insecure Connections 70
3.7. Guest Login Accepting No Credentials or Invalid Credentials 72
3.8. Guest Login Accepting No Credentials Only 73
3.9. LDAP Login Entry 75
4.1. Simple Authorization Plug-In Configuration 86
4.2. LDAP Authorization Plug-In Configuration 89
4.3. Implementation of MessageAuthorizationPolicy 94
B.1. LDIF for the LDAP Tutorial 149

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 14
Secure Transport Protocols 17
Java Keystores 18
How to Use X.509 Certificates 20
Configuring JSSE System Properties 24
Setting Security Context for the Openwire/SSL Protocol 27
Securing Java Clients 29
SSL/TLS Tutorial 31

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 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:

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

- 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 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 37.

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

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 17 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

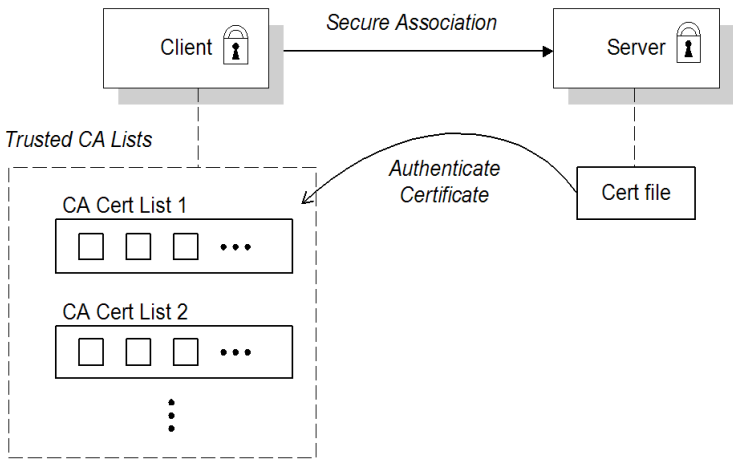
Overview

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 20, 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



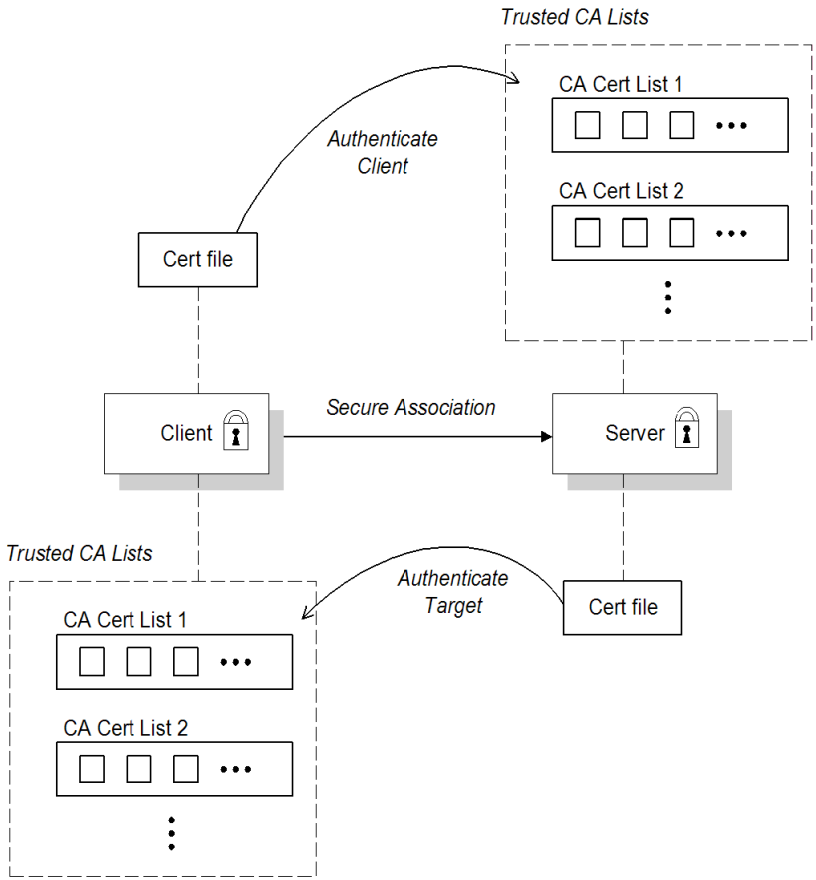
The broker is configured to have its own certificate and private key, which are both stored in the file, `broker.ks`. The client is configured to have a trust store, `client.ts`, that contains the certificate that originally signed the broker certificate. Normally, the trusted certificate is a Certificate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

Mutual authentication

In the mutual authentication scenario, as shown in Figure 1.2 on page 21, 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 37

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 24 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, `\`.

System Property Name	Description
	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): 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-5.5.1-fuse-00-xx/conf/broker.ks
```

```
        -Djavax.net.ssl.keyStorePassword=password
        -Djavax.net.ssl.trustStore=C:/Programs/FUSE/fuse-message-broker-5.5.1-fuse-00-
xx/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-5.5.1-fuse-00-
xx/conf/broker.ks
        -Djavax.net.ssl.keyStorePassword=password
        -Djavax.net.ssl.trustStore=/local/FUSE/fuse-message-broker-5.5.1-fuse-00-
xx/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-5.5.1-fuse-00-xx/conf/client.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:${activemq.base}/conf/broker.ks"
                        keyStorePassword="password"
                        trustStore="file:${activemq.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`—defaults to JKS.
- `trustStore`—equivalent to setting `javax.net.ssl.trustStore`.
- `trustStorePassword`—equivalent to setting `javax.net.ssl.trustStorePassword`.
- `trustStoreType`—equivalent to setting `javax.net.ssl.trustStoreType`.

Securing Java Clients

ActiveMQSslConnectionFactory class

To support SSL/TLS security in Java clients, Apache ActiveMQ provides the `org.apache.activemq.ActiveMQSslConnectionFactory` class. Use the `ActiveMQSslConnectionFactory` class in place of the insecure `ActiveMQConnectionFactory` class in order to enable SSL/TLS security in your clients.

The `ActiveMQConnectionFactory` class exposes the following methods for configuring SSL/TLS security:

`setTrustStore(String)`
: Specifies the location of the client's trust store file, in JKS format (as managed by the Java `keystore` utility).

`setTrustStorePassword(String)`
: Specifies the password that unlocks the client trust store.

`setKeyStore(String)`
: *(Optional)* Specifies the location of the client's own X.509 certificate and private key in a key store file, in JKS format (as managed by the Java `keystore` utility). Clients normally do *not* need to provide their own certificate, unless the broker SSL/TLS configuration specifies that client authentication is required.

`setKeyStorePassword(String)`
: *(Optional)* Specifies the password that unlocks the client key store. This password is also used to decrypt the private key from in the key store.

Note

For more advanced applications, `ActiveMQSslConnectionFactory` also exposes the `setKeyAndTrustManagers` method, which lets you specify the `javax.net.ssl.KeyManager[]` array and the `javax.net.ssl.TrustManager[]` array directly.

Specifying the trust store and key store locations

Location strings passed to the `setTrustStore` and `setKeyStore` methods can have either of the following formats:

- *A pathname*—where no scheme is specified, for example, `/conf/client.ts`. In this case the resource is loaded from the classpath, which is convenient to use when the client and its certificates are packaged in a JAR file.
- *A Java URL*—where you can use any of the standard Java URL schemes, such as `http` or `file`. For example, to reference the file, `C:\ActiveMQ\conf\client.ts`, in the filesystem on a Windows O/S, use the URL, `file:///C:/ActiveMQ/conf/client.ts`.

Sample client code

Example 1.1 on page 30 shows an example of how to initialize a message producer client in Java, where the message producer connects to the broker using the SSL/TLS protocol. The key step here is that the client uses the `ActiveMQSslConnectionFactory` class to create the connection, also setting the trust store and trust store password (no key store is required here, because we are assuming that the broker port does not require client authentication).

Example 1.1. Java Client Using the ActiveMQSslConnectionFactory Class

```
// Java
import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.MessageProducer;
import javax.jms.Session;

import org.apache.activemq.ActiveMQSslConnectionFactory;
...
String url = "ssl://localhost:61617" // The broker URL

// Configure the secure connection factory.
ActiveMQSslConnectionFactory connectionFactory = new ActiveMQSslConnectionFactory(url);
connectionFactory.setTrustStore("/conf/client.ts");
connectionFactory.setTrustStorePassword("password");

// Create the connection.
Connection connection = connectionFactory.createConnection();
connection.start();

// Create the session
Session session = connection.createSession(transacted, Session.AUTO_ACKNOWLEDGE);
Destination destination = session.createQueue(subject);

// Create the producer.
MessageProducer producer = session.createProducer(destination);
```

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 32.
2. "Configure the broker" on page 32.
3. "Configure the consumer and the producer clients" on page 33.
4. "Run the broker" on page 34
5. "Run the consumer with the SSL protocol" on page 34.
6. "Run the producer with the HTTPS protocol" on page 34.
7. "Enable SSL logging in the consumer" on page 35.

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

Warning

The demonstration broker key store and broker trust store are provided for testing purposes only. *Do not deploy these certificates in a production system.* To set up a genuinely secure SSL/TLS system, you must generate custom certificates, as described in "Managing Certificates" on page 37.

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:${activemq.base}/conf/broker.ks"
                        keyStorePassword="password"
                        trustStore="file:${activemq.base}/conf/broker.ts"
                        trustStorePassword="password"/>
        </sslContext>

        <transportConnectors>
             <transportConnector name="ssl" uri="ssl://localhost:61617"/>
            <transportConnector name="https" uri="https://localhost: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" description="Runs a simple consumer">
        ...
          <java classname="ConsumerTool" fork="yes" maxmemory="100M">
               <classpath refid="javac.classpath" />
               <jvmarg value="-server" />
              <sysproperty key="activemq.home" value="${activemq.home}"/>
              <sysproperty key="javax.net.ssl.trustStore"
                           value="${activemq.home}/conf/client.ts"/>
              <sysproperty key="javax.net.ssl.trustStorePassword"
                           value="password"/>
               <arg value="--url=${url}" />
            ...
          </java>
    </target>

     <target name="producer" depends="compile" description="Runs a simple producer">
        ...
          <java classname="ProducerTool" fork="yes" maxmemory="100M">
               <classpath refid="javac.classpath" />
               <jvmarg value="-server" />
              <sysproperty key="activemq.home" value="${activemq.home}"/>
              <sysproperty key="javax.net.ssl.trustStore"
                           value="${activemq.home}/conf/client.ts"/>
              <sysproperty key="javax.net.ssl.trustStorePassword"
                           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 message(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" description="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.SSLSocketFactory
Impl
     [java] class com.sun.net.ssl.internal.ssl.SSLSocketFactoryImpl 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. You can create X.509 certificates that identify your Fuse Message Broker applications.

What is an X.509 Certificate? 38
Certification Authorities 40
    Commercial Certification Authorities 41
    Private Certification Authorities 42
Certificate Chaining 43
Special Requirements on HTTPS Certificates 45
Creating Your Own Certificates 48

What is an X.509 Certificate?

Role of certificates

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

Integrity of the public key

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

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

Digital signatures

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

Warning

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

Contents of an X.509 certificate

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

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

- A *subject distinguished name (DN)* that identifies the certificate owner.
- The *public key* associated with the subject.

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

Distinguished names

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

See Appendix A on page 141 for more details about DNs.

Certification Authorities

Commercial Certification Authorities 41
Private Certification Authorities 42

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

There are two types of CA you can use:

- commercial CAs are companies that sign certificates for many systems.
- private CAs are trusted nodes that you set up and use to sign certificates for your system only.

Commercial Certification Authorities

Signing certificates

There are several commercial CAs available. The mechanism for signing a certificate using a commercial CA depends on which CA you choose.

Advantages of commercial CAs

An advantage of commercial CAs is that they are often trusted by a large number of people. If your applications are designed to be available to systems external to your organization, use a commercial CA to sign your certificates. If your applications are for use within an internal network, a private CA might be appropriate.

Criteria for choosing a CA

Before choosing a commercial CA, consider the following criteria:

- What are the certificate-signing policies of the commercial CAs?
- Are your applications designed to be available on an internal network only?
- What are the potential costs of setting up a private CA compared to the costs of subscribing to a commercial CA?

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>). 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 48 .

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.

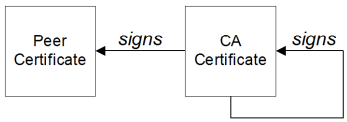
Certificate Chaining

Certificate chain

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

Figure 2.1 on page 43 shows an example of a simple certificate chain.

Figure 2.1. A Certificate Chain of Depth 2



Self-signed certificate

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

Chain of trust

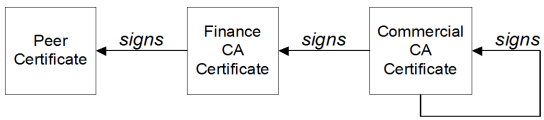
The purpose of a certificate chain is to establish a chain of trust from a peer certificate to a trusted CA certificate. The CA vouches for the identity in the peer certificate by signing it. If the CA is one that you trust (indicated by the presence of a copy of the CA certificate in your root certificate directory), this implies you can trust the signed peer certificate as well.

Certificates signed by multiple CAs

A CA certificate can be signed by another CA. For example, an application certificate could be signed by the CA for the finance department of Progress Software, which in turn is signed by a self-signed commercial CA.

Figure 2.2 on page 43 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.

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 "Generate a certificate and private key pair" on page 52.

