

Fuse MQ Enterprise **Managing and Monitoring a Broker**

Version 7.1
December 2012

Integration Everywhere

Managing and Monitoring a Broker

Version 7.1

Updated: 07 Jan 2014

Copyright © 2012 Red Hat, Inc. and/or its affiliates.

Trademark Disclaimer

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Fuse, Red Hat, Fuse ESB, Fuse ESB Enterprise, Fuse MQ Enterprise, Fuse Mediation Router, Fuse Message Broker, Fuse Services Framework, Fuse IDE, Fuse HQ, Fuse Management Console, and Integration Everywhere are trademarks or registered trademarks of Red Hat Corp. or its parent corporation, Progress Software Corporation, or one of their subsidiaries or affiliates in the United States. Apache, ServiceMix, Camel, CXF, and ActiveMQ are trademarks of Apache Software Foundation. Any other names contained herein may be trademarks of their respective owners.

Third Party Acknowledgements

One or more products in the Fuse MQ Enterprise 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 <mwpl@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. Introduction	13
2. Editing a Broker's Configuration	17
Understanding the Fuse MQ Enterprise Configuration Model	18
Editing a Standalone Broker's Configuration	22
Editing a Broker's Configuration in a Fabric	24
3. Security Basics	29
Fuse MQ Enterprise Security Overview	30
Basic Security Configuration	32
Disabling Broker Security	34
4. Setting up and Accessing the Fuse MQ Enterprise Web Console	35
Using the Embedded Console	36
Securing the Web Console	38
Deploying a Standalone Console	41
5. Installing Fuse MQ Enterprise as a Service	43
Configuring the Wrapper	44
Installing and Starting the Service	48
6. Starting a Broker	51
7. Securing the Fuse MQ Enterprise Container	55
Defining JAAS Realms	56
Enabling LDAP Authentication	61
Using Encrypted Property Placeholders	65
Configuring Roles for the Administrative Protocols	69
8. Sending Commands to the Broker	71
9. Deploying a New Broker	75
Deploying a Standalone Broker	76
Deploying a New Broker into a Fabric	77
10. Shutting Down a Broker	81
Shutting Down a Local Broker	82
Shutting Down a Broker Remotely	83
11. Connecting a Broker to a Fabric	87
Joining a Broker to a Fabric	88
Joining a Fabric as a Managed Container	89
Joining a Fabric as a Non-Managed Container	93
Creating a New Fabric	95
12. Using Logging	99
Logging Configuration	100
Viewing the Log	103
13. Using JMX	105
Configuring JMX	106
Statistics Collected by JMX	108
Managing the Broker with JMX	111

14. Applying Patches	117
Patching a Standalone Broker	118
Patching a Broker in a Fabric	120
A. Required JARS	123
Index	125

List of Figures

2.1. Fuse MQ Enterprise Configuration System	19
3.1. Ports Exposed by the Fuse MQ Enterprise Container	30

List of Tables

5.1. Wrapper Logging Properties	46
6.1. Start up Commands for Console Mode	51
6.2. Start up Commands for Daemon Mode	52
7.1. Flags for Defining a JAAS Module	57
7.2. Properties for the Fuse MQ Enterprise LDAP Login Module	61
8.1. Administration Client Arguments	71
13.1. Broker JMX Configuration Properties	106
13.2. Broker JMX Statistics	108
13.3. Destination JMX Statistics	109
13.4. Connection JMX Statistics	110
13.5. Broker MBean Operations	111
13.6. Connector MBean Operations	112
13.7. Network Connector MBean Operations	113
13.8. Queue MBean Operations	113
13.9. Topic MBean Operations	115
13.10. Subscription MBean Operations	115

List of Examples

2.1. Adding Property Placeholder Support to Fuse MQ Enterprise Configuration	20
2.2. Configuration with Property Placeholders	20
2.3. Importing an XML Configuration Template	25
2.4. Creating a Profile Using an XML Configuration Template	26
2.5. Creating a Deployment Profile	26
2.6. Setting Properties in a Profile	26
2.7. Setting the Broker Name Property	26
2.8. Assigning a Profile to a Broker	26
4.1. Changing the Web Console's Port	36
4.2. SSL Enabled Web Console Configuration	39
4.3. Disabling the Embedded Web Console	41
4.4. Configuration for Deploying the Web Console in Tomcat	42
4.5. Configuration for Monitoring a Cluster with the Web Console	42
5.1. Default Environment Settings	45
5.2. Default Java System Properties	45
5.3. Default Wrapper Classpath	45
5.4. Wrapper JMX Properties	46
6.1. Broker Console	52
6.2. Starting a Broker in a Fabric	52
6.3. Starting a Broker in a Fabric with the Administration Client	53
7.1. JAAS Blueprint Namespace	56
7.2. Defining a JAAS Realm in Blueprint XML	56
7.3. Standard JAAS Properties	58
7.4. Blueprint JAAS Properties	58
7.5. Configuring a JAAS Realm	59
7.6. Fuse ESB Enterprise LDAP JAAS Login Module	61
7.7. Configuring a JAAS Realm that Uses LDAP Authentication	64
7.8. Property File with an Encrypted Property	65
7.9. Encrypted Property Namespaces	66
7.10. Aries Placeholder Extension	66
7.11. Jasypt Blueprint Configuration	67
7.12. Jasypt Blueprint Configuration	67
7.13. Installing the Jasypt Feature	68
8.1. Client Command	71
8.2. Console Help	73
8.3. Help for a Command	73
9.1. Uploading a Template to a Fabric Ensemble	77
9.2. Creating a New Broker in an Existing Container	78
9.3. Creating a New Broker in a New Container	78
9.4. Editing a Broker Profile	78
10.1. Using the Console's Shutdown Command	82

10.2. Stop Command Syntax	83
10.3. Stopping a Remote Broker	84
10.4. Shutting DOWn a Broker using a Remote Console Connection	84
10.5. Shutting Down a Broker in a Fabric	85
12.1. Changing Logging Levels	101
12.2. Changing the Log Information Displayed on the Console	102
13.1. Configuring a Broker's JMX Connection	107
14.1. Adding a Patch to a Broker's Environment	118
14.2. Rolling Back a Patch	119

Chapter 1. Introduction

Once a messaging solution is deployed it needs to be monitored to ensure it performs at peak performance. When problems do arise, many of them can be solved using the broker's administrative tools. The broker's administrative tools can also be used to provide important debugging information when troubleshooting problems.

Overview

Message brokers are long lived and usually form the backbone of the applications of which they are a part. Over the course of a broker's life span, there are a number of management tasks that you may need to do to keep the broker running at peak performance. This includes monitoring the health of the broker, adding destinations, and security certificates.

If applications run into trouble one of the first places to look for clues is the broker. The broker is unlikely to be the root cause of the problem, but its logs and metrics will provide clues as to what is the root cause. You may also be able to resolve the problem using the broker's administrative interface.