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:*.fusesource.com
```

This certificate identity matches any three-component host name in the domain `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 `*fusesource.com`, your certificate could be used on *any* domain that ends in the letters `fusesource`.

Creating Your Own Certificates

Overview

If you choose to use a private CA you will need to generate your own certificates for your applications to use. The OpenSSL project provides free command-line utilities for setting up a private CA, creating signed certificates, and adding the CA to your Java keystore.

OpenSSL utilities

You can download the OpenSSL utilities from http://openssl.org/.

This section describes using the OpenSSL command-line utilities to create certificates. Further documentation of the OpenSSL command-line utilities can be obtained at http://www.openssl.org/docs.

Procedure

To create your own CA and certificates:

1. Add the OpenSSL `bin` directory to your path.
2. Create your own private CA.
   a. Create the directory structure for the CA.

      The directory structure should be:

      - *`X509CA`*`/ca`
      - *`X509CA`*`/certs`
      - *`X509CA`*`/newcerts`
      - *`X509CA`*`/crl`

      Where *`X509CA`* is the name of the CA's home directory.

   b. Copy the `openssl.cnf` file from your OpenSSL installation to your *`X509CA`* directory.
   c. Open your copy of `openssl.cnf` in a text editor.
   d. Edit the `[CA_default]` section to look like Example 2.1 on page 49.

Example 2.1. OpenSSL Configuration

```
#####################################################################
[ CA_default ]

dir        = X509CA            # Where CA files are kept
certs      = $dir/certs  # Where issued certs are kept
crl_dir    = $dir/crl          # Where the issued crl are kept
database   = $dir/index.txt    # Database index file
new_certs_dir = $dir/newcerts  # Default place for new certs

certificate   = $dir/ca/new_ca.pem # The CA certificate
serial        = $dir/serial        # The current serial number
crl           = $dir/crl.pem       # The current CRL
private_key   = $dir/ca/new_ca_pk.pem  # The private key
RANDFILE      = $dir/ca/.rand
# Private random number file

x509_extensions = usr_cert  # The extensions to add to the cert
...
```

Tip

You might decide to edit other details of the OpenSSL configuration at this point. For more details, see the OpenSSL documentation[1].

e. Initialize the CA database as described in "CA database files" on page 52.

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

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

You are prompted for a pass phrase for the CA private key and details of the CA distinguished name as shown in Example 2.2 on page 49.

Example 2.2. Creating a CA Certificate

```
Using configuration from X509CA/openssl.cnf
Generating a 512 bit RSA private key
....+++++
.+++++
writing new private key to 'new_ca_pk.pem'
```

[1] http://www.openssl.org/docs

```
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) []:FuseSource
Organizational Unit Name (eg, section) []:Finance
Common Name (eg, YOUR name) []:Gordon Brown
Email Address []:gbrown@fusesource.com
```

Note

The security of the CA depends on the security of the private key file and the private key pass phrase used in this step.

You must ensure that the file names and location of the CA certificate and private key, `new_ca.pem` and `new_ca_pk.pem`, are the same as the values specified in `openssl.cnf` during Step 2.d on page 48.

3. Create signed certificates in a Java keystore.

   a. Generate a certificate and private key pair using the **keytool -genkeypair** command.

      For details on the options to use when using **keytool -genkeypair** see "Generate a certificate and private key pair" on page 52.

   b. Create a certificate signing request using the **keystore -certreq** command.

      Example 2.3 on page 50 creates a new certificate signing request for the `fusesample.jks` certificate and exports it to the `fusesample_csr.pem` file.

Example 2.3. Creating a CSR

```
keytool -certreq -alias fuse -file fusesample_csr.pem -keypass fusepass -keystore fuses
ample.jks -storepass fusestorepass
```

c. Sign the CSR using the **openssl ca** command.

   You will prompted to enter the CA private key pass phrase you used when creating the CA in Step 2.f on page 49).

   See "Signing a CSR" on page 53 for details on the options to use when signing the CSR.

d. Convert the signed certificate to PEM only format using the **openssl x509** command with the `-outform` option set to `PEM`.

   Example 2.4 on page 51 converts the signed certificate `fusesigned.pem`.

Example 2.4. Converting a Signed Certificate to PEM

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

e. Concatenate the CA certificate file and the converted, signed certificate file to form a certificate chain.

   The CA certificate file is stored in the CA's `ca` directory. For example, the certificate file for the CA created in Step 2.f on page 49 would be `ca/new_ca.pem`.

f. Import the new certificate's full certificate chain into the Java keystore using the **keytool -import** command.

   Example 2.5 on page 51 imports the chain `fusesample.chain` into the `fusesample.jks` keystore.

Example 2.5. Importing a Certificate Chain

```
keytool -import -file fusesample.chain -keypass fusepass -keystore fusesample.jks -storepass
 fusestorepass
```

4. Repeat Step 3 on page 50 to create a full set of certificates for your system.

5. Add trusted CAs to your Java trust store.

   a. Assemble the collection of trusted CA certificates that you want to deploy.

      The trusted CA certificates can be obtained from public CAs or private CAs. The trusted CA certificates can be in any format that is compatible with the Java **keystore** utility; for example, PEM format. All you need are the certificates themselves—the private keys and passwords are *not* required.

   b. Add a CA certificate to the trust store using the **keytool -import** command.

      Example 2.6 on page 52 adds the CA certificate `cacert.pem`, in PEM format, to a JKS trust store.

Example 2.6. Adding a CA to the Trust Store

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

`truststore.ts` is a keystore file containing CA certificates. If this file does not already exist, the **keytool** command creates one. `StorePass` is the password required to access the keystore file.

c. Repeat Step 5.b on page 51 to add all of the CA certificates to the trust store.

CA database files

The CA uses two files, `serial` and `index.txt` to maintain its database of certificate files. Both files must be stored in the *`X509CA`* directory.

When you first create your CA the OpenSSL tools require that they have very specific initial contents:

- `serial`

  The initial contents of this file must be `01`.

- `index.txt`

  Initially this file *must* be completely empty. It cannot even contain white space.

Generate a certificate and private key pair

To generate a certificate and private key pair you use the **keytool -genkeypair** command. For example, Example 2.7 on page 52 creates a certificate and key pair that are valid for 365 days and is stored in the keystore file `fusesample.jks`. The generated key store entry will use the alias `fuse` and the password `fusepass`.

Example 2.7. Creating a Certificate and Private Key using Keytool

```
keytool -genkeypair -dname "CN=Alice, OU=Engineering, O=Progress, ST=Co. Dublin, C=IE" -
validity 365 -alias fuse -keypass fusepass -keystore fusesample.jks -storepass fusestorepass
```

Because the specified keystore, `fusessample.jks`, did not exist prior to issuing the command implicitly creates a new keystore and sets its password to `fusestorepass`.

The `-dname` and `-validity` flags define the contents of the newly created X.509 certificate.

The `-dname` flag specifies the subject DN. For more details about DN format, see Appendix A on page 141. Some parts of the subject DN must match the values in the CA certificate (specified in the CA Policy section of the `openssl.cnf` file). The default `openssl.cnf` file requires the following entries to match:

- Country Name (C)
- State or Province Name (ST)
- Organization Name (O)

Note

If you do not observe the constraints, the OpenSSL CA will refuse to sign the certificate (see Step 2.f on page 49).

The `-validity` flag specifies the number of days for which the certificate is valid.

Signing a CSR

To sign a CSR using your CA, you use the **openssl ca** command. At a minimum you will need to specify the following options:

- `-config`—the path to the CA's `openssl.cnf` file
- `-in`—the path to certificate to be signed
- `-out`—the path to the signed certificates

Example 2.8 on page 53 signs the `fusesample_csr.pem` certificate using the CA stored at `/etc/fuseCA`.

Example 2.8. Signing a CSR

```
openssl ca -config /etc/fuse/openssl.cnf -days 365 -in fusesample_csr.pem -out fusesigned.pem
```

For more details on the **openssl ca** command see http://www.openssl.org/docs/apps/ca.html#.

Chapter 3. Authentication

Fuse Message Broker has a flexible authentication model, which includes support for several different JAAS authentication plug-ins.

Programming Client Credentials 56
Configuring Credentials for Broker Components 57
Simple Authentication Plug-In 59
JAAS Authentication 61
    Introduction to JAAS 62
    JAAS Username/Password Authentication Plug-In 64
    JAAS Certificate Authentication Plug-In 66
    JAAS Dual Authentication Plug-In 70
    JAAS Guest Login Module 72
    JAAS LDAP Login Module 75
Broker-to-Broker Authentication 80
Web Console Security 81

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 ActiveMQConnectionFactory(url);
  Connection connection = connectionFactory.createConnection(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>
```

Apache Camel

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

```
<beans>
  ...
  <camelContext id="camel" xmlns="http://activemq.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.component.ActiveMQComponent" >
    <property name="connectionFactory">
      <bean class="org.apache.activemq.ActiveMQConnectionFactory">
       <property name="brokerURL" value="vm://localhost?create=false&amp;waitForStart=10000"
 />
        <property name="userName" value="${activemq.username}"/>
        <property name="password" value="${activemq.password}"/>
      </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.

Anonymous access

By default, if a client does not provide any JMS username/password credentials, access to the broker is denied. It is possible, however, to enable anonymous access by setting the `anonymousAccessAllowed` attribute to `true` on the `simpleAuthenticationPlugin` element, as shown in Example 3.2 on page 60.

Example 3.2. Enabling Anonymous Access

```
<simpleAuthenticationPlugin anonymousAccessAllowed="true">
  <users>
    <authenticationUser username="system"
                        password="manager"
                        groups="users,admins"/>
    ...
  </users>
</simpleAuthenticationPlugin>
```

Now, with anonymous access enabled, when a client without credentials connects to the broker, it is automatically assigned the username, `anonymous`, and the group ID, `anonymous`. When used in combination with the authorization plug-in, you can assign strictly limited privileges to the `anonymous` group in order to protect your system.

You can optionally change the username and group ID that gets assigned to anonymous users by setting the `anonymousUser` and `anonymousGroup` attributes—for example:

```
<simpleAuthenticationPlugin
    anonymousAccessAllowed="true"
    anonymousUser="JohnDoe"
    anonymousGroup="unauthenticated">
              <users>
    ...
              </users>
        </simpleAuthenticationPlugin>
```

Note

If you enable anonymous access, it is highly recommended that you also enable authorization, otherwise your broker would be completely exposed to all users.

JAAS Authentication

Introduction to JAAS 62
JAAS Username/Password Authentication Plug-In 64
JAAS Certificate Authentication Plug-In 66
JAAS Dual Authentication Plug-In 70
JAAS Guest Login Module 72
JAAS LDAP Login Module 75

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 3.3 on page 62.

Example 3.3. 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 class name of Fuse Message Broker's JAAS simple authentication login module.

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

- *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/config/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 Username/Password Authentication Plug-In

Overview

The JAAS username/password authentication plug-in performs login based on the JMS username/password credentials received from a client. This plug-in can be used with any JAAS login module that stores username/password credentials—for example, the properties login module or the LDAP login module.

Properties login module

The JAAS properties login module provides a simple store of authentication data, where the relevant user 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 Login Module" on page 75).

The properties login module is implemented by the following class:

```
org.apache.activemq.jaas.PropertiesLoginModule
```

Defining the JAAS realm

You can define a JAAS realm by creating a corresponding login entry in a `login.config` file. The following `PropertiesLogin` login entry shows how to configure the properties login module in the `login.config` file:

Example 3.4. JAAS Login Entry for Simple Authentication

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

In the preceding example, the JAAS realm is configured to use a single `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 properties login module, 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
```

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. For more details, see "Location of the login configuration file" on page 63.

Enable the JAAS username/password authentication plug-in

To enable the JAAS username/password 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 3.4 on page 64). 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).

Certificate login module

The JAAS certificate login module stores a collection of certificate DNs in a pair of flat files. The files associate a username and a list of group IDs with each DN.

The certificate login module is implemented by the following class:

```
org.apache.activemq.jaas.TextFileCertificateLoginModule
```

Defining the JAAS realm

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

Example 3.5. JAAS Login Entry for Certificate Authentication

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

In the preceding example, the JAAS realm is configured to use a single `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 login module, 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, `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:

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

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
```

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. For more details, see "Location of the login configuration file" on page 63.

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 configuration="CertLogin" />
    </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 66). In the preceding example, the `CertLogin` login entry is selected.

JAAS Dual Authentication Plug-In

Overview

The JAAS dual authentication plug-in behaves effectively like a hybrid of the username/password authentication plug-in and the certificate authentication plug-in. It enables you to specify one JAAS realm to use when a client connection uses SSL, and another JAAS realm to use when the client connection is non-SSL.

For example, this makes it possible to use certificate authentication for SSL connections and JMS username/password authentication for non-SSL connections, where the selection is made dynamically at run time.

Sample JAAS realms

Example 3.6 on page 70 shows the definitions of two sample JAAS realms: a realm for non-SSL connections, `activemq-domain`; and a realm for SSL connections, `activemq-ssl-domain`.

Example 3.6. JAAS Login Entries for Secure and Insecure Connections

```
activemq-domain {
  org.apache.activemq.jaas.PropertiesLoginModule sufficient
      debug=true
      org.apache.activemq.jaas.properties.user="users.properties"
      org.apache.activemq.jaas.properties.group="groups.properties";
  org.apache.activemq.jaas.GuestLoginModule sufficient
      debug=true
      org.apache.activemq.jaas.guest.user="guest"
      org.apache.activemq.jaas.guest.group="guests";
};

activemq-ssl-domain {
  org.apache.activemq.jaas.TextFileCertificateLoginModule required
      debug=true
      org.apache.activemq.jaas.textfiledn.user="dns.properties"
      org.apache.activemq.jaas.textfiledn.group="groups.properties";
};
```

The `activemq-domain` login entry illustrates how to use multiple login modules in a single realm. With this configuration, JAAS tries first of all to authenticate a client using the `PropertiesLoginModule` login module. If that authentication step fails, JAAS then attempts to authenticate the client using the next login module, `GuestLoginModule`. The guest login module assigns a default username and group ID to the client and it always succeeds at authenticating—for more details, see "JAAS Guest Login Module" on page 72.

Enabling the JAAS dual authentication plug-in

To enable the JAAS dual authentication plug-in, add the `jaasDualAuthenticationPlugin` element to the list of plug-ins in the broker configuration file and initialize both the `configuration` attribute (to specify the JAAS realm used for non-SSL connections) and the `sslConfiguration` attribute (to specify the JAAS realm used for SSL connections).