Routine tasks

While Fuse MQ Enterprise is designed to require a light touch for management, there are a few routine management tasks that need to be performed:

- installing SSL certificates
  - starting the broker
  - creating destinations
  - stopping the broker
  - maintaining the advisory topics
  - monitoring the health of the broker
  - monitoring the health of the destinations
-

Troubleshooting

If an application runs into issues the broker will usually be able to provide clues to what is going wrong. Because the broker is central to the operation of any application that relies on messaging, it will be able to provide clues even if the broker is functioning properly. You may also be able to solve the problem by making adjustments to the broker's configuration.

Common things to check for clues as to the nature of a problem include:

- the broker's log file
- the advisory topics
- the broker's overall memory footprint
- the size of individual destination
- the total number of messages in the broker
- the size of the broker's persistent store
- a thread dump of the broker

One or more of these items can provide information about the problem. For example, if a destination grows to a very large size it could indicate that one of its consumers is having trouble keeping up with the messages. If the broker's log also shows that the consumer is repeatedly connecting and disconnecting from the destination, that could indicate a networking problem or a problem with the machine hosting the consumer.

Tools

There are a number of tools that you can use to monitor and administer Fuse MQ Enterprise.

The following tools are included with Fuse MQ Enterprise:

- [administration client on page 71](#)—a command line tool that can be used to manage a broker and do rudimentary metric reporting
- [console mode on page 72](#)—a runtime mode that presents you with a custom console that provides a number of administrative options
- [Web console on page 35](#)—a browser based console that provides metric reporting, destination browsing, and other administrative functions

Red Hat also provides management tools that you can install as part of your subscription:

- [Fuse Management Console](#)¹—a browser based console for viewing, monitoring, and deploying broker clusters

---

¹ <http://fusesource.com/docs/fmc>

- [Fuse HQ](#)²—an advanced monitoring and management tool that can provide detailed metrics and alerting.

In addition to the Red Hat supplied tools there are a number of third party tools that can be used to administer and monitor a broker including:

- jconsole—a JMX tool that is shipped with the JDK
- [VisualVM](#)³—a visual tool integrating several command line JDK tools and lightweight profiling capabilities

---

² <http://fusesource.com/docs/hq>

³ <http://visualvm.java.net/>

Chapter 2. Editing a Broker's Configuration

Fuse MQ Enterprise configuration uses a combination of an XML configuration template and OSGi PID configuration. This combination makes it possible to change specified broker properties on the fly. How you change the configuration depends on how the broker instance is deployed.

Understanding the Fuse MQ Enterprise Configuration Model	18
Editing a Standalone Broker's Configuration	22
Editing a Broker's Configuration in a Fabric	24

Configuring a broker involves making changes to a number of properties that are stored in multiple locations including:

- an XML configuration file
- OSGi persistent identifier properties

How you make the changes depends on how the broker is deployed:

- standalone—if a broker is deployed as a standalone entity and not a part of a fabric, you change the configuration using a combination of directly editing the broker's configuration template file and the console's **config** shell.
- in a fabric—if a broker is deployed into a fabric its configuration is managed by the Fabric Agent which draws all of the configuration from the fabric's registry. To modify the container of a broker running as part of a fabric, you need to modify the profile(s) deployed into it. You can do this by using either the **fabric:profile-edit** console command or Fuse Management Console.



Note

Many of the configuration properties are managed by the OSGi Admin Service and are organized by *persistent identifier* or PID. The container services look in a specific PID for particular properties, so it is important to set the properties in the correct PID.

Understanding the Fuse MQ Enterprise Configuration Model

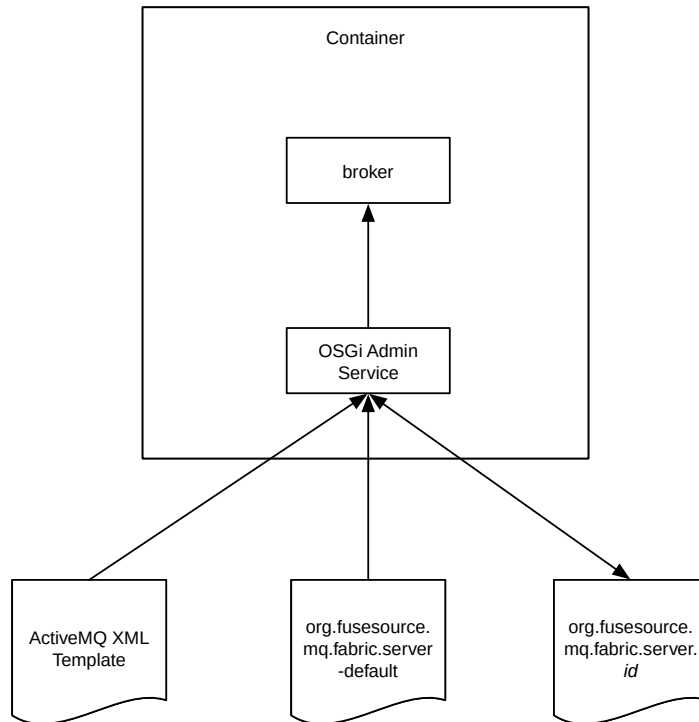
Overview

One of the weaknesses of the Apache ActiveMQ configuration model is that any changes require a broker restart. Fuse MQ Enterprise addresses this weakness by capitalizing on the OSGi Admin service. The container combines both the Apache ActiveMQ XML configuration and OSGi persistent identifier(PID) properties to manage a broker instances runtime configuration.

In Fuse MQ Enterprise your Apache ActiveMQ XML configuration file becomes a configuration template. It can contain property placeholders for any settings that may need to be set on the fly. It can also be used as a baseline for configuring a group of brokers and the placeholders represent settings that need to be modified for individual brokers.

As shown in [Figure 2.1 on page 19](#), the configuration template is combined with the OSGi PID properties. While the broker is running the OSGi Admin service monitors the appropriate PIDs for changes. When it detects a change, the admin service will automatically change the broker's runtime configuration.

Figure 2.1. Fuse MQ Enterprise Configuration System



Configuration templates

The Fuse MQ Enterprise configuration template is an XML file that is based on the [Apache ActiveMQ configuration file](http://fusesource.com/docs/mirrors/activemq/using-activemq-5/configure-version-5-brokers.html)¹. The main differences between an Apache ActiveMQ and a Fuse MQ Enterprise configuration template are:

- configuration templates use property placeholders for settings that will be controlled via the OSGi Admin service
- configuration templates do not configure the broker's name
- configuration templates do not configure the location of the data directory

¹ <http://fusesource.com/docs/mirrors/activemq/using-activemq-5/configure-version-5-brokers.html>

- configuration templates do not configure transport connectors
- configuration templates do not configure network connectors
- configuration templates do not control if a broker is a master or a slave node
- configuration templates can be used as a baseline for multiple brokers on the same machine

The networking properties and role in a master/slave group are specified by the broker's PID and do not need to appear in the template. The broker's name and data directory are replaced in the template with property placeholders. Property placeholders can also be substituted for any attribute value or element value in the XML configuration. This allows the OSGi Admin system populate them from the broker's PID.