```
<beans>
  <broker ...>
    ...
    <plugins>
      <jaasDualAuthenticationPlugin
              configuration="activemq-domain"
              sslConfiguration="activemq-ssl-domain" />
    </plugins>
    ...
  </broker>
</beans>
```

JAAS Guest Login Module

Overview

The JAAS guest login module allows users without credentials (and, depending on how it is configured, possibly also users with invalid credentials) to access the broker. Normally, the guest login module is chained with another login module, such as a properties login module.

The guest login module responds to successful login requests with a principal that has a fixed username and a fixed group ID.

Guest login use cases

There are two basic use cases for the guest login module, as follows:

- "Guests with no credentials or invalid credentials" on page 72.
- "Guests with no credentials only" on page 73.

Guests with no credentials or invalid credentials

Example 3.7 on page 72 shows how to configure a JAAS login entry for the use case where users with *no credentials or invalid credentials* are logged in as guests. In this example, the guest login module is used in combination with the properties login module.

Example 3.7. Guest Login Accepting No Credentials or Invalid Credentials

```
activemq-domain {
  org.apache.activemq.jaas.PropertiesLoginModule sufficient
      debug=true
      org.apache.activemq.jaas.properties.user="users.properties"
      org.apache.activemq.jaas.properties.group="groups.properties";

  org.apache.activemq.jaas.GuestLoginModule sufficient
      debug=true
      org.apache.activemq.jaas.guest.user="anyone"
      org.apache.activemq.jaas.guest.group="restricted";
};
```

Depending on the user login data, authentication proceeds as follows:

- *User logs in with a valid password*—the properties login module sucessfully authenticates the user and returns immediately. The guest login module is not invoked.

- *User logs in with an invalid password*—the properties login module fails to authenticate the user, and authentication proceeds to the guest login module. The guest login module successfully authenticates the user and returns the guest principal.
- *User logs in with a blank password*—the properties login module fails to authenticate the user, and authentication proceeds to the guest login module. The guest login module successfully authenticates the user and returns the guest principal.

Guests with no credentials only

Example 3.8 on page 73 shows how to configure a JAAS login entry for the use case where only those users with *no credentials* are logged in as guests. To support this use case, you must set the `credentialsInvalidate` option to `true` in the configuration of the guest login module. You should also note that, compared with the preceding example, the order of the login modules is reversed and the flag attached to the properties login module is changed to `requisite`.

Example 3.8. Guest Login Accepting No Credentials Only

```
activemq-guest-when-no-creds-only-domain {
    org.apache.activemq.jaas.GuestLoginModule sufficient
       debug=true
       credentialsInvalidate=true
       org.apache.activemq.jaas.guest.user="guest"
       org.apache.activemq.jaas.guest.group="guests";

    org.apache.activemq.jaas.PropertiesLoginModule requisite
        debug=true
        org.apache.activemq.jaas.properties.user="org/apache/activemq/security/users.prop
erties"
      org.apache.activemq.jaas.properties.group="org/apache/activemq/security/groups.prop
erties";
};
```

Depending on the user login data, authentication proceeds as follows:

- *User logs in with a valid password*—the guest login module fails to authenticate the user (because the user has presented a password while the `credentialsInvalidate` option is enabled) and authentication proceeds to the properties login module. The properties login module sucessfully authenticates the user and returns.
- *User logs in with an invalid password*—the guest login module fails to authenticate the user and authentication proceeds to the properties login module. The properties login module also fails to authenticate the user. *The nett result is authentication failure.*
- *User logs in with a blank password*—the guest login module sucessfully authenticates the user and returns immediately. The properties login module is not invoked.

Guest login entry options

The guest login module supports the following options:

`debug`
(Optional) Boolean debugging flag. If `true`, enable debugging. This is used only for testing or debugging. Normally, it should be set to `false`, or omitted.

`credentialsInvalidate`
(Optional) Boolean flag. If `true`, reject login requests that include a password. In other words, with this option enabled, guest login succeeds only when the user does not provide a password. Default is `false`.

`org.apache.activemq.jaas.guest.user`
(Optional) Specifies the username assigned to guest users. Default is `guest`.

`org.apache.activemq.jaas.guest.group`
(Optional) Specifies the group ID assigned to guest users. Default is `guests`.

Enabling authentication with the guest login module

You can use the guest login module by combining either with the username/password authentication plug-in or with the dual authentication plug-in. For example, see "JAAS Dual Authentication Plug-In" on page 70.

JAAS LDAP Login Module

Overview

The LDAP login module 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.

Defining the JAAS realm

Example 3.9 on page 75 shows an example of a login entry for the LDAP login module, connecting to a directory server with the URL, `ldap://localhost:10389`.

Example 3.9. 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=User,ou=ActiveMQ,ou=system"
    userSearchMatching="(uid={0})"
    userSearchSubtree=false
    roleBase="ou=Group,ou=ActiveMQ,ou=system"
    roleName=cn
    roleSearchMatching="(member=uid={1})"
    roleSearchSubtree=false
    ;
};
```

The preceding login entry, `LDAPLogin`, is configured to search for users under the `ou=User,ou=ActiveMQ,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=User,ou=ActiveMQ,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`.
- `authentication`—*(mandatory)*specifies the authentication method used when binding to the LDAP server. Can take either of the values, `simple` (username and password) or `none` (anonymous).

Note

Simple Authentication and Security Layer (SASL) authentication is currently *not* supported.

- `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.

- `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=User,ou=ActiveMQ,ou=system`, the search for user entries is restricted to the subtree beneath the `ou=User,ou=ActiveMQ,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 Oracle'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=User,ou=ActiveMQ,ou=system`, it would match the entry, `uid=jdoe,ou=User,ou=ActiveMQ,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`).
  - `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/group entries. The subtree is specified by a DN, which specifes the base node of the subtree. For example, by setting this option to `ou=Group,ou=ActiveMQ,ou=system`, the search for role/group entries is restricted to the subtree beneath the `ou=Group,ou=ActiveMQ,ou=system` node.
- `roleName`—*(optional)*specifies the attribute type on page 144 of the role entry that contains the name of the role/group. If you omit this option, the role search feature is effectively disabled.
- `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=User,ou=ActiveMQ,ou=system`.
  - `{1}` substitutes the received username. For example, `jdoe`.

[3] http://www.ietf.org/rfc/rfc2254.txt
[4] http://download.oracle.com/javase/jndi/tutorial/basics/directory/filter.html

For example, if this option is set to `(member=uid={1})` and the received username is `jdoe`, the search filter becomes `(member=uid=jdoe)` after string substitution (assuming ApacheDS search filter syntax). If the resulting search filter is applied to the subtree selected by the role base, `ou=Group,ou=ActiveMQ,ou=system`, it matches all role entries that have a `member` attribute equal to `uid=jdoe` (the value of a `member` attribute is a DN).

Note

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

Tip

If you use OpenLDAP, the syntax of the search filter is `(member:=uid=jdoe)`.

- `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`).

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.

Storing roles in role entries

If you want to store role data in dedicated role entries (where each node represents a particular role), create a role entry as follows. Create a new child of the `roleBase` node, where the `objectClass` of the child is `groupOfNames`. Set the `cn` (or whatever attribute type is specified by `roleName`) of the new child node equal to the name of the role/group. Define a `member` attribute for each member of the role/group, setting the member value to the DN of the corresponding user (where the DN is specified either fully, `uid=jdoe,ou=User,ou=ActiveMQ,ou=system`, or partially, `uid=jdoe`).

Storing roles in user entries

If you want to add roles to user entries, you would 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.

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. For more details, see "Location of the login configuration file" on page 63.

Enable the JAAS username/password authentication plug-in

To enable the JAAS username/password 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 3.9 on page 75). 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="static://(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.

Web Console Security

Web console for Apache ActiveMQ

The Apache ActiveMQ Web console is a web-based administration tool for Apache ActiveMQ. When you start a standalone broker instance using the script, `bin/activemq[.bat]`, with the default configuration, `conf/activemq.xml`, the Web console is automatically enabled. After starting the broker, you can access the Web console by entering the following URL in your Web browser:

```
http://localhost:8161/admin
```

The Web console is hosted inside a Jetty[5] server, which is configured by the Spring file, `conf/jetty.xml`. The `conf/activemq.xml` configuration file imports the `jetty.xml` file, using the following `<import/>` tag:

```
<beans ... >
    ...
    <import resource="jetty.xml"/>
</beans>
```

Enabling basic authentication

The Jetty server can be configured to enable HTTP basic authentication. Although the `conf/jetty.xml` file already includes most of the configuration required for basic authentication, the authentication feature is disabled by default. To enable it, search for the following line in the `jetty.xml` file:

```
        <property name="authenticate" value="false" />
```

Edit the `value` attribute, changing its value to `true`.

```
        <property name="authenticate" value="true" />
```

When you restart the broker, basic authentication will be enabled on the Web console. For example, you can log on using the credentials, `username=admin`, `password=admin`.

Editing user credentials

The Jetty user data are stored in the `conf/jetty-realm.properties` file, which you can edit to add user credentials and roles. Each user is defined on a separate line, which has the following format:

```
Username: Password [, Role01, Role02, ... ]
```

For example, to define the user with username, `jblogs`, password, `secret`, and role, `developer`, you would add the following line to the `jetty-realm.properties` file:

[5] http://jetty.codehaus.org/jetty/

```
jblogs: secret, developer
```

Enabling SSL security

To enable SSL security on the Jetty server, edit the `Connector` bean in the `conf/jetty.xml` file. Instead of the `SelectChannelConnector` class, define the `Connector` bean to be an instance of the `org.eclipse.jetty.server.ssl.SslSelectChannelConnector` class. Specify the relevant properties of the `SslSelectChannelConnector` class in order to configure the Jetty server's HTTPS port.

Sample SSL configuration

Search for the existing definition of the `Connector` bean in the `conf/jetty.xml` file. In the default file, you should see some lines like the following:

```
<property name="connectors">
    <list>
        <bean id="Connector" class="org.eclipse.jetty.server.nio.SelectChannelConnector">
            <property name="port" value="8161" />
        </bean>
    </list>
</property>
```

Replace the preceding lines by the following lines:

```
<property name="connectors">
    <list>
       <bean id="Connector" class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">

            <property name="port"        value="8443" />
            <property name="maxIdleTime" value="30000"/>
            <property name="keystore"    value="${activemq.home}/conf/broker.ks"/>
            <property name="password"    value="testjetty"/>
            <property name="keyPassword" value="testjetty"/>
            <property name="truststore"  value="${activemq.home}/conf/broker.ks"/>
        </bean>
    </list>
</property>
```

Where the `SslSelectChannelConnector` properties can be explained as follows:

`port`
: Specifies the secure IP port number (accessible through HTTPS).

`maxIdleTime`
: Specifies the connection idle time in units of milliseconds. If there is no activity on a connection for longer than this timeout, the connection will be closed.

`keystore`
: Specifies the location of the Jetty server's own X.509 certificate, which is stored in a Java keystore file on the file system. The Jetty server uses this certificate to identify itself to a client, during the SSL handshake.

`password`
: Specifies the *store password*, which is needed to unlock the keystore file (see "Java Keystores" on page 18).

`keyPassword`
: Specifies the *key password*, which is used to decrypt the private key that is stored within the keystore file. Typically, the store password and the key password are identical (some SSL implementations even require this to be the case).

`truststore`
: Specifies the location of a Java keystore file that contains a list of one or more trusted certificates, which can be used during the SSL handshake to check that incoming client certificates are correctly signed. In the current example, the truststore is actually irrelevant, because clients are *not* required to send a certificate to the Jetty server.

When SSL security is configured as shown, you can access the Web console through the HTTPS protocol using the following URL:

```
https://localhost:8443/admin
```

Warning

The `broker.ks` certificate used in the preceding example is insecure, because anyone can access its private key. To secure your system properly, you must create new certificates signed by a trusted CA, as described in "Managing Certificates" on page 37.

Reference

For more details about the properties you can set on the `SslSelectChannelConnector` class, see http://download.eclipse.org/jetty/stable-7/apidocs/org/eclipse/jetty/server/ssl/SslSocketConnector.html.

Chapter 4. Authorization

Apache ActiveMQ authorization implements group-based access control and allows you to control access at the granularity level of destinations or of individual messages. Two plug-in implementations are provided: a simple authorization plug-in and an LDAP authorization plug-in.

Simple Authorization Plug-In 86
LDAP Authorization Plug-In 89
Programming Message-Level Authorization 94

Simple 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 simple authorization plug-in, on the other hand, you can restrict access to specific destinations based on a user's group membership.

Configuring the simple authorization plug-in

To configure the simple authorization plug-in, add an `authorizationPlugin` element to the list of plug-ins in the broker configuration, as shown in Example 4.1 on page 86.

Example 4.1. Simple 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>
```

Entry types

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

- "Authorization entries for named destinations" on page 87.
- "Authorization entries for temporary destinations" on page 88.

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.

LDAP Authorization Plug-In

Overview

Using the LDAP authorization plug-in, you can configure a broker to retrieve its authorization data from an X.500 directory server.

Configuring the LDAP authorization plug-in

To configure the LDAP authorization plug-in, add the `authorizationPlugin` element to the list of plug-ins in the broker configuration, as shown in Example 4.2 on page 89.

Example 4.2. LDAP Authorization Plug-In Configuration

```
<beans ...>
  <broker ...>
    ...
    <plugins>
      ...
      <authorizationPlugin>
        <map>
          <bean id="lDAPAuthorizationMap" class="org.apache.activemq.security.LDAPAuthoriz
ationMap"
                xmlns="http://www.springframework.org/schema/beans">
           <property name="initialContextFactory" value="com.sun.jndi.ldap.LdapCtxFactory"/>

             <property name="connectionURL" value="ldap://localhost:10389"/>
             <property name="authentication" value="simple"/>
             <property name="connectionUsername" value="uid=admin,ou=system"/>
             <property name="connectionPassword" value="secret"/>
             <property name="connectionProtocol" value=""/>
             <property name="topicSearchMatchingFormat"
                       value="cn={0},ou=Topic,ou=Destination,ou=ActiveMQ,ou=system"/>
             <property name="topicSearchSubtreeBool" value="true"/>
             <property name="queueSearchMatchingFormat"
                       value="cn={0},ou=Queue,ou=Destination,ou=ActiveMQ,ou=system"/>
             <property name="queueSearchSubtreeBool" value="true"/>
             <property name="advisorySearchBase"
                      value="cn=ActiveMQ.Advisory,ou=Topic,ou=Destination,ou=ActiveMQ,ou=sys
tem"/>
             <property name="tempSearchBase"
                     value="cn=ActiveMQ.Temp,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system"/>

             <property name="adminBase" value="(cn=admin)"/>
             <property name="adminAttribute" value="member"/>
             <property name="readBase" value="(cn=read)"/>
             <property name="readAttribute" value="member"/>
```