Property placeholders are specified using the syntax `${propName}` and are resolved by matching properties in the broker's PID. In order to use property placeholder the configuration template must include the bean definition shown in [Example 2.1 on page 20](#).

Example 2.1. Adding Property Placeholder Support to Fuse MQ Enterprise Configuration

```
<!-- Allows us to use system properties and fabric as variables
in this configuration file -->
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="properties">
    <bean class="org.fusesource.mq.fabric.ConfigurationProperties"/>
  </property>
</bean>

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

The configuration template shown in [Example 2.2 on page 20](#) uses three property placeholders that allow you to modify the base configuration using fabric properties.

Example 2.2. Configuration with Property Placeholders

```
<broker xmlns="http://activemq.apache.org/schema/core"
  brokerName="${broker-name}"
```

```

        dataDirectory="${data}"
        persistent="${persists}"
        start="false">
...
<persistenceAdapter>
  <jdbcPersistenceAdapter dataDirectory="${data}/derby"
                        dataSource="#derby-ds" />
</persistenceAdapter>
</broker>

```

OSGi PIDs

Persistent identifiers are described in chapter 104, [Configuration Admin Service Specification], of the [OSGi Compendium Services Specification]. It is a unique key used by the OSGi framework's admin service to associate configuration properties with active services. The PIDs for a Fuse MQ Enterprise instance have the prefix `org.fusesource.mq.fabric.server.`

Every PID has a physical representation as a file of name value pairs. For standalone brokers the files are located in the `etc/` folder and use the `.cfg` extension and are updated using the **config** shell. For broker's in a fabric the files are stored in the Fabric Ensemble and are edited using the **fabric** shell's profile management commands.

Editing a Standalone Broker's Configuration

Overview

A standalone broker is one that is not part of a fabric. A standalone broker can, however, be part of a network of broker, a master/slave cluster, or a failover cluster. The distinction is that a standalone is responsible for managing and storing its own configuration.

All of the configuration changes are made directly on the local instance. You make changes using a combination of edits to local configuration template and commands from the console's **config** shell. The configuration template must be edited using an external editor. The configuration the control's the behavior of the broker's runtime container is changed using the console commands.

Editing the configuration template

The default broker [configuration template](#) is `etc/activemq.xml`. You can the location of the configuration template by changing the `config` property in the broker's `etc/org.fusesource.mq.fabric.server-default.cfg` file.

The template can be edited using any text or XML editor.

The broker must be restarted for any changes in the template to take effect.

Editing the OSGi properties

The initial values for all of the OSGi properties configuring the broker are specified in the `etc/org.fusesource.mq.fabric.server-default.cfg` file. You can edit these values using the command console's **config** shell. The PID for these values are `org.fusesource.mq.fabric.server.id`. The `id` is assigned by the container when the broker is started.

In addition to the broker's messaging behavior, a number of the broker's runtime behavior such as logging levels, the Web console behavior, and the JMX behavior are controlled by by OSGi properties stored in different PIDs.

To find the value for a broker's `id` use and the PIDs for the other runtime configuration settings, use the **config:list** command.

Config shell

The **config** shell has a series of commands for editing OSGi properties:

- **config:list**—lists all of the runtime configuration files and the current values for their properties

- **config:edit**—opens an editing session for a configuration file
- **config:propset**—changes the value of a configuration property
- **config:propdel**—deletes a configuration property
- **config:update**—saves the changes to the configuration file being edited

For more information about the **config** shell commands, see ["Config Console Commands"](#) in *Console Reference*.

Editing a Broker's Configuration in a Fabric

Overview

When a broker is part of a fabric, it does not manage its configuration. The broker's configuration is managed by the Fabric Agent. The agent runs along with the broker and updates the broker's configuration based on information from the fabric's registry.

Because the configuration is managed by the Fabric Agent, any changes to the broker's configuration needs to be done by updating the fabric's registry. In a fabric, broker configuration is determined by one or more profiles that are deployed into the broker. To change a broker's configuration, you must update the profile(s) deployed into the broker using either the console's **fabric:** shell or Fuse Management Console.

Profiles

All configuration in a fabric is stored as *profiles* in the Fabric Registry. One or more profiles are assigned to brokers that are part of the fabric. A profile is a collection of configuration that specifies:

- the Apache Karaf features to be deployed
- OSGi bundles to be deployed
- the repositories from which artifacts can be provisioned
- properties that configure the broker's runtime behavior

The configuration profiles are collected into versions. Versions are typically used to make updates to an existing profile without effecting deployed brokers. When a container is configured it is assigned a profile version from which it draws the profiles. Therefore, when you create a new version and edit the profiles in the new version, the profiles that are in use are not changed. When you are ready to test the changes, you can roll them out incrementally by moving brokers to a new version one at a time.

When a broker joins a fabric, a Fabric Agent is deployed with the broker and takes control of the broker's configuration. The agent will ask the Fabric Registry what version and profile(s) are assigned to the broker and configure the broker based on the profiles. The agent will download and install of the

specified bundles and features. It will also set all of the specified configuration properties.

Procedure

The recommended approach to configuring brokers in a fabric is:

1. Create a [configuration template](#).
2. Create a base profile for all of the brokers in the fabric using the configuration template.
3. Create profiles that inherit from the base profile that will be assigned to one or more brokers.
4. Modify the properties in each of the profiles to the desired values for the brokers to which the profile will be assigned.
5. Assign the new profiles to the desired brokers.

You should always create new profiles or a new version of the existing profiles before making configuration changes. Changes to profiles that are assigned to running brokers take effect immediately. Using new profiles, or a new version, allows you make the changes and test them on a subset of your brokers before rolling the changes to the entire fabric.

Creating a base profile

To create a base profile:

1. Optionally create a new profile version using the **fabric:version-create** command.

This will create a new copy of the existing profiles. See [fabric:version-create](#) in *Console Reference* for more information.

2. Import the new XML template into the registry using the **fabric:import** command as shown in [Example 2.4 on page 26](#).

Example 2.3. Importing an XML Configuration Template

```
FuseMQ:karaf@root> fabric:import -t /fabric/configs/versions/version/profiles/mq-base/xmlTemplate.xml
TemplatePath
```

See [fabric:import](#) in *Console Reference* for more information.

3. Create a new configuration profile instance to hold the new XML template using the **fabric:mq-create** command as shown in [Example 2.4 on page 26](#).

Example 2.4. Creating a Profile Using an XML Configuration Template

```
FuseMQ:karaf@root> fabric:mq-create --config xmlTemplate profileName
```

This will create a new profile that is based on the default broker profile but uses the imported XML template. See [fabric:mq-create](#) in *Console Reference* for more information.

Creating deployment profiles and assigning them to brokers

To create deployment profiles and assigned them to the brokers:

1. Create new profile using the **fabric:profile-create** command as shown in [Example 2.5 on page 26](#).

Example 2.5. Creating a Deployment Profile