```
            <property name="writeBase" value="(cn=write)"/>
            <property name="writeAttribute" value="member"/>
          </bean>
        </map>
      </authorizationPlugin>
    </plugins>
    ...
  </broker>
</beans>
```

LDAP authorization plug-in properties

The LDAP authorization plug-in supports the following properties:

- `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`.
- `authentication`—*(mandatory)*specifies the authentication method used when binding to the LDAP server. Can take either of the values, `simple` (username and password) or `none` (anonymous).

> **Note**
>
> Simple Authentication and Security Layer (SASL) authentication is currently *not* supported.

- `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.

- `topicSearchMatchingFormat`—*(optional)*specifies the DN of the node whose children provide the permissions for the current topic. 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 name of the current topic.

  For example, if this property is set to `cn={0},ou=Topic,ou=Destination,ou=ActiveMQ,ou=system` and the current topic is `TEST.FOO`, the DN becomes `cn=TEST.FOO,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system`.

- `topicSearchSubtreeBool`—*(optional)* specify the search depth for permission entries (admin, read or write entries), relative to the node specified by `topicSearchMatchingFormat`. This option can take boolean values, as follows:

  - `false`—*(default)* try to match one of the child entries of the `topicSearchMatchingFormat` node (maps to `javax.naming.directory.SearchControls.ONELEVEL_SCOPE`).

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

- `queueSearchMatchingFormat`—*(optional)* specifies the DN of the node whose children provide the permissions for the current queue. The special string, `{0}`, is substituted by the name of the current queue.

  For example, if this property is set to `cn={0},ou=Queue,ou=Destination,ou=ActiveMQ,ou=system` and the current queue is `TEST.FOO`, the DN becomes `cn=TEST.FOO,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system`.

- `queueSearchSubtreeBool`—*(optional)* specify the search depth for permission entries (admin, read or write entries), relative to the node specified by `queueSearchMatchingFormat`. This option can take boolean values, as follows:

  - `false`—*(default)* try to match one of the child entries of the `queueSearchMatchingFormat` node (maps to `javax.naming.directory.SearchControls.ONELEVEL_SCOPE`).

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

- `advisorySearchBase`—*(optional)* specifies the DN of the node whose children provide the permissions for *all* advisory topics. In this case the DN is a literal value (that is, no string substitution is performed on the property value).

  For example, a typical value of this property is `cn=ActiveMQ.Advisory,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system`.

- `tempSearchBase`—*(optional)* specifies the DN of the node whose children provide the permissions for *all* temporary queues and topics (apart from advisory topics). In this case the DN is a literal value (that is, no string substitution is performed on the property value).

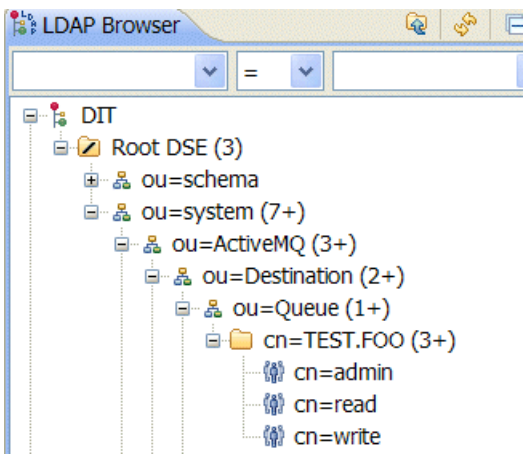
  For example, a typical value of this property is `cn=ActiveMQ.Temp,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system`.

- `adminBase`—*(optional)* specifies an LDAP search filter, which is used when looking up the *admin permissions* for any kind of queue or topic. The search filter attempts to match one of the children (or descendants, if `SUBTREE_SCOPE` is enabled) of the queue or topic node.

  For example, if this property is set to `(cn=admin)`, it will match any child whose `cn` attribute is set to `admin`.

- `adminAttribute`—*(optional)*specifies an attribute of the node matched by `adminBase`, whose value is the DN of a role/group that has admin permissions.

  For example, consider a `cn=admin` node that is a child of the node, `cn=TEST.FOO,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system`, as shown:



The `cn=admin` node might typically have some attributes, as follows:

Attribute Description	Value
objectClass	*groupOfNames (structural)*
objectClass	*top (abstract)*
cn	admin
member	cn=admins
member	cn=users

If you now set the `adminAttribute` property to `member`, the authorization plug-in grants admin privileges over the `TEST.FOO` queue to the `cn=admins` group and the `cn=users` group.

- `readBase`—*(optional)* specifies an LDAP search filter, which is used when looking up the *read permissions* for any kind of queue or topic. The search filter attempts to match one of the children (or descendants, if `SUBTREE_SCOPE` is enabled) of the queue or topic node.

  For example, if this property is set to `(cn=read)`, it will match any child whose `cn` attribute is set to `read`.

- `readAttribute`—*(optional)*specifies an attribute of the node matched by `readBase`, whose value is the DN of a role/group that has read permissions.

- `writeBase`—*(optional)* specifies an LDAP search filter, which is used when looking up the *write permissions* for any kind of queue or topic. The search filter attempts to match one of the children (or descendants, if `SUBTREE_SCOPE` is enabled) of the queue or topic node.

  For example, if this property is set to `(cn=write)`, it will match any child whose `cn` attribute is set to `write`.

- `writeAttribute`—*(optional)*specifies an attribute of the node matched by `writeBase`, whose value is the DN of a role/group that has write permissions.

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 4.3 on page 94 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 4.3. Implementation of MessageAuthorizationPolicy

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

public class MsgAuthzPolicy implements MessageAuthorizationPolicy {

  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 4.3 on page 94, 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 5. JMX Security

INSERT ABSTRACT HERE...

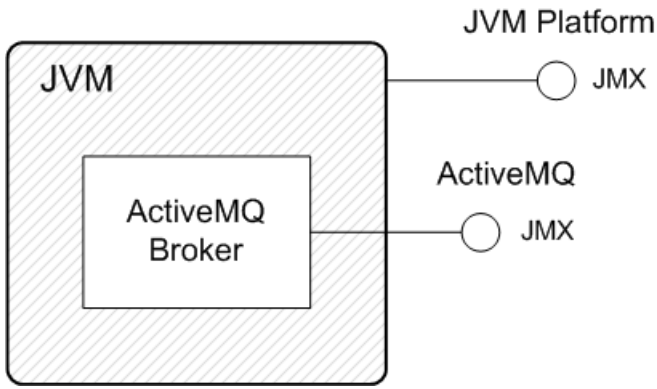
JMX Connectors 98
Apache ActiveMQ JMX Connector 99
Authentication for the Apache ActiveMQ JMX Connector 100
SSL/TLS for the Apache ActiveMQ JMX Connector 106
Platform JMX Connector 107
Authentication for the Platform JMX Connector 108
SSL/TLS for the Platform JMX Connector 114

JMX Connectors

Overview

You can connect to one of two alternative JMX connectors (IP ports), in order to manage the MBeans in an Apache ActiveMQ broker instance. As shown in Figure 5.1 on page 98, you can connect either to the JMX connector exposed by the Java Virtual Machine (JVM) or to the JMX connector exposed directly by the broker.

Figure 5.1. Alternative JMX Connectors



Apache ActiveMQ JMX connector

The Apache ActiveMQ JMX connector is a dedicated IP port that enables you to monitor the MBeans inside the broker instance (and only those MBeans). It is enabled by default.

Platform JMX connector

The JVM platform JMX connector is an IP port that enables you to monitor all of the MBeans that are active in the JVM instance, including the MBeans in the broker. This JMX connector is not used by default, but is automatically activated, if you disable the Apache ActiveMQ JMX connector.

Apache ActiveMQ JMX Connector

Authentication for the Apache ActiveMQ JMX Connector 100
SSL/TLS for the Apache ActiveMQ JMX Connector 106

Authentication for the Apache ActiveMQ JMX Connector

Authentication alternatives

The following alternative authentication mechanisms are supported for the Apache ActiveMQ JMX connector:

- *Simple authentication*—configure authentication by providing two files: a `jmx.password` file, which contains login details, and a `jmx.access` file, which defines access rights for the defined users.
- *JAAS authentication*—configure authentication using JAAS realms and login modules.

Enable simple authentication

To enable the simple authentication mechanism for the Apache ActiveMQ JMX connector, perform the following steps:

1. If not already present in the *`ACTIVEMQ_BASE`*`/conf/` directory, create the password file, *`ACTIVEMQ_BASE`*`/conf/jmx.password`, and add the following lines using your favorite text editor:

   ```
   # The "admin" user has password "activemq".
   admin activemq
   ```

   The preceding file defines a single user identity, `admin`, and the corresponding password, `activemq`.

2. If not already present in the *`ACTIVEMQ_BASE`*`/conf/` directory, create the access file, *`ACTIVEMQ_BASE`*`/conf/jmx.access`, and add the following lines:

   ```
   # The "admin" user has readwrite access.
   admin readwrite
   ```

   This file enables you to define two kinds of access:

   `readonly`
   : Users can read MBean attributes.

   `readwrite`
   : Users can read and write MBean attributes, invoke operations on MBeans, and create and delete MBeans. Additional clauses can be added to `readwrite` access in order to specify which types of MBean users can create and to specify whether or not users have permission to delete MBeans. For details of this syntax, see the comments in the `JAVA_HOME/jre/lib/management/jmxremote.access` file.

3. Using file system permissions, ensure that the `jmx.password` file you created in the previous step is readable and writable only by the user that runs the Apache ActiveMQ broker. All other users must have read and write permissions disabled.

   Windows
   : Form Windows Explorer, right-click on `jmx.password` and select **Properties**. Click on the **Security** tab and remove all groups or users that have access to this file, except for the current user.

   *NIX
   : While logged in as the user that runs the broker, enter the following command to clear all permissions for group and others on the `jmx.password` file:

     ```
     chmod u=rw,go= jmx.password
     ```

> **Note**
>
> If you do not modify the file permissions as specified here, the Apache ActiveMQ broker will refuse to start up, when JMX authentication is enabled.

4. Using your favorite text editor, edit the broker configuration file, *`ACTIVEMQ_BASE`*`/conf/activemq.xml`, adding the following lines to configure the JMX connector:

```
<beans ... >
  <broker xmlns="http://activemq.apache.org/schema/core" ... >
    ...
    <managementContext>
      <managementContext createConnector="true"
                         connectorPort="2011"
                         jmxDomainName="org.apache.activemq">
        <property xmlns="http://www.springframework.org/schema/beans" name="environment">
          <map xmlns="http://www.springframework.org/schema/beans">
            <entry xmlns="http://www.springframework.org/schema/beans"
                   key="jmx.remote.x.password.file"
                   value="${activemq.base}/conf/jmx.password"/>
            <entry xmlns="http://www.springframework.org/schema/beans"
                   key="jmx.remote.x.access.file"
                   value="${activemq.base}/conf/jmx.access"/>
          </map>
        </property>
      </managementContext>
    </managementContext>
    ...
  </broker>
```

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

Warning

In the current example, SSL is disabled. This configuration is *not* recommended in a production environment, because it leaves your JMX login credentials vulnerable to snooping.

5. Start up the standalone broker. Open a new command prompt and run the startup script, as follows:

   ```
   bin/activemq
   ```

6. You should now be able to connect to the JVM platform JMX connector using the following JMX URL:

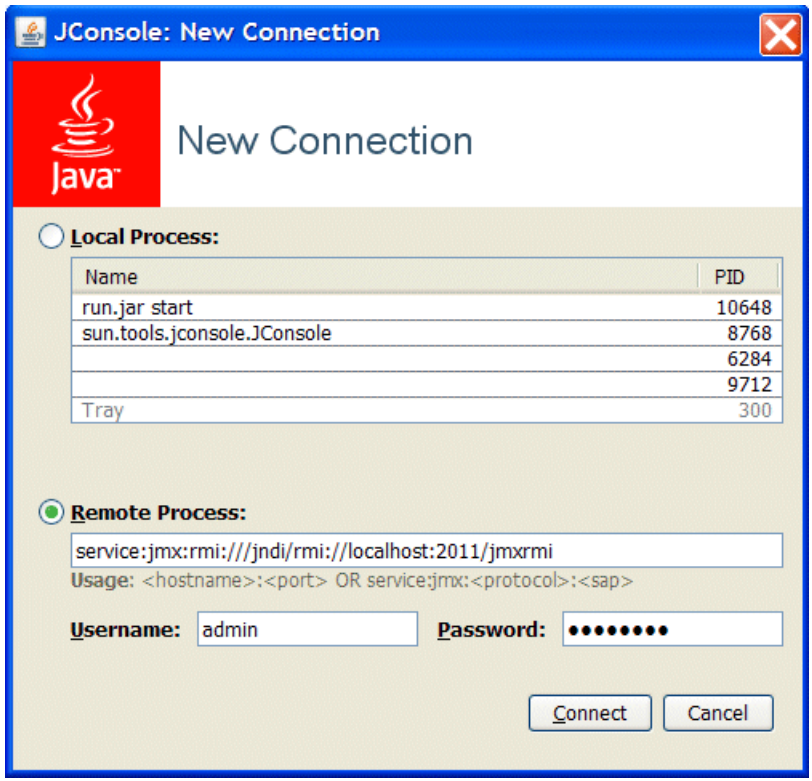
   ```
   service:jmx:rmi:///jndi/rmi://Hostname:2011/jmxrmi
   ```

   Where you substitute *`Hostname`* with the name of the host where the broker is running. In the course of establishing the connection, you will be prompted to log in.

   For example, to run the standard JConsole provided with the JDK, enter the following command at a command prompt:

   ```
   jconsole
   ```

   When the **JConsole: New Connection** dialog pops up, enter the preceding JMX URL in the **Remote Process:** field, and enter the credentials for one of the JMX users in the **Username** and **Password** fields, as shown. Click **Connect**.



Enable JAAS authentication

To enable JAAS authentication for the Apache ActiveMQ JMX connector, perform the following steps:

1. Using your favorite text editor, create the file, `conf/login.config`, and add the following lines:

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

Note

The current example illustrates how to configure JAAS using the `PropertiesLoginModule`, as described in "JAAS Username/Password Authentication Plug-In" on page 64. In practice, however, you could use any of the login modules described in "JAAS Authentication" on page 61.

2. Create the `conf/users.properties` file (which defines credentials in the format, *`User=Password`*), and add the following line:

   ```
   system=manager
   ```

   Create the `conf/groups.properties` file (which defines user groups in the format, *`Group=UserList`*), and add the following lines:

   ```
   admins=system
   readwrite=system
   users=system
   ```

3. Using your favorite text editor, edit the broker configuration file, *`ACTIVEMQ_BASE`*`/conf/activemq.xml`, adding the following lines to configure the JMX connector:

```
<beans ... >
  <broker xmlns="http://activemq.apache.org/schema/core" ... >
    ...
    <managementContext>
      <managementContext createConnector="true" connectorPort="2010">
        <property xmlns="http://www.springframework.org/schema/beans" name="environment">
          <map xmlns="http://www.springframework.org/schema/beans">
            <entry xmlns="http://www.springframework.org/schema/beans"
                   key="jmx.remote.x.login.config"
                   value="PropertiesAuth"/>
          </map>
        </property>
      </managementContext>
    </managementContext>
    ...
  </broker>
  ...
</beans>
```

Warning

In the current example, SSL is disabled. This configuration is *not* recommended in a production environment, because it leaves your JMX login credentials vulnerable to snooping.

4. Start up the standalone broker. Open a new command prompt and run the startup script, as follows:

```
bin/activemq
```

5. Using a JMX client, you should now be able to connect to the JVM platform JMX connector using the following JMX URL:

```
service:jmx:rmi:///jndi/rmi://Hostname:2010/jmxrmi
```

When prompted, enter one of the configured credentials to log on to JMX—for example, with the username, `system`, and the password, `manager`.

SSL/TLS for the Apache ActiveMQ JMX Connector

Overview

Enable SSL/TLS

```
<beans ... >
  <broker xmlns="http://activemq.apache.org/schema/core" ... >
    ...
    <managementContext>
      <managementContext createConnector="true"
                         connectorPort="2011"
                         jmxDomainName="org.apache.activemq">
        <property xmlns="http://www.springframework.org/schema/beans" name="environment">
          <map xmlns="http://www.springframework.org/schema/beans">
            <entry xmlns="http://www.springframework.org/schema/beans"
                   key="javax.net.ssl.keyStore"
                   value="${activemq.base}/conf/broker.ks"/>
            <entry xmlns="http://www.springframework.org/schema/beans"
                   key="javax.net.ssl.keyStorePassword"
                   value="password"/>
          </map>
        </property>
      </managementContext>
    </managementContext>
    ...
  </broker>
  ...
</beans>
```

Test the SSL/TLS connection

```
jconsole -J-Djavax.net.ssl.trustStore=ActiveMQInstallDir/conf/client.ts
         -J-Djavax.net.ssl.trustStorePassword=password
```

Tip

Don't forget the `-J` switch, which passes the options through to the underlying Java virtual machine (JVM).

Platform JMX Connector

Authentication for the Platform JMX Connector 108
SSL/TLS for the Platform JMX Connector 114

Authentication for the Platform JMX Connector

Prerequisites

In order for Fuse Message Broker to use the JVM platform JMX connector, you must disable the Apache ActiveMQ JMX connector by setting the `createConnector` attribute to `false` on the `managementContext` element. Edit the default broker configuration file, *`ACTIVEMQ_BASE`*`/conf/activemq.xml` using your favorite text editor, so that the `managementContext` element is defined as follows:

```
<broker xmlns="http://activemq.org/config/1.0" brokerName="localhost" useJmx="true">
  ...
  <managementContext>
     <managementContext createConnector="false"/>
  </managementContext>
  ...
</broker>
```

Authentication alternatives

The following alternative authentication mechanisms are supported for the platform JMX connector:

- *Simple authentication*—configure authentication by providing two files: a `jmx.password` file, which contains login details, and a `jmx.access` file, which defines access rights for the defined users.
- *JAAS authentication*—configure authentication using JAAS realms and login modules.

Enable simple authentication

To enable the simple authentication mechanism for the JVM platform JMX connector, perform the following steps:

1. If not already present in the *`ACTIVEMQ_BASE`*`/conf/` directory, create the password file, *`ACTIVEMQ_BASE`*`/conf/jmx.password`, and add the following lines using your favorite text editor:

   ```
   # The "admin" user has password "activemq".
   admin activemq
   ```

   The preceding file defines a single user identity, `admin`, and the corresponding password, `activemq`.

2. If not already present in the *`ACTIVEMQ_BASE`*`/conf/` directory, create the access file, *`ACTIVEMQ_BASE`*`/conf/jmx.access`, and add the following lines:

   ```
   # The "admin" user has readwrite access.
   admin readwrite
   ```

This file enables you to define two kinds of access:

`readonly`
: Users can read MBean attributes.

`readwrite`
: Users can read and write MBean attributes, invoke operations on MBeans, and create and delete MBeans. Additional clauses can be added to `readwrite` access in order to specify which types of MBean users can create and to specify whether or not users have permission to delete MBeans. For details of this syntax, see the comments in the `JAVA_HOME/jre/lib/management/jmxremote.access` file.

3. Using file system permissions, ensure that the `jmx.password` file you created in the previous step is readable and writable only by the user that runs the Apache ActiveMQ broker. All other users must have read and write permissions disabled.

   Windows
   : Form Windows Explorer, right-click on `jmx.password` and select **Properties**. Click on the **Security** tab and remove all groups or users that have access to this file, except for the current user.

   *NIX
   : While logged in as the user that runs the broker, enter the following command to clear all permissions for group and others on the `jmx.password` file:

   ```
   chmod u=rw,go= jmx.password
   ```

Note

If you do not modify the file permissions as specified here, the Apache ActiveMQ broker will refuse to start up, when JMX authentication is enabled.

4. Modify the `activemq[.bat]` startup script in the *`ACTIVEMQ_BASE`*`/bin/` directory, as appropriate for your platform:

   Windows
   : Search the `activemq.bat` script for `SUNJMX` and replace the lines you find with the following lines:

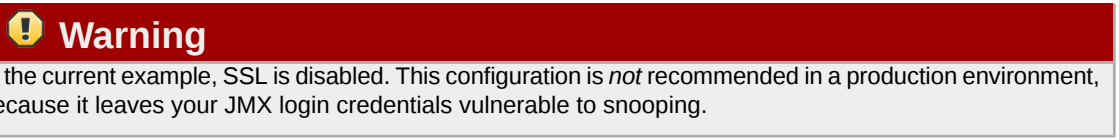
   ```
   set SUNJMX=-Dcom.sun.management.jmxremote
   set SUNJMX=%SUNJMX% -Dcom.sun.management.jmxremote.port=11099
   set SUNJMX=%SUNJMX% -Dcom.sun.management.jmxremote.password.file=%ACT
   IVEMQ_HOME%\conf\jmx.password
   set SUNJMX=%SUNJMX% -Dcom.sun.management.jmxremote.access.file=%ACT
   ```

```
IVEMQ_HOME%\conf\jmx.access
set SUNJMX=%SUNJMX% -Dcom.sun.management.jmxremote.ssl=false
```

*NIX

Search the `activemq` script for `ACTIVEMQ_SUNJMX_START` and uncomment the following lines:

```
ACTIVEMQ_SUNJMX_START="-Dcom.sun.management.jmxremote.port=11099 "
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.pass
word.file=${ACTIVEMQ_CONFIG_DIR}/jmx.password"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.access.file=${ACT
IVEMQ_CONFIG_DIR}/jmx.access"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.ssl=false"
```



5. Start up the standalone broker. Open a new command prompt and run the startup script, as follows:

   ```
   bin/activemq
   ```

6. You should now be able to connect to the JVM platform JMX connector using the following JMX URL:

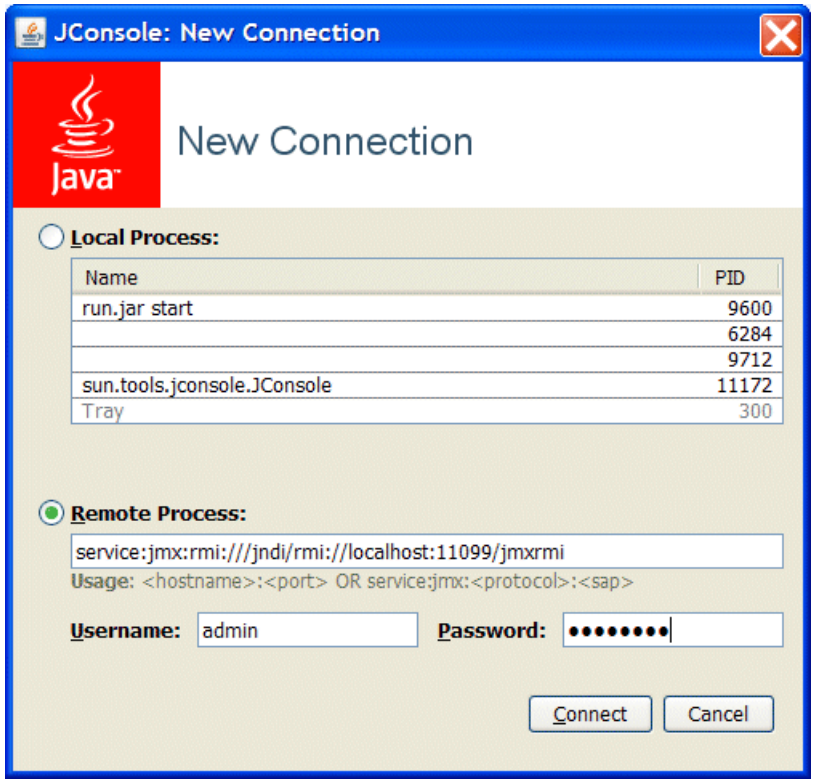
   ```
   service:jmx:rmi:///jndi/rmi://Hostname:11099/jmxrmi
   ```

   Where you substitute *`Hostname`* with the name of the host where the broker is running. In the course of establishing the connection, you will be prompted to log in.

   For example, to run the standard JConsole provided with the JDK, enter the following command at a command prompt:

   ```
   jconsole
   ```

   When the **JConsole: New Connection** dialog pops up, enter the preceding JMX URL in the **Remote Process:** field, and enter the credentials for one of the JMX users in the **Username** and **Password** fields, as shown. Click **Connect**.



Enable JAAS authentication

To enable JAAS authentication for the JVM platform JMX connector, perform the following steps:

1. Using your favorite text editor, create the file, `conf/login.config`, and add the following lines:

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

Note

The current example illustrates how to configure JAAS using the `PropertiesLoginModule`, as described in "JAAS Username/Password Authentication Plug-In" on page 64. In practice, however, you could use any of the login modules described in "JAAS Authentication" on page 61.

2. Create the `conf/users.properties` file (which defines credentials in the format, *`User=Password`*), and add the following line:

```
system=manager
```

   Create the `conf/groups.properties` file (which defines user groups in the format, *`Group=UserList`*), and add the following lines:

```
admins=system
readwrite=system
users=system
```

3. Modify the `activemq[.bat]` startup script in the *`ACTIVEMQ_BASE`*`/bin/` directory, as appropriate for your platform:

   Windows
   : Search the `activemq.bat` script for `SUNJMX` and replace the lines you find with the following lines:

```
set SUNJMX=-Dcom.sun.management.jmxremote
set SUNJMX=%SUNJMX% -Dcom.sun.management.jmxremote.port=11099
set SUNJMX=%SUNJMX% -Djava.security.auth.login.config=%ACTIVEMQ_HOME%\conf\login.config
set SUNJMX=%SUNJMX% -Dcom.sun.management.jmxremote.login.config=PropertiesAuth
set SUNJMX=%SUNJMX% -Dcom.sun.management.jmxremote.ssl=false
```

   *NIX
   : Search the `activemq` script for `ACTIVEMQ_SUNJMX_START` and replace the lines you find with the following lines:

```
ACTIVEMQ_SUNJMX_START="-Dcom.sun.management.jmxremote"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.port=2010"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.login.con
fig=PropertiesAuth"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Djava.security.auth.login.config=${ACTIVEMQ_CON
FIG_DIR}/login.config"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.ssl=false"
```

Warning

In the current example, SSL is disabled. This configuration is *not* recommended in a production environment, because it leaves your JMX login credentials vulnerable to snooping.

4. Because the JVM platform JMX security is initialized before the Apache ActiveMQ classpath is set, you need to copy the requisite JAR libraries to the JVM extension folder. Copy the following JAR files from *ACTIVEMQ_HOME*`/lib` to *JAVA_HOME*`/lib/ext` (or *JAVA_HOME*`/jre/lib/ext`, as appropriate):

   ```
   activemq-jaas-5.5.1-fuse-00-xx.jar
   commons-logging-CommonsVersion.jar
   ```

   Copy the following JAR file from *ACTIVEMQ_HOME*`/lib/optional` to *JAVA_HOME*`/lib/ext`:

   ```
   log4j-Log4jVersion.jar
   ```

5. Using your favorite text editor, modify the file, *JAVA_HOME*`/jre/lib/management/jmxremote.access`, to add access rights to your users, by adding the following line:

   ```
   system readwrite
   ```

6. Start up the standalone broker. Open a new command prompt and run the startup script, as follows:

   ```
   bin/activemq
   ```

7. Using a JMX client, you should now be able to connect to the JVM platform JMX connector using the following JMX URL:

   ```
   service:jmx:rmi:///jndi/rmi://Hostname:11099/jmxrmi
   ```

   When prompted, enter one of the configured credentials to log on to JMX—for example, with the username, `system`, and the password, `manager`.

SSL/TLS for the Platform JMX Connector

Overview

System properties for SSL/TLS

To enable SSL/TLS on the platform JMX connector, you need to set the following system properties in the environment (see also "Configuring JSSE System Properties" on page 24):

`com.sun.management.jmxremote.ssl`
: A boolean property, which must be set to `true` to enable SSL/TLS on the JMX endpoint.

`javax.net.ssl.keyStore`
: Specifies the location of the key store containing the broker's own X.509 certificate.

`javax.net.ssl.keyStorePassword`
: Specifies the password that unlocks the key store and decrypts the private key stored in the key store.

Note

It is *not* possible to enable SSL/TLS security without JMX remote authentication. JMX remote authentication is a prerequisite for enabling SSL/TLS on the JMX port.

Enable SSL/TLS

Modify the `activemq[.bat]` startup script in the *`ACTIVEMQ_BASE`*`/bin/` directory, as appropriate for your platform:

Windows
: Search the `activemq.bat` script for `SUNJMX` and replace the lines you find with the following lines:

```
set SUNJMX=-Dcom.sun.management.jmxremote
set SUNJMX=%SUNJMX% -Dcom.sun.management.jmxremote.ssl=true
set SUNJMX=%SUNJMX% -Dcom.sun.management.jmxremote.port=11099
set SUNJMX=%SUNJMX% -Dcom.sun.management.jmxremote.password.file=%ACT
IVEMQ_HOME%\conf\jmx.password
set SUNJMX=%SUNJMX% -Dcom.sun.management.jmxremote.access.file=%ACTIVEMQ_HOME%\conf\jmx.ac
cess
set SUNJMX=%SUNJMX% -Djavax.net.ssl.keyStore=%ACTIVEMQ_HOME%\conf\broker.ks
set SUNJMX=%SUNJMX% -Djavax.net.ssl.keyStorePassword=password
```

*NIX

Search the `activemq` script for `ACTIVEMQ_SUNJMX_START` and replace the lines you find with the following lines:

```
ACTIVEMQ_SUNJMX_START="-Dcom.sun.management.jmxremote.port=11099 "
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.pass
word.file=${ACTIVEMQ_CONFIG_DIR}/jmx.password"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.access.file=${ACT
IVEMQ_CONFIG_DIR}/jmx.access"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote.ssl=true"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Dcom.sun.management.jmxremote"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Djavax.net.ssl.keyStore=${ACTIVEMQ_CON
FIG_DIR}/broker.ks"
ACTIVEMQ_SUNJMX_START="$ACTIVEMQ_SUNJMX_START -Djavax.net.ssl.keyStorePassword=password"
```

If you have already enabled JMX remote authentication (as described in "Authentication for the Platform JMX Connector" on page 108) the changes that you need to make here are to enable SSL, by setting `com.sun.management.jmxremote.ssl` to `true`, and to add the `keyStore` and `keyStorePassword` settings as shown.

Test the secure connection

To test the secure platform JMX connector, perform the following steps:

1. Start up the standalone broker. Open a new command prompt and run the startup script, as follows:

   ```
   bin/activemq
   ```

2. Start up the JConsole with the required SSL/TLS client settings, as follows:

   ```
   jconsole -J-Djavax.net.ssl.trustStore=ActiveMQInstallDir/conf/client.ts
            -J-Djavax.net.ssl.trustStorePassword=password
   ```

   Where the `jconsole` command uses the standard JSSE system properties to specify the relevant client trust store (see "Configuring JSSE System Properties" on page 24 for details).

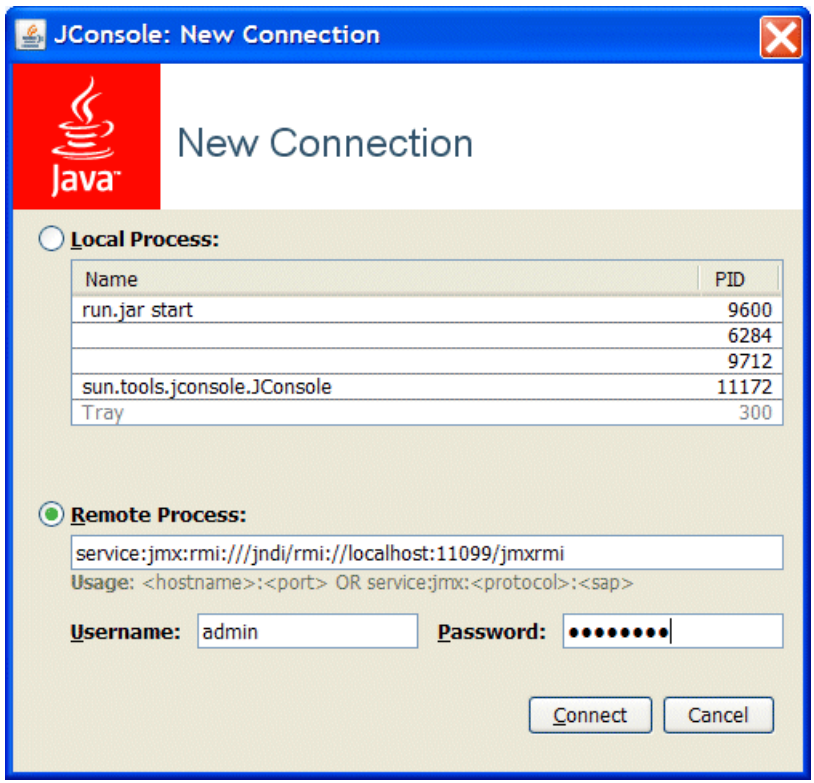
> **Tip**
>
> Don't forget the `-J` switch, which passes the options through to the underlying Java virtual machine (JVM).

3. You should be able to connect to the JVM platform JMX connector using the following JMX URL:

   ```
   service:jmx:rmi:///jndi/rmi://Hostname:11099/jmxrmi
   ```

Where you substitute *`Hostname`* with the name of the host where the broker is running.

When the **JConsole: New Connection** dialog pops up, enter the preceding JMX URL in the **Remote Process:** field, and enter the credentials for one of the JMX users in the **Username** and **Password** fields, as shown. Click **Connect**.



Chapter 6. LDAP Tutorial

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

Tutorial Overview 118
Tutorial: Install a Directory Server and Browser 119
Tutorial: Add User Entries and Group Entries 121
Tutorial: Enable LDAP Authentication in the Broker and its Clients 131
Tutorial: Add Authorization Entries 134
Tutorial: Enable LDAP Authorization in the Broker 139

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 and authorization 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 119.
2. "Tutorial: Add User Entries and Group Entries" on page 121.
3. "Tutorial: Enable LDAP Authentication in the Broker and its Clients" on page 131.
4. "Tutorial: Add Authorization Entries" on page 134.
5. "Tutorial: Enable LDAP Authorization in the Broker" on page 139.

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:

- *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].

[1] http://directory.apache.org/apacheds/1.5/13-installing-and-starting-the-server.html
[2] http://directory.apache.org/studio/downloads.html
[3] http://directory.apache.org/studio/static/users_guide/apache_directory_studio/download_install.html

- *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].

[4] http://directory.apache.org/studio/static/users_guide/ldap_browser/gettingstarted_download_install.html

Tutorial: Add User Entries and Group Entries

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 and group entries. For users who are unfamiliar with X.500 directory servers, this section briefly describes how to create user entries and group entries using the Apache Directory Studio as an administrative tool.

Alternative approach

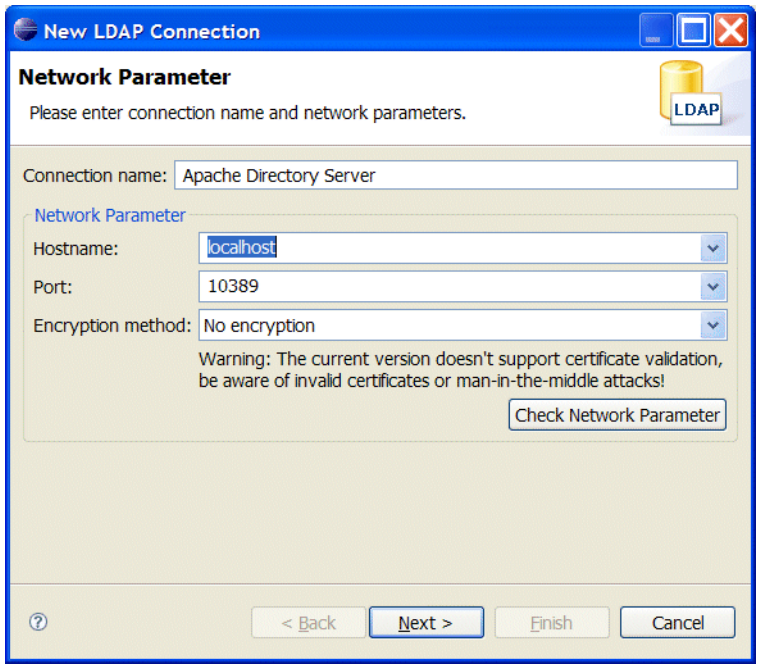
As an alternative to creating the user entries and group entries manually, as described here, you could create the entries by importing an LDIF file—for details, see Appendix B on page 147.

Steps to add a user entry

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

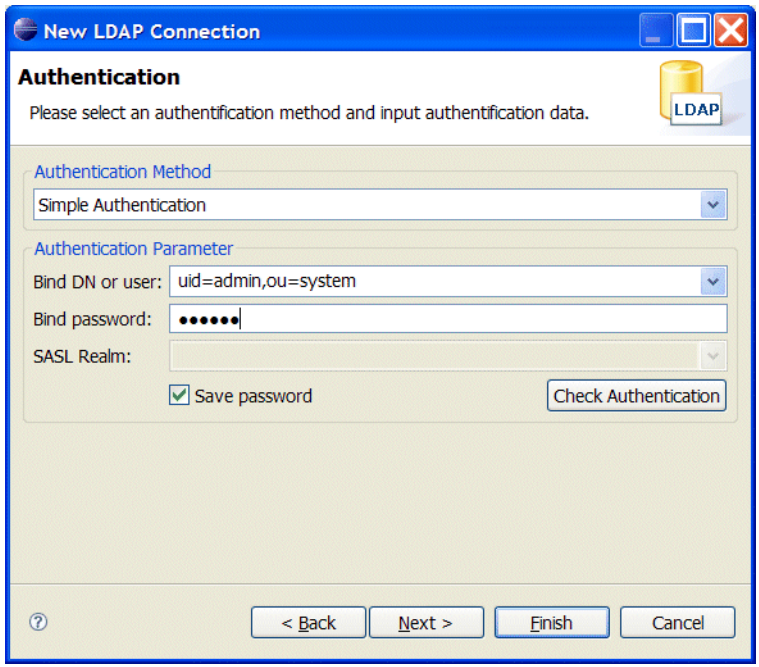
1. Ensure that the X.500 directory server is running (see "Install Apache Directory Server" on page 119).
2. Start the *LDAP Browser*, as follows:
   - If you installed the standalone version of Apache Directory Studio, double-click the relevant icon to launch the application.
   - If you installed the *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 6.1. New LDAP Connection Wizard

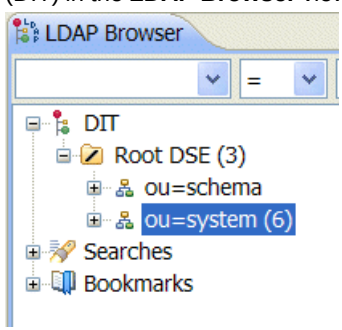


5. Enter the parameters for simple authentication. In the **Bind DN or user** field, enter the DN of the administrator's account on the directory server (for the default instance of the Apache directory server, this is `uid=admin,ou=system`). In the **Bind password** field, enter the administrator's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

Figure 6.2. Authentication Step of New LDAP Connection



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=system` node, as shown.

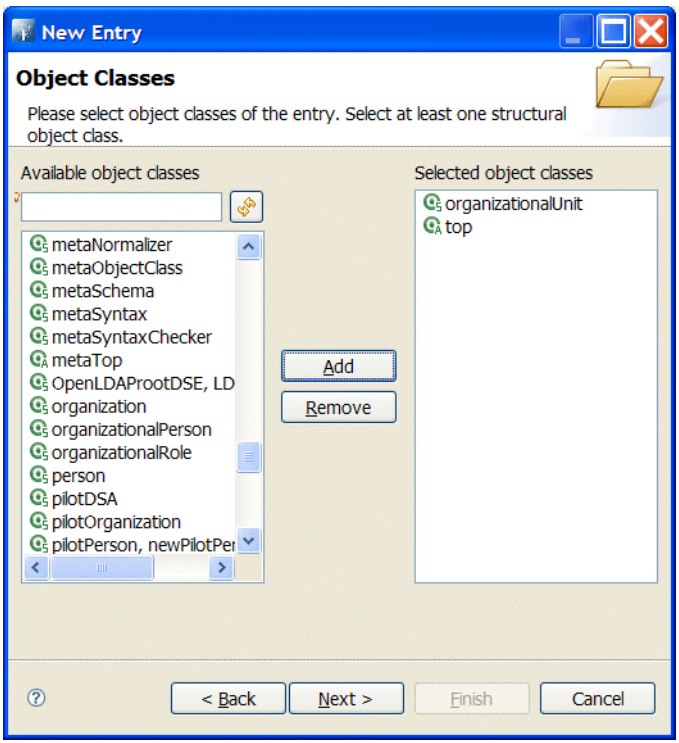


7. The next few steps describe how to create some new nodes to hold the user entries and group entries.

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

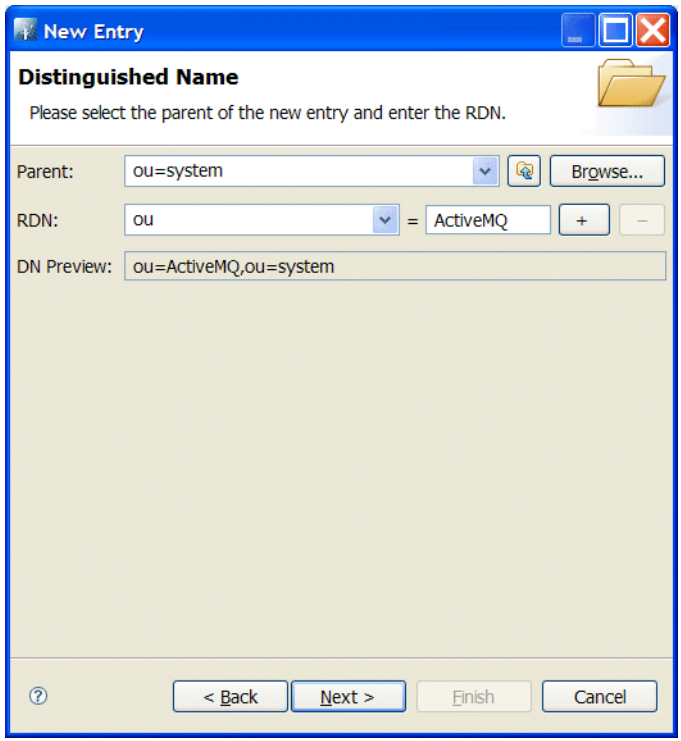
8. In the **Entry Creation Method** pane, select the **Create entry from scratch** radiobutton. Click **Next**.
9. In the **Object Classes** pane, select `organisationalUnit` from the list of **Available object classes** on the left and then click **Add** to populate the list of **Selected object classes**. Click **Next**.

Figure 6.3. New Entry Wizard



10. In the **Distinguished Name** pane, complete the **RDN** field, putting `ou` in front and `ActiveMQ` after the equals sign. Click **Next** and then click **Finish**.

Figure 6.4. Distinguished Name Step of New Entry Wizard

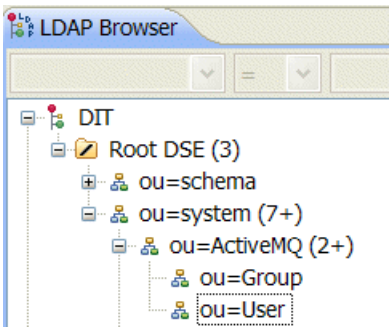


11. In a similar manner as described in steps 7–10, by right-clicking on the `ou=ActiveMQ` node and invoking the **New Entry** wizard, create the following `organisationalUnit` nodes as children of the `ou=ActiveMQ` node:

```
ou=User,ou=ActiveMQ,ou=system
ou=Group,ou=ActiveMQ,ou=system
```

In the LDAP Browser window, you should now see the following tree:

Figure 6.5. DIT after Creating ActiveMQ, User, and Group Nodes



12. The next few steps describe how to create a `jdoe` user entry.

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

13. In the **Entry Creation Method** pane, select the **Create entry from scratch** radiobutton. Click **Next**.

14. 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**.

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

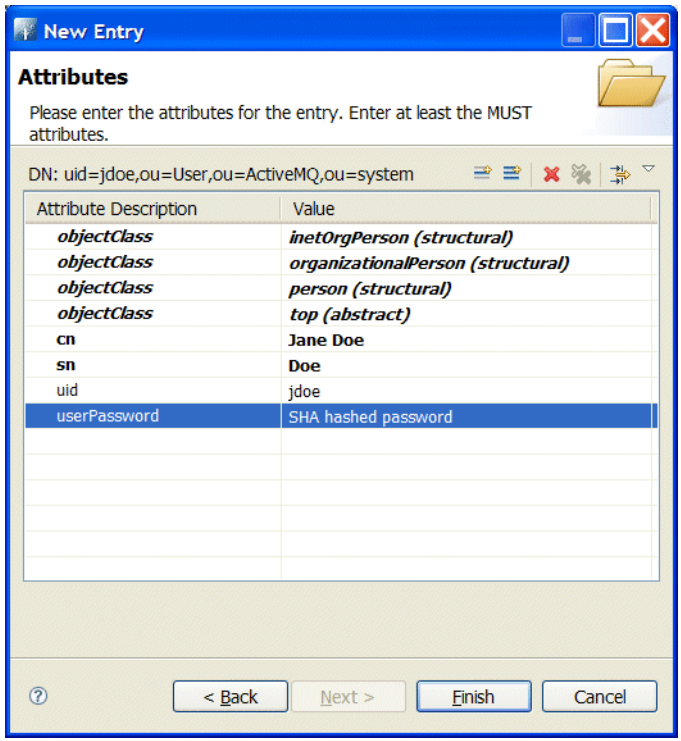
16. Now fill in the mandatory attributes in the **Attributes** pane. Set the **cn** (common name) attribute to `Jane Doe` and the **sn** (surname) attribute to `Doe`.

17. Add a `userPassword` attribute in the **Attributes** pane. Right-click inside the list of attributes and select **New Attribute**. The **New Attribute** wizard appears.

18. From the **Attribute type** drop-down list, select `userPassword`. Click **Finish**.

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

Figure 6.6. Attributes Step of New Entry Wizard



20. Click **Finish**, to close the **New Entry** wizard.

21. The next few steps describe how to create an `admin` user entry.

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

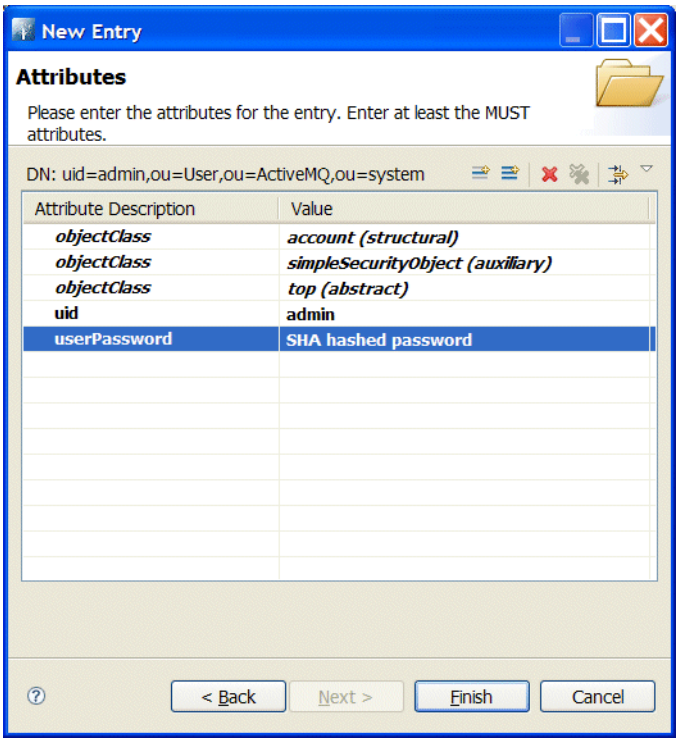
22. In the **Entry Creation Method** pane, select the **Create entry from scratch** radiobutton. Click **Next**.

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

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

25. You are now prompted to provide a password, through the **Password Editor** dialog. In the **Enter New Password** field, enter the password, `sunflower`. Click **Ok**.

Figure 6.7. Attributes Step of New Entry Wizard



26. Click **Finish**, to close the **New Entry** wizard.

27. The next few steps describe how to create the `admins` group entry.

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

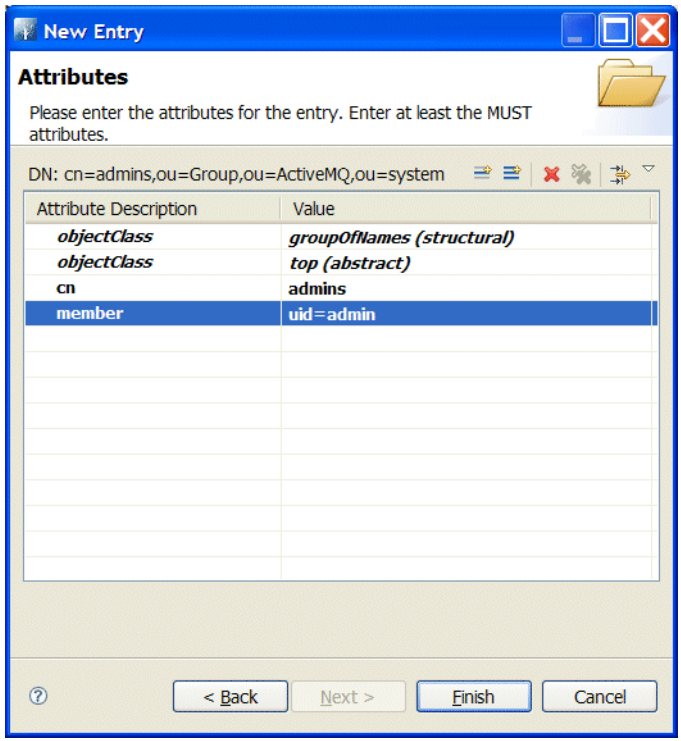
28. In the **Entry Creation Method** pane, select the **Create entry from scratch** radiobutton. Click **Next**.

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

30. In the **Distinguished Name** pane, complete the **RDN** field, putting `cn` in front and `admins` after the equals sign. Click **Next**.

31. You are now prompted to provide a value for the mandatory `member` attribute, through the **DN Editor** dialog. In the text field, enter the last part of the DN for the `admin` user, `uid=admin`. Click **Ok**.

Figure 6.8. Attributes Step of New Entry Wizard



32. Click **Finish**, to close the **New Entry** wizard.

33. The next few steps describe how to create the `users` group entry.

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

34. In the **Entry Creation Method** pane, select the **Create entry from scratch** radiobutton. Click **Next**.

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

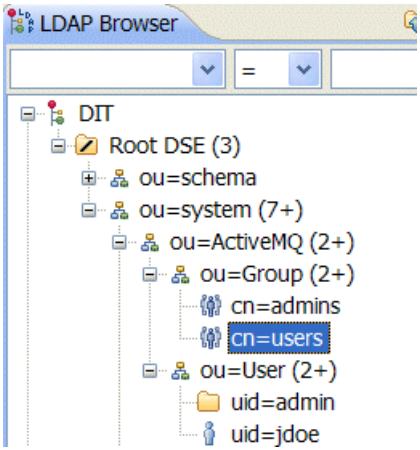
36. In the **Distinguished Name** pane, complete the **RDN** field, putting `cn` in front and `users` after the equals sign. Click **Next**.

37. You are now prompted to provide a value for the mandatory `member` attribute, through the **DN Editor** dialog. In the text field, enter the last part of the DN for the `jdoe` user, `uid=jdoe`. Click **Ok**.

38. Click **Finish**, to close the **New Entry** wizard.

39. You should now be able to see the following tree in the **LDAP Browser** window:

Figure 6.9. Complete Tree of User Entries and Group Entries



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.config` under the directory, `$ACTIVEMQ_HOME/conf`. Paste the following text into the `login.config` file:

```
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=User,ou=ActiveMQ,ou=system"
    userSearchMatching="(uid={0})"
    userSearchSubtree=false
    roleBase="ou=Group,ou=ActiveMQ,ou=system"
    roleName=cn
    roleSearchMatching="(member=uid={1})"
    roleSearchSubtree=false
    ;
};
```

   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.

Note

If you are using the OpenLDAP Directory Server, the syntax required for the `roleSearchMatching` property is different. You must set it as `roleSearchMatching="(member:=uid={1})"`.

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.config` 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 = "sunflower";
  ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(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.

Tutorial: Add Authorization Entries

Overview

Before enabling LDAP authorization in the broker, you need to create a suitable tree of entries in the directory server to represent permissions. You need to create the following kinds of entry:

Queue entries
: For each queue in your application, you need to create an entry that specifies the admin, read, and write permissions.

Topic entries
: For each topic in your application, you need to create an entry that specifies the admin, read, and write permissions.

Advisory topics entry
: A single advisory topics entry contains the admin, read, and write permissions that apply to *all* advisory topics.

Temporary queues entry
: A single temporary queues entry contains the admin, read, and write permissions that apply to *all* temporary queues.

Alternative approach

As an alternative to creating the authorization entries manually, as described here, you could create the entries by importing an LDIF file—for details, see Appendix B on page 147.

Steps to add authorization entries

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

1. The next few steps describe how to create the `ou=Destination`, `ou=Queue`, and `ou=Topic` nodes.

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

2. In the **Entry Creation Method** pane, select the **Create entry from scratch** radiobutton. Click **Next**.

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

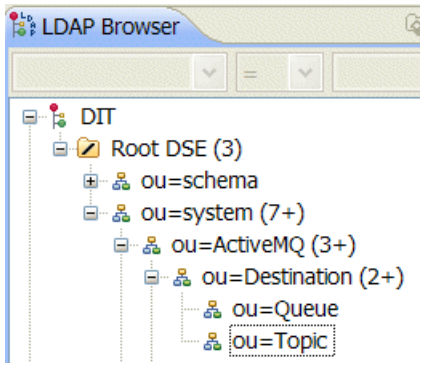
4. In the **Distinguished Name** pane, complete the **RDN** field, putting `ou` in front and `Destination` after the equals sign. Click **Next** and then click **Finish**.

5. In a similar manner as described in steps 1–4, by right-clicking on the `ou=Destination` node and invoking the **New Entry** wizard, create the following `organisationalUnit` nodes as children of the `ou=Destination` node:

```
ou=Queue,ou=Destination,ou=ActiveMQ,ou=system
ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
```

   In the LDAP Browser window, you should now see the following tree:

Figure 6.10. DIT after Creating Destination, Queue, and Topic Nodes



6. The next few steps describe how to create the `cn=TEST.FOO,ou=Queue,ou=Destination`, `cn=ActiveMQ.Advisory,ou=Topic,ou=Destination`, and `cn=ActiveMQ.Temp,ou=Topic,ou=Destination` nodes.

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

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

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

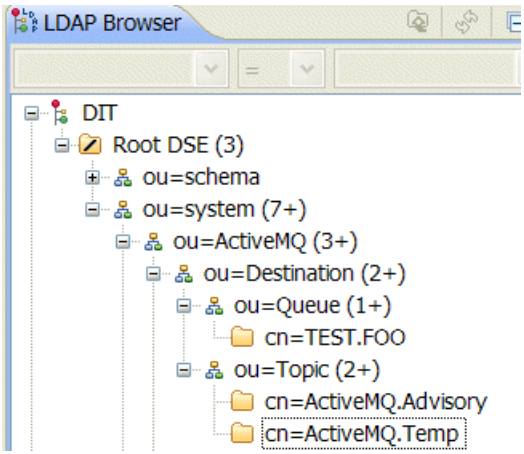
9. In the **Distinguished Name** pane, complete the **RDN** field, putting `cn` in front and `TEST.FOO` after the equals sign. Click **Next** and then click **Finish**.

10. In a similar manner as described in steps 6–9, by right-clicking on the `ou=Topic` node and invoking the **New Entry** wizard, create the following `applicationProcess` nodes as children of the `ou=Topic` node:

```
cn=ActiveMQ.Advisory,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
cn=ActiveMQ.Temp,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
```

    In the LDAP Browser window, you should now see the following tree:

Figure 6.11. DIT after Creating Children of Queue and Topic Nodes



11. The next few steps describe how to create nodes that represent `admin`, `read`, and `write` permissions for the queues and topics.

    Right-click on the `cn=TEST.FOO` node and select **New** → **New Entry**. The **New Entry** wizard appears.

12. In the **Entry Creation Method** pane, select the **Create entry from scratch** radiobutton. Click **Next**.

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

14. In the **Distinguished Name** pane, complete the **RDN** field, putting `cn` in front and `admin` after the equals sign. Click **Next**.

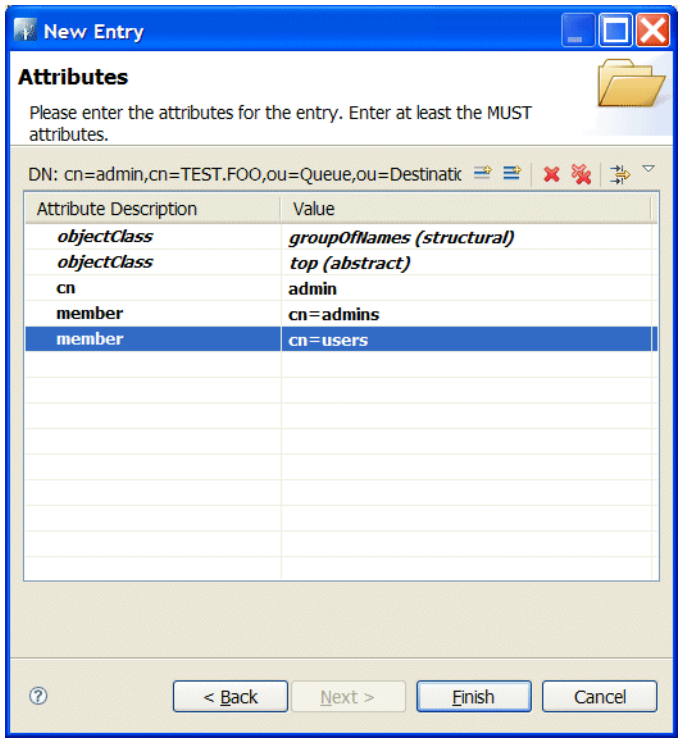
15. You are now prompted to provide a value for the mandatory `member` attribute, through the **DN Editor** dialog. In the text field, enter the last part of the DN for the `admins` group, `cn=admins`. Click **Ok**.

16. Add another `member` attribute in the **Attributes** pane. Right-click inside the list of attributes and select **New Attribute**. The **New Attribute** wizard appears.

17. In the **Attribute type** field, enter `member` (if you want to use the drop-down list, you must first uncheck the **Hide existing attributes** option). Click **Finish**.

18. The **DN Editor** dialog opens. In the text field, enter the last part of the DN for the `users` group, `cn=users`. Click **Ok**.

Figure 6.12. Attributes of the cn=admin Permission Node



19. Click **Finish**, to close the **New Entry** wizard.

20. In a similar manner as described in steps 11–19, by right-clicking on the `cn=TEST.FOO` node and invoking the **New Entry** wizard, create the following `groupOfNames` nodes as children of the `cn=TEST.FOO` node:

```
cn=read,cn=TEST.FOO,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system
cn=write,cn=TEST.FOO,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system
```

    The new `cn=read` node and the new `cn=write` node should include both of the members, `cn=admins` and `cn=users`.

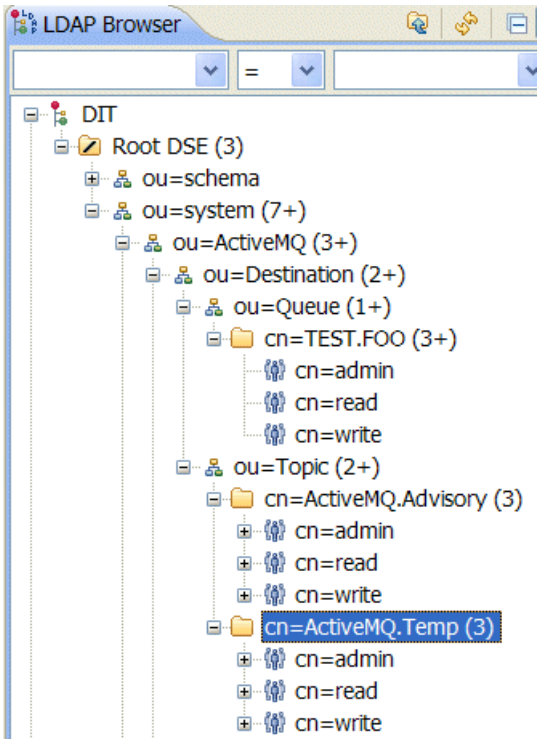
21. Copy the `cn=admin`, `cn=read`, and `cn=write` permission nodes and paste them as children of the `cn=ActiveMQ.Advisory` node, as follows.

Using a combination of mouse and keyboard, select the three nodes, `cn=admin`, `cn=read`, and `cn=write`, and type `Ctrl-C` to copy them. Select the `cn=ActiveMQ.Advisory` node and type `Ctrl-V` to paste the copied nodes as children.

22. Similarly, copy the `cn=admin`, `cn=read`, and `cn=write` permission nodes and paste them as children of the `cn=ActiveMQ.Temp` node.

23. In the LDAP Browser window, you should now see the following tree:

Figure 6.13. DIT after Creating Children of Queue and Topic Nodes



Tutorial: Enable LDAP Authorization in the Broker

Overview

This section explains how to enable LDAP authorization in the broker, so that the broker obtains its authorization data from the directory server. For each queue and topic, you can specify three different kinds of permission:

- *admin*—allows you to create and destroy topics or queues.
- *read*—allows you to read messages from topics or queues.
- *write*—allows you to write messages to topics or queues.

Steps to enable LDAP authorization

Perform the following steps to enable LDAP authorization:

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

```
<beans ...>
  <broker ...>
    ...
    <plugins>
      ...
      <authorizationPlugin>
        <map>
          <bean id="lDAPAuthorizationMap" class="org.apache.activemq.security.LDAPAuthor
izationMap"
                xmlns="http://www.springframework.org/schema/beans">
            <property name="initialContextFactory" value="com.sun.jndi.ldap.LdapCtxFact
ory"/>
            <property name="connectionURL" value="ldap://localhost:10389"/>
            <property name="authentication" value="simple"/>
            <property name="connectionUsername" value="uid=admin,ou=system"/>
            <property name="connectionPassword" value="secret"/>
            <property name="connectionProtocol" value=""/>
            <property name="topicSearchMatchingFormat"
                      value="cn={0},ou=Topic,ou=Destination,ou=ActiveMQ,ou=system"/>
            <property name="topicSearchSubtreeBool" value="true"/>
            <property name="queueSearchMatchingFormat"
                      value="cn={0},ou=Queue,ou=Destination,ou=ActiveMQ,ou=system"/>
            <property name="queueSearchSubtreeBool" value="true"/>
            <property name="advisorySearchBase"
```

```
                      value="cn=ActiveMQ.Advisory,ou=Topic,ou=Destination,ou=Act
iveMQ,ou=system"/>
            <property name="tempSearchBase"
                     value="cn=ActiveMQ.Temp,ou=Topic,ou=Destination,ou=ActiveMQ,ou=sys
tem"/>
            <property name="adminBase" value="(cn=admin)"/>
            <property name="adminAttribute" value="member"/>
            <property name="readBase" value="(cn=read)"/>
            <property name="readAttribute" value="member"/>
            <property name="writeBase" value="(cn=write)"/>
            <property name="writeAttribute" value="member"/>
          </bean>
        </map>
      </authorizationPlugin>
    </plugins>
    ...
  </broker>
</beans>
```

2. If you have not already done so, add username/password credentials to the consumer tool, `example/src/ConsumerTool.java`, and to the producer tool, `example/src/ProducerTool.java`, as described in "Tutorial: Enable LDAP Authentication in the Broker and its Clients" on page 131.

3. 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.

4. Run the broker. Open a new command prompt and start the broker by entering the following command:

```
activemq
```

5. 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
```

6. 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
```

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.1 142
Distinguished Names 143

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 Apache CXF, DNs occur in the following contexts:

- X.509 certificates—for example, one of the DNs in a certificate identifies the owner of the certificate (the security principal).
- LDAP—DNs are used to locate objects in an LDAP directory tree.

String representation of DN

Although a DN is formally defined in ASN.1, there is also an LDAP standard that defines a UTF-8 string representation of a DN (see `RFC 2253`). The string representation provides a convenient basis for describing the structure of a DN.

Note

The string representation of a DN does *not* provide a unique representation of DER-encoded DN. Hence, a DN that is converted from string format back to DER format does not always recover the original DER encoding.

DN string example

The following string is a typical example of a DN:

```
C=US,O=IONA Technologies,OU=Engineering,CN=A. N. Other
```

Structure of a DN string

A DN string is built up from the following basic elements:

- OID .
- Attribute Types .
- AVA .

- RDN .

OID

An OBJECT IDENTIFIER (OID) is a sequence of bytes that uniquely identifies a grammatical construct in ASN.1.

Attribute types

The variety of attribute types that can appear in a DN is theoretically open-ended, but in practice only a small subset of attribute types are used. Table A.1 on page 144 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 144). 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. LDAP Entries as an LDIF File

This appendix provides the complete authentication and authorization entries for the LDAP tutorial in LDIF format. You can use this data to recreate the required directory tree quickly.

Importing from LDIF 148

Importing from LDIF

What is LDIF?

LDAP Data Interchange Format (LDIF) is a draft Internet standard for dumping the contents of an LDAP directory tree to a plain text file. Using a suitable LDIF utility, it is possible to export the contents of a directory tree or subtree to a file and then re-import the directory tree at a later time. Alternatively, you can use an LDIF file to recreate a directory tree in a different directory server instance.

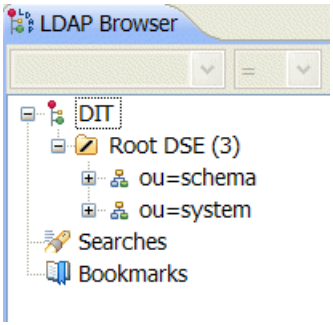
Import LDIF using Apache Directory Studio

To recreate the directory tree for the LDAP tutorial, perform the following steps in Apache Directory Studio:

1. Copy the contents of the LDIF from Example B.1 on page 149 and paste into a text file, `activemq.ldif`, in any convenient location.

> **Important**
>
> Make sure to include a blank line at the end of the file, otherwise Apache Directory Studio will throw an error as it tries to read the last record.

2. Ensure that the X.500 directory server is running.
3. Start up Apache Directory Studio and open a connection to the directory server.
4. In the **LDAP Browser** view, expand the `DIT` node. Right-click on the `Root DSE` node and select **Import** → **LDIF Import**.



5. The **LDIF Import** dialog appears. In the `LDIF File` field, enter the location of the LDIF file to import or use the **Browse** button. Click **Finish**.

LDIF for the LDAP tutorial

Example B.1 on page 149 gives the complete LDIF for recreating the directory tree of the LDAP tutorial.

Example B.1. LDIF for the LDAP Tutorial

```
## ---------------------------------------------------------------------------
## Licensed to the Apache Software Foundation (ASF) under one or more
## contributor license agreements.  See the NOTICE file distributed with
## this work for additional information regarding copyright ownership.
## The ASF licenses this file to You under the Apache License, Version 2.0
## (the "License"); you may not use this file except in compliance with
## the License.  You may obtain a copy of the License at
##
## http://www.apache.org/licenses/LICENSE-2.0
##
## Unless required by applicable law or agreed to in writing, software
## distributed under the License is distributed on an "AS IS" BASIS,
## WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
## See the License for the specific language governing permissions and
## limitations under the License.
## ---------------------------------------------------------------------------


############################
## Define basic objects ##
############################

# Uncomment if adding to open ldap
#dn: ou=system
#objectclass: organizationalUnit
#objectclass: top
#ou: system

dn: ou=ActiveMQ,ou=system
objectClass: organizationalUnit
objectClass: top
ou: ActiveMQ

dn: ou=Services,ou=system
ou: Services
objectClass: organizationalUnit
objectClass: top

dn: cn=mqbroker,ou=Services,ou=system
```

```
cn: mqbroker
objectClass: organizationalRole
objectClass: top
objectClass: simpleSecurityObject
userPassword: {SSHA}YvMAkkd66cDecNoejo8jnw5uUUBziyl0
description: Bind user for MQ broker


####################
## Define groups ##
####################


dn: ou=Group,ou=ActiveMQ,ou=system
objectClass: organizationalUnit
objectClass: top
ou: Group

dn: cn=admins,ou=Group,ou=ActiveMQ,ou=system
cn: admins
member: uid=admin
objectClass: groupOfNames
objectClass: top

dn: cn=users,ou=Group,ou=ActiveMQ,ou=system
cn: users
member: uid=jdoe
objectClass: groupOfNames
objectClass: top


####################
## Define users ##
####################


dn: ou=User,ou=ActiveMQ,ou=system
objectClass: organizationalUnit
objectClass: top
ou: User

dn: uid=admin,ou=User,ou=ActiveMQ,ou=system
uid: admin
userPassword: {SSHA}YvMAkkd66cDecNoejo8jnw5uUUBziyl0
objectClass: account
objectClass: simpleSecurityObject
objectClass: top
```

```
dn: uid=jdoe,ou=User,ou=ActiveMQ,ou=system
uid: jdoe
userPassword: {SSHA}YvMAkkd66cDecNoejo8jnw5uUUBziyl0
objectclass: inetOrgPerson
objectclass: organizationalPerson
objectclass: person
objectclass: top
cn: Jane Doe
sn: Doe


#########################
## Define destinations ##
#########################

dn: ou=Destination,ou=ActiveMQ,ou=system
objectClass: organizationalUnit
objectClass: top
ou: Destination

dn: ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
objectClass: organizationalUnit
objectClass: top
ou: Topic

dn: ou=Queue,ou=Destination,ou=ActiveMQ,ou=system
objectClass: organizationalUnit
objectClass: top
ou: Queue

## TEST.FOO

dn: cn=TEST.FOO,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system
cn: TEST.FOO
description: A queue
objectClass: applicationProcess
objectClass: top

dn: cn=admin,cn=TEST.FOO,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system
cn: admin
description: Admin privilege group, members are roles
member: cn=admins
member: cn=users
objectClass: groupOfNames
objectClass: top

dn: cn=read,cn=TEST.FOO,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system
cn: read
member: cn=users
```

```
member: cn=admins
objectClass: groupOfNames
objectClass: top

dn: cn=write,cn=TEST.FOO,ou=Queue,ou=Destination,ou=ActiveMQ,ou=system
cn: write
objectClass: groupOfNames
objectClass: top
member: cn=users
member: cn=admins



#######################
## Define advisories ##
#######################

dn: cn=ActiveMQ.Advisory,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
cn: ActiveMQ.Advisory
objectClass: applicationProcess
objectClass: top
description: Advisory topics

dn: cn=read,cn=ActiveMQ.Advisory,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
cn: read
member: cn=admins
member: cn=users
objectClass: groupOfNames
objectClass: top

dn: cn=write,cn=ActiveMQ.Advisory,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
cn: write
member: cn=admins
member: cn=users
objectClass: groupOfNames
objectClass: top

dn: cn=admin,cn=ActiveMQ.Advisory,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
cn: admin
member: cn=admins
member: cn=users
objectClass: groupOfNames
objectClass: top

######################
## Define temporary ##
######################

dn: cn=ActiveMQ.Temp,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
```

```
cn: ActiveMQ.Temp
objectClass: applicationProcess
objectClass: top
description: Temporary destinations

dn: cn=read,cn=ActiveMQ.Temp,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
cn: read
member: cn=admins
member: cn=users
objectClass: groupOfNames
objectClass: top

dn: cn=write,cn=ActiveMQ.Temp,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
cn: write
member: cn=admins
member: cn=users
objectClass: groupOfNames
objectClass: top

dn: cn=admin,cn=ActiveMQ.Temp,ou=Topic,ou=Destination,ou=ActiveMQ,ou=system
cn: admin
member: cn=admins
member: cn=users
objectClass: groupOfNames
objectClass: top

## Important: Leave a blank line after the last record!
```

Index

A

Abstract Syntax Notation One (see ASN.1)
administration
  OpenSSL command-line utilities, 48
ASN.1, 38, 141
  attribute types, 144
  AVA, 144
  OID, 144
ASN.1:
  RDN, 145
attribute value assertion, 144
AVA, 144

B

Basic Encoding Rules (see BER)
BER, 142

C

CA, 38
  choosing a host, 42
  commercial CAs, 41
  index file, 52
  list of trusted, 44
  multiple CAs, 43
  private CAs, 42
  private key, creating, 49
  security precautions, 42
  self-signed, 49
  serial file, 52
  setting up, 48
certificate signing request, 50
  signing, 51
certificates
  chaining, 43
  peer, 43
  public key, 38
  self-signed, 43, 49
  signing, 38, 51
  signing request, 50
  X.509, 38
chaining of certificates, 43
CSR, 50

D

DER, 142
Distinguished Encoding Rules (see DER)
distinguished names
  definition, 143
DN
  definition, 143
  string representation, 143

I

index file, 52

M

multiple CAs, 43

O

OpenSSL, 42
OpenSSL command-line utilities, 48

P

peer certificate, 43
private key, 49
public keys, 38

R

RDN, 145
relative distinguished name, 145
root certificate directory, 44

S

self-signed CA, 49
self-signed certificate, 43
serial file, 52
signing certificates, 38
SSLeay, 42

T

trusted CAs, 44

X

X.500, 141
X.509 certificate
    definition, 38