```
FuseMQ:karaf@root> fabric:profile-create --parents baseProfile profileName
```

This will create a new profile that inherits from the base profile. See [fabric:profile-create](#) in *Console Reference* for more information.

2. Add values for the property placeholders using the **fabric:profile-edit** command as shown in [Example 2.6 on page 26](#).

Example 2.6. Setting Properties in a Profile

```
FuseMQ:karaf@root> fabric:profile-edit -p org.fusesource.mq.fabric.server-profileName/propName=propVal  
profileName
```

The fabric properties for a broker are specified using the PID `org.fusesource.mq.fabric.server-profileName`, so to specify a value for the `broker-name` property for the profile called `myBroker` you would use the command shown in [Example 2.7 on page 26](#).

Example 2.7. Setting the Broker Name Property

```
FuseMQ:karaf@root> fabric:profile-edit -p org.fusesource.mq.fabric.server-myBroker/broker-name=esmeralda  
myBroker
```

3. Assign the new profile to one or more brokers using the **fabric:container-add-profile** command as shown in [Example 2.8 on page 26](#).

Example 2.8. Assigning a Profile to a Broker

```
FuseMQ:karaf@root> fabric:container-add-profile brokerName profileName
```

See [fabric:container-add-profile](#), in *Console Reference* for more information.

Using Fuse Management Console

Fuse Management Console simplifies the process of configuring brokers in a fabric by providing an easy to use Web-based interface and reducing the number of steps required to make the changes. For more information on using Fuse Management Console see the [Fuse Management Console Documentation](#)².

---

² <http://fusesource.com/documentation/fuse-management-console-documentation>

Chapter 3. Security Basics

By default, Fuse MQ Enterprise is secure because none of its ports are remotely accessible. You want to open a few basic ports for remote access for management purposes.

Fuse MQ Enterprise Security Overview	30
Basic Security Configuration	32
Disabling Broker Security	34

Fuse MQ Enterprise Security Overview

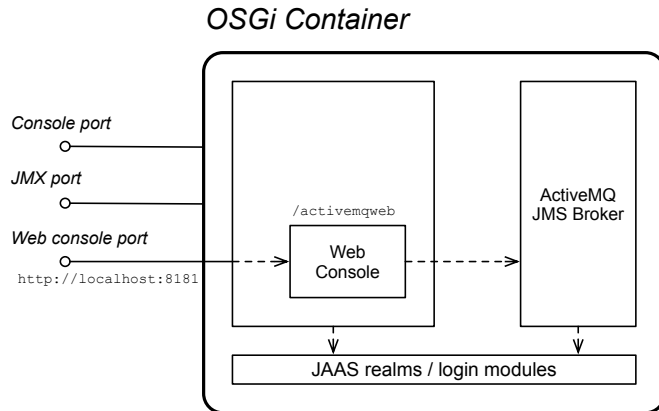
Overview

The Fuse MQ Enterprise runtime exposes three ports for remote access. These ports, which are mostly intended for managing the broker, are essentially disabled by default. They are configured to require authentication, but have no defined users. This makes the broker immune to breaches, but is not ideal for remote management.

Ports exposed by the Fuse MQ Enterprise container

Figure 3.1 on page 30 shows the ports exposed by the Fuse MQ Enterprise container by default.

Figure 3.1. Ports Exposed by the Fuse MQ Enterprise Container



The following ports are exposed by the container:

- *Console port*—enables remote control of a container instance, through Apache Karaf shell commands. This port is enabled by default and is secured both by JAAS authentication and by SSL.
- *JMX port*—enables management of the container through the JMX protocol. This port is enabled by default and is secured by JAAS authentication.

- *Web console port*—provides access to an embedded Jetty container that can host Web console servlet.
-

Authentication and authorization system

Fuse MQ Enterprise uses Java Authentication and Authorization Service (JAAS) for ensuring the users trying to access the broker have the proper credentials. The implementation is modular, with individual JAAS modules providing the authentication implementations. Fuse MQ Enterprise's command console provides commands to configure the JAAS system.

Basic Security Configuration

Overview

The default security settings block access to a broker's remote ports. If you want to access the Fuse MQ Enterprise runtime remotely, you must first customize the security configuration. The first thing you will want to do is create at least one JAAS user. This will enable remote access to the broker.

Other common configuration changes you may want to make are:

- configure access to the Web console
- assign roles to each of the remote ports to limit access
- strengthen the credentials needed to access the remote console

---

Create a secure JAAS user

By default, no JAAS users are defined for the container, which effectively disables remote access (it is impossible to log on).

To create a secure JAAS user, edit the `InstallDir/etc/users.properties` file and add a new user field, as follows:

```
Username=Password,admin
```

Where `Username` and `Password` are the new user credentials. The `admin` role gives this user the privileges to access all administration and management functions of the container. For more details about JAAS, see "[JAAS Authentication](#)" in *Security Guide*.



Warning

It is strongly recommended that you define custom user credentials with a strong password.

Configure the Web console

If you want to access the Web console (for remote administration of JMS messaging), you must provide the Web console servlet with the credentials it needs to login to the JMS broker.

Edit the `InstallDir/etc/system.properties` file and modify the `webconsole.jmx.*` and `webconsole.jms.*` properties as follows:


```
webconsole.jmx.user=Username
webconsole.jmx.password=Password
webconsole.jms.user=Username
webconsole.jms.password=Password
```

Where the *Username* and *Password* are the credentials of a JAAS user with admin privileges.

For more details on securing the Web console see ["Securing the Web Console" on page 38](#).

Assigning roles for remote access

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

- SSH (remote console login)

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

```
sshRole=admin
```

- JMX management

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

```
jmxRole=jmx
```

- Web console

Strengthening security on the remote console port

You can employ the following measures to strengthen security on the remote console port:

- Make sure that the JAAS user credentials have strong passwords.
- Customize the X.509 certificate (replace the Java keystore file, `InstallDir/etc/host.key`, with a custom key pair).

For more details, see ["Securing the Fuse MQ Enterprise Container" on page 55](#).

Disabling Broker Security

Overview

Prior to Fuse MQ Enterprise version 7.0.2, the Apache ActiveMQ broker was insecure (JAAS authentication not enabled). This section explains how to revert the Apache ActiveMQ broker to an insecure mode of operation, so that it is unnecessary to provide credentials when connecting to the broker.



Warning

After performing the steps outlined in this section, the broker has no protection against hostile clients. This type of configuration is suitable only for use on internal, trusted networks.

Standalone server

These instructions assume that you are running Fuse MQ Enterprise in standalone mode (that is, running in an OSGi container, but not using Fuse Fabric). In your installation of Fuse MQ Enterprise, open the *InstallDir/etc/activemq.xml* file using a text editor and look for the following lines:

```
...
<plugins>
  <jaasAuthenticationPlugin configuration="karaf" />
</plugins>
...
```

To disable JAAS authentication, delete (or comment out) the `jaasAuthenticationPlugin` element. The next time you start up the Fuse MQ Enterprise container (using the *InstallDir/bin/fusemq* script), the broker will run with unsecured ports.

Chapter 4. Setting up and Accessing the Fuse MQ Enterprise Web Console

Fuse MQ Enterprise's Web console is an easy way to query a standalone broker for JMX statistics, destination statuses, and information on any routes deployed in the broker.

Using the Embedded Console	36
Securing the Web Console	38
Deploying a Standalone Console	41

The Fuse MQ Enterprise Web console is an embedded console for administering standalone broker. Using the embedded console is an easy way to manage your broker with minimal configuration.



Warning

Because the default configuration profile of a broker deployed into a Fabric container does not include the Web console, you must use Fuse Management Console to monitor fabric-deployed brokers, unless you install the `a-mq` profile in the Fabric container when you create the fabric. To do so, use the command `fabric:create -p a-mq`.

If you want more security, more reliability, or the ability to monitor master/slave clusters, you can deploy the Web console as a standalone application. It is easy to deploy into Tomcat or any standard Web container.

Using the Embedded Console

Overview

For a standalone broker the default configuration is for the Web console to be loaded along with the broker. The default address for the standalone broker's console is `localhost:8181/activemqweb`.



Warning

Because the default configuration profile of a broker deployed into a Fabric container does not include the Web console, you must use Fuse Management Console to monitor fabric-deployed brokers, unless you install the `a-mq` profile in the Fabric container when you create the fabric. To do so, use the command `fabric:create -p a-mq`.

The default port and security configuration can easily be modified.

Accessing the console

The Fuse MQ Enterprise Web console's address is

`http://hostName:portNum/activemqweb`.

For example, to access the default broker administration console on your local machine, you would point your Web browser at

`http://localhost:8181/activemqweb`.

Changing the port

The port number at which the Web console is accessed is controlled by the broker's Pax Web server. To change the port, you need to edit the `org.osgi.service.http.port` property in the `org.ops4j.pax.web` OSGi Admin PID.

[Example 4.1 on page 36](#) shows the console commands used to change the port number of a standalone broker's Web console to 8536.

Example 4.1. Changing the Web Console's Port

```
FuseMQ:karaf@root> config:edit org.ops4j.pax.web
FuseMQ:karaf@root> config:proplist
    service.pid = org.ops4j.pax.web
    javax.servlet.context.tempdir = /Applications/FuseMQEnterprise-7.0.0/fuse-mq-7.0.0.fuse-beta-
042/data/pax-web-jsp
    org.osgi.service.http.port = 8181
    org.ops4j.pax.web.config.file = /Applications/FuseMQEnterprise-7.0.0/fuse-mq-7.0.0.fuse-beta-
042/etc/jetty.xml
    org.apache.karaf.features.configKey = org.ops4j.pax.web
```

```
FuseMQ:karaf@root> config:propset org.osgi.service.http.port 8536
FuseMQ:karaf@root> config:update
```

Securing the console

The security for the Web console is provided by the Web container in which it is deployed. For the embedded instance of the Web console, you need to configure the embedded Jetty container's security by editing `conf/jetty.xml`.

See ["Securing the Web Console" on page 38](#) for details.

Securing the Web Console

Overview

By default the Fuse MQ Enterprise Web console is insecure. It does not require authentication and uses the standard HTTP transport. For most commercial deployments, it is advisable that the Web console is secured. Adding security is done by editing the `etc/jetty.xml` configuration file.

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`. The result should be similar to:

```
<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:

```
jblogs: secret, developer
```

Enabling SSL security

To enable SSL security on the Jetty server, edit the `Connector` bean in the `conf/jetty.xml` file. Replace the `org.eclipse.jetty.server.nio.SelectChannelConnector` class with 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 as shown in [Example 4.2 on page 39](#).

Example 4.2. SSL Enabled Web Console Configuration

```
<property name="connectors">
  <list>
    <bean id="Connector" class="org.eclipse.jetty.serv
er.ssl.SslSelectChannelConnector">
      <property name="port" value="8443" />
      <property name="maxIdleTime" value="30000"/>
      <property name="keystore" value="{act
ivemq.home}/conf/broker.ks"/>
      <property name="password" value="testjetty"/>
      <property name="keyPassword" value="testjetty"/>
      <property name="truststore" value="{act
ivemq.home}/conf/broker.ks"/>
    </bean>
  </list>
</property>
```

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 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 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"](#) in *Security Guide*.
- `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.



Note

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 URL

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



Warning

The `broker.ks` certificate used in the example is insecure. 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](#)" in *Security Guide*.

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>.

Deploying a Standalone Console

Overview

For more security or reliability reasons you can deploy the Web console as a standalone application in Tomcat or another Web container. When running as a standalone application, the Web console can be set up to monitor Master/Slave clusters.

Disabling the embedded console

There is no need to disable the embedded console when using a standalone console. However, If you are using a standalone Web console, there is no reason to use the resources required by the embedded console. Nor is there a reason to leave an extra administrative access point open.

To disable the embedded Web console, you simply need to comment out, or remove, the `import` element that imports the Jetty configuration into your broker's configuration file as shown in [Example 4.3 on page 41](#).

Example 4.3. Disabling the Embedded Web Console

```
<beans ... >

  <broker ... >
    ...
  </broker>

  <!-- <import resource="jetty.xml"/> -->

</beans>
```

Configuring Tomcat

To deploy the Web console in Tomcat 5.x:

1. Download the Web console's WAR, `activemq-web-console-7.1.0.fuse-047.war`, from <http://repo.fusesource.com/nexus/content/repositories/releases/org/apache/activemq/activemq-web-console/5.7.0.fuse-71-047/>.
2. Copy the Web console's WAR to the `TOMCAT_HOME/webapps` folder.
3. Download `activemq-all-7.1.0.fuse-047.jar` from <http://repo.fusesource.com/nexus/content/repositories/releases/org/apache/activemq/activemq-all/5.7.0.fuse-71-047/>.

4. Copy `activemq-all-7.1.0.fuse-047.jar` to the `TOMCAT_HOME/common/lib` folder.
5. Modify `TOMCAT_HOME/bin/catalina.sh(.bat)` to include the configuration in [Example 4.4 on page 42](#).

Example 4.4. Configuration for Deploying the Web Console in Tomcat

```
JAVA_OPTS="-Dwebconsole.type=properties \  
-Dwebconsole.jms.url=brokerURL \  
-Dwebconsole.jmx.url=brokerJMXURL \  
-Dwebconsole.jmx.user=JMXUserName \  
-Dwebconsole.jmx.password=JMXPassword"
```

6. Restart Tomcat.

The Web console will be available at

`tomcatURI/activemq-web-console-5.7.0.fuse-71-047`.

Monitoring clusters

It's possible to configure the Web console to monitor a master/slave cluster. To do so:

- Specify the JMS URL, `webconsole.jms.url`, with a failover: URI specifying the brokers in the cluster.
- Specify the JMX URL, `webconsole.jmx.url` as a comma separated list that contains the JMX URL for each of the brokers in the cluster.

[Example 4.5 on page 42](#) shows the properties for monitoring a cluster using the Web console.

Example 4.5. Configuration for Monitoring a Cluster with the Web Console

```
-Dwebconsole.jms.url=failover:(tcp://serverA:61616,tcp://serverB:61616)  
-Dwebconsole.jmx.url=service:jmx:rmi:///jndi/rmi://serverA:1099/jmxrmi,ser  
vice:jmx:rmi:///jndi/rmi://serverB:1099/jmxrmi
```

For more information about master/slave clusters see ["Master/Slave"](#) in *Fault Tolerant Messaging*.

Chapter 5. Installing Fuse MQ Enterprise as a Service

The Fuse MQ Enterprise installer generates a service wrapper that can be easily configured to install Fuse MQ Enterprise as a system service.

Configuring the Wrapper	44
Installing and Starting the Service	48

Installing Fuse MQ Enterprise as a system service is a two step process:

1. [Configure on page 44](#) the service wrapper for your system.
2. [Install on page 48](#) the service wrapper as system service.

Configuring the Wrapper

Overview

The service wrapper is configured by the `ServiceName-wrapper.conf` file, which is located under the `InstallDir/etc/` directory.

There are several settings you may want to change including:

- the default environment settings
- the properties passed to the JVM
- the classpath
- the JMX settings
- the logging settings

Specifying the Fuse MQ Enterprise's environment

A broker's environment is controlled by three environment variables:

- `KARAF_HOME`—the location of the Fuse MQ Enterprise install directory.
- `KARAF_BASE`—the root directory containing the configuration and OSGi data specific to the broker instance.

The configuration for the broker instance is stored in the `KARAF_BASE/conf` directory. Other data relating to the OSGi runtime is also stored beneath the base directory.

- `KARAF_DATA`—the directory containing the logging and persistence data for the broker.

[Example 5.1 on page 45](#) shows the default values.

Example 5.1. Default Environment Settings

```
set.default.KARAF_HOME=InstallDir
set.default.KARAF_BASE=InstallDir
set.default.KARAF_DATA=InstallDir\data
```

Passing parameters to the JVM

If you want to pass parameters to the JVM, you do so by setting wrapper properties using the form `wrapper.java.additional.<n>`. `<n>` is a sequence number that must be distinct for each parameter.

One of the most useful things you can do by passing additional parameters to the JVM is to set Java system properties. The syntax for setting a Java system property is `wrapper.java.additional.<n>=-DPropName=PropValue`.

[Example 5.2 on page 45](#) shows the default Java properties.

Example 5.2. Default Java System Properties

```
# JVM
# note that n is the parameter number starting from 1.
wrapper.java.additional.1=-Dkaraf.home="%KARAF_HOME%"
wrapper.java.additional.2=-Dkaraf.base="%KARAF_BASE%"
wrapper.java.additional.3=-Dkaraf.data="%KARAF_DATA%"
wrapper.java.additional.4=-Dcom.sun.management.jmxremote
wrapper.java.additional.5=-Dkaraf.startLocalConsole=false
wrapper.java.additional.6=-Dkaraf.startRemoteShell=true
wrapper.java.additional.7=-Djava.endorsed.dirs="%JAVA_HOME%/jre/lib/en
dorsed;%JAVA_HOME%/lib/endorsed;%KARAF_HOME%/lib/endorsed"
wrapper.java.additional.8=-
Djava.ext.dirs="%JAVA_HOME%/jre/lib/ext;%JAVA_HOME%/lib/ext;%KARAF_HOME%/lib/ext"
```

Adding classpath entries

You add classpath entries using the syntax `wrapper.java.classpath.<n>`. `<n>` is a sequence number that must be distinct for each classpath entry.

[Example 5.3 on page 45](#) shows the default classpath entries.

Example 5.3. Default Wrapper Classpath

```
wrapper.java.classpath.1=%KARAF_BASE%/lib/karaf-wrapper.jar
wrapper.java.classpath.2=%KARAF_HOME%/lib/karaf.jar
```

```
wrapper.java.classpath.3=%KARAF_HOME%/lib/karaf-jaas-boot.jar
wrapper.java.classpath.4=%KARAF_BASE%/lib/karaf-wrapper-main.jar
```

JMX configuration

The default service wrapper configuration does not enable JMX. It does, however, include template properties for enabling JMX. To enable JMX:

1. Locate the line `# Uncomment to enable jmx.`

There are three properties, shown in [Example 5.4 on page 46](#), that are used to configure JMX.

Example 5.4. Wrapper JMX Properties

```
# Uncomment to enable jmx
#wrapper.java.additional.n=-Dcom.sun.management.jmxre
mote.port=1616
#wrapper.java.additional.n=-Dcom.sun.management.jmxre
mote.authenticate=false
#wrapper.java.additional.n=-Dcom.sun.management.jmxre
mote.ssl=false
```

2. Remove the `#` from in front of each of the properties.
3. Replace the `n` in each property to a number that fits into the sequence of addition properties established in the configuration.

You can change the settings to use a different port or secure the JMX connection.

For more information about using JMX see ["Using JMX" on page 105](#).

Configuring logging

The wrapper's logging is configured using the properties described in [Table 5.1 on page 46](#).

Table 5.1. Wrapper Logging Properties

Property	Description
<code>wrapper.console.format</code>	Specifies how the logging information sent to the console is formatted. The format consists of the following tokens: • <code>L</code>—log level

Property	Description
	• <code>P</code>—prefix • <code>D</code>—thread name • <code>T</code>—time • <code>Z</code>—time in milliseconds • <code>U</code>—approximate uptime in seconds (based on internal tick counter) • <code>M</code>—message
<code>wrapper.console.loglevel</code>	Specifies the logging level displayed on the console.
<code>wrapper.logfile</code>	Specifies the file used to store the log.
<code>wrapper.logfile.format</code>	Specifies how the logging information sent to the log file is formatted.
<code>wrapper.console.loglevel</code>	Specifies the logging level sent to the log file.
<code>wrapper.console.maxsize</code>	Specifies the maximum size, in bytes, that the log file can grow to before the log is archived. The default value of 0 disables log rolling.
<code>wrapper.console.maxfiles</code>	Specifies the maximum number of archived log files which will be allowed before old files are deleted. The default value of 0 implies no limit.
<code>wrapper.syslog.loglevel</code>	Specifies the logging level for the sys/event log output.

For more information about Fuse MQ Enterprise logging see ["Using Logging" on page 99](#).

Installing and Starting the Service

Overview

The operating system determines the exact steps using to complete the installation of Fuse MQ Enterprise as a service. The **wrapper:install** command provides basic instructions for your operating system.

Windows

To install the service run `InstallDir\bin\ServiceName-service.bat install`. If you used the default start setting, the service will start when Windows is launched. If you specified `DEMAND_START`, you will need to start the service manually.

To start the service manually run `net start "ServiceName"`. You can also use the Windows service UI.

To manually stop the service run `net stop "ServiceName"` You can also use the Windows service UI.

You remove the installed the service by running `InstallDir\bin\ServiceName-service.bat remove`.

Redhat Linux

To install the service and configure it to start when the machine boots, run the following commands:

```
# ln -s InstallDir\bin\ServiceName-service /etc/init.d/  
# chkconfig ServiceName-service --add  
# chkconfig ServiceName-service on
```

To start the service manually run `service ServiceName-service start`.

To manually stop the service run `service ServiceName-service stop`.

You remove the installed the service by running the following commands:

```
#service ServiceName-service stop  
# chkconfig ServiceName-service --del  
# rm /etc/init.d/ServiceName-service
```

Ubuntu Linux

To install the service and configure it to start when the machine boots, run the following commands:

```
# ln -s InstallDir\bin\ServiceName-service /etc/init.d/  
# update-rc.d ServiceName-service defaults
```


To start the service manually run `/etc/init.d/ServiceName-service start`.

To manually stop the service run `/etc/init.d/ServiceName-service stop`.

You remove the installed the service by running the following commands:

```
#/etc/init.d/ServiceName-service stop
# rm /etc/init.d/ServiceName-service
```


Chapter 6. Starting a Broker

You start a broker using a simple command. The broker can either be started so that it launches a command console or so that it runs as a daemon process. When a broker is part of a fabric, you can remotely start the broker remotely.

Overview

A broker can be run in one of two modes:

- console mode—the broker starts up as a foreground process and presents the user with a command shell
- daemon mode—the broker starts up as a background process that can be managed using a remote console or the provided command line tools

The default location for the broker's configuration for the broker is the `InstallDir/etc/activemq.xml` configuration file. The configuration uses values loaded from the `InstallDir/etc/system.properties` file and the `InstallDir/etc/org.fusesource.mq.fabric.server-default.cfg` file.

Starting in console mode

When you start the broker in console mode you will be placed into a command shell that provides access to a number of commands for managing the broker and its OSGi runtime.



Important

When the broker is started in console mode, you cannot close the console without killing the broker.

To launch a broker in console mode, change to `InstallDir` and run one of the commands in [Table 6.1 on page 51](#).

Table 6.1. Start up Commands for Console Mode

Windows	<code>bin\fusemq.bat</code>
Unix	<code>bin/fusemq</code>

If the server starts up correctly you should see something similar to [Example 6.1 on page 52](#) on the console.

Example 6.1. Broker Console

[illegible]

Starting in daemon mode

Launching a broker in daemon mode runs Fuse MQ Enterprise in the background without a console. To launch a broker in daemon mode, change to `InstallDir` and run one of the commands in [Table 6.2 on page 52](#).

Table 6.2. Start up Commands for Daemon Mode

Windows	bin\start.bat
Unix	bin/start

Starting a broker in a fabric

If a broker is deployed as part of a fabric you can start it remotely in one of three ways:

- using the console of one of the other broker's in the fabric

If one of the brokers in the fabric is running in console mode you can use the **fabric:container-start** command to start any of the other brokers in the fabric. The command requires that you supply the container name used when creating the broker in the fabric. For example to start a broker named `fabric-broker3` you would use the command shown in [Example 6.2 on page 52](#).

Example 6.2. Starting a Broker in a Fabric

```
FuseMQ:karaf@root> fabric:container-start fabric-broker3
```

For more information see [fabric:container-start](#), in *Console Reference*.

- using the administration client of one of the broker's in the fabric

If none of the brokers are running in console mode, you can use the administration client on one of the brokers to execute the **fabric:container-start** command. The administration client is run using the **client** command in Fuse MQ Enterprise's `bin` folder.

[Example 6.3 on page 53](#) shows how to use the remote client to start remote broker in the fabric.

Example 6.3. Starting a Broker in a Fabric with the Administration Client

```
bin/client fabric:container-start fabric-broker3
```

- using Fuse Management Console

Fuse Management Console can start and stop any of the brokers in the fabric it manages from a Web based console.

For more information see the [Fuse Management Console Documentation](#)¹.

---

¹ <http://fusesource.com/docs/fmc>

Chapter 7. Securing the Fuse MQ Enterprise Container

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

Defining JAAS Realms	56
Enabling LDAP Authentication	61
Using Encrypted Property Placeholders	65
Configuring Roles for the Administrative Protocols	69

Defining JAAS Realms

Overview

When defining a JAAS realm in the OSGi container, you *cannot* put the definitions in a conventional JAAS [login configuration](#)¹ file. Instead, the OSGi container uses a special `jaas:config` element for defining JAAS realms in a blueprint configuration file. The JAAS realms defined in this way are made available to *all* of the application bundles deployed in the container, making it possible to share the JAAS security infrastructure across the whole container.

Namespace

The `jaas:config` element is defined in the `http://karaf.apache.org/xmlns/jaas/v1.0.0` namespace. When defining a JAAS realm you will need to include the line shown in [Example 7.1 on page 56](#).

Example 7.1. JAAS Blueprint Namespace

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

Configuring a JAAS realm

The syntax for the `jaas:config` element is shown in [Example 7.2 on page 56](#).

Example 7.2. Defining a JAAS Realm in Blueprint XML

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

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

</blueprint>
```

The elements are used as follows:

¹ <http://download.oracle.com/javase/6/docs/technotes/guides/security/jaas/JAASRefGuide.html#AppendixB>


```
jaas:config
```

Defines the JAAS realm. It has the following attributes:

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

```
jaas:module
```

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

- **className**—the fully-qualified class name of a JAAS login module. The specified class must be available from the bundle classloader.
- **flags**—determines what happens upon success or failure of the login operation. [Table 7.1 on page 57](#) describes the valid values.

Table 7.1. Flags for Defining a JAAS Module

Value	Description
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.

Value	Description
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.

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



Note

You can define multiple login modules in a realm.

Converting standard JAAS login properties to XML

Fuse ESB Enterprise uses the same properties as a standard Java login configuration file, however Fuse ESB Enterprise requires that they are specified slightly differently. To see how the Fuse ESB Enterprise approach to defining JAAS realms compares with the standard Java login configuration file approach, consider how to convert the login configuration shown in [Example 7.3 on page 58](#), which defines the `PropertiesLogin` realm using the Fuse MQ Enterprise properties login module class, `PropertiesLoginModule`:

Example 7.3. Standard JAAS Properties

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

The equivalent JAAS realm definition, using the `jaas:config` element in a blueprint file, is shown in [Example 7.4 on page 58](#).

Example 7.4. Blueprint JAAS Properties

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

    <jaas:config name="PropertiesLogin">
```

```

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

</blueprint>

```



Important

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

Example

Fuse ESB Enterprise also provides an adapter that enables you to store JAAS authentication data in an X.500 server. [Example 7.5 on page 59](#) defines the LDAPLogin realm to use Fuse ESB Enterprise's LDAPLoginModule class, which connects to the LDAP server located at ldap://localhost:10389.

Example 7.5. Configuring a JAAS Realm

```

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

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

```

```
</jaas:config>  
</blueprint>
```

For a detailed description and example of using the LDAP login module, see ["Enabling LDAP Authentication" on page 61](#).

Enabling LDAP Authentication

Overview

Fuse MQ Enterprise supplies a JAAS login module that enables it to use LDAP to authenticate users. The Fuse MQ Enterprise JAAS LDAP login module is implemented by the

`org.apache.karaf.jaas.modules.ldap.LDAPLoginModule` class. It is preloaded in the container, so you do not need to install its bundle.

Procedure

To enable Fuse MQ Enterprise to use LDAP for user authentication you need to create a JAAS realm that includes the Fuse MQ Enterprise LDAP login module. As shown in [Example 7.6 on page 61](#), this is done by adding a `jaas:module` element to the realm and setting its `className` attribute to `org.apache.karaf.jaas.modules.ldap.LDAPLoginModule`.

Example 7.6. Fuse ESB Enterprise LDAP JAAS Login Module

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

You will also need to provide values for the properties described in [Table 7.2 on page 61](#).

LDAP properties

[Table 7.2 on page 61](#) describes the properties used to configure the Fuse MQ Enterprise JAAS LDAP login module.

Table 7.2. Properties for the Fuse MQ Enterprise LDAP Login Module

Property	Description
<code>connection.url</code>	Specifies specify the location of the directory server using an ldap URL, <code>ldap://Host:Port</code> . You can optionally qualify this URL, by adding a forward slash, <code>/</code> , followed by the DN of a particular node in the directory tree.

Property	Description
<code>connection.username</code>	Specifies the DN of the user that opens the connection to the directory server. For example, <code>uid=admin,ou=system</code> .
<code>connection.password</code>	Specifies the password that matches the DN from <code>connection.username</code> . In the directory server, the password is normally stored as a <code>userPassword</code> attribute in the corresponding directory entry.
<code>user.base.dn</code>	Specifies the DN of the subtree of the DIT to search for user entries.
<code>user.filter</code>	Specifies the LDAP search filter used to locate user credentials. It is applied to the subtree selected by <code>user.base.dn</code> . Before being passed to the LDAP search operation, the value is subjected to string substitution such that all occurrences of <code>%u</code> are replaced by the user name extracted from the incoming credentials.
<code>user.search.subtree</code>	Specifies if the user entry search's scope includes the subtrees of the tree selected by <code>user.base.dn</code> .
<code>role.base.dn</code>	Specifies the DN of the subtree of the DIT to search for role entries.
<code>role.filter</code>	Specifies the LDAP search filter used to locate roles. It is applied to the subtree selected by <code>role.base.dn</code> . Before being passed to the LDAP search operation, the value is subjected to string substitution such that all occurrences of <code>%u</code> are replaced by the user name extracted from the incoming credentials.

Property	Description
<code>role.name.attribute</code>	Specifies the attribute type of the role entry that contains the name of the role/group. If you omit this option, the role search feature is effectively disabled.
<code>role.search.subtree</code>	Specifies if the role entry search's scope includes the subtrees of the tree selected by <code>role.base.dn</code> .
<code>authentication</code>	Specifies the authentication method used when binding to the LDAP server. Valid values are • <code>simple</code>—bind with user name and password authentication • <code>none</code>—bind anonymously
<code>initial.context.factory</code>	Specifies the class of the context factory used to connect to the LDAP server. This must always be set to <code>com.sun.jndi.ldap.LdapCtxFactory</code> .
<code>ssl</code>	Specifies if the connection to the LDAP server is secured via SSL. If <code>connection.url</code> starts with <code>ldaps://</code> SSL is used regardless of this property.
<code>ssl.provider</code>	Specifies the SSL provider to use for the LDAP connection. If not specified, the default SSL provider is used.
<code>ssl.protocol</code>	Specifies the protocol to use for the SSL connection.
<code>ssl.algorithm</code>	Specifies the algorithm used by the trust store manager.
<code>ssl.keystore</code>	Specifies the keystore name.
<code>ssl.keyalias</code>	Specifies the name of the private key in the keystore.

Property	Description
ssl.truststore	Specifies the trust keystore name.

All of the properties are mandatory except the SSL properties.

Example

[Example 7.7 on page 64](#) defines a JASS realm that uses the LDAP server located at `ldap://localhost:10389`.

Example 7.7. Configuring a JAAS Realm that Uses LDAP Authentication

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

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

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


Using Encrypted Property Placeholders

Overview

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

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

1. Create a properties file with encrypted values.
2. Add the proper namespaces to your blueprint file.
3. Import the properties using the Aries property placeholder extension.
4. Configure the Jasypt encryption algorithm.
5. Use the placeholders in your blueprint file.
6. Ensure that the Jasypt features are installed into the Fuse MQ Enterprise container.

Encrypted properties

Encrypted properties are stored in plain properties files. They are identified by wrapping them in the `ENC()` function as shown in [Example 7.8 on page 65](#).

Example 7.8. Property File with an Encrypted Property

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



Important

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

Namespaces

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

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

[Example 7.9 on page 66](#) shows a blueprint file with the required namespaces.

Example 7.9. Encrypted Property Namespaces

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

Placeholder extension

In order to use encrypted property placeholders in a blueprint file you need to include an Aries `property-placeholder` element to your blueprint file. As shown in [Example 7.10 on page 66](#), it must come before the Jasypt configuration or the use of placeholders.

Example 7.10. Aries Placeholder Extension

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

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

...
</blueprint>
```

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

Jasypt configuration

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

The `encoder` element's mandatory `class` attribute specifies the fully qualified classname of the Jasypt encryptor to use for decrypting the properties. The

encoder element can take a `property` child that defines a Jasypt `PBEConfig` bean for configuring the encryptor.

For detailed information on how to configure the different Jasypt encryptors, see the [Jasypt documentation](http://www.jasypt.org/)².

[Example 7.11 on page 67](#) shows configuration for using the string encryptor and retrieving the password from an environment variable.

Example 7.11. Jasypt Blueprint Configuration

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

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

  <enc:property-placeholder>
    <enc:encryptor class="org.jasypt.encryption.pbe.StandardPBESStringEncryptor">
      <property name="config">
        <bean class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
          <property name="algorithm" value="PBEWithMD5AndDES" />
          <property name="passwordEnvName" value="FUSE_ENCRYPTION_PASSWORD" />
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>
  ...
</blueprint>
```

Placeholders

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

[Example 7.12 on page 67](#) shows an LDAP JAAS realm that uses the properties file in [Example 7.8 on page 65](#).

Example 7.12. Jasypt Blueprint Configuration

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

  <ext:property-placeholder>
```

² <http://www.jasypt.org/general-usage.html>

```

<location>file://ldap.properties</location>

<enc:property-placeholder>
  <enc:encryptor class="org.jasypt.encryption.pbe.StandardPBESStringEncryptor">
    <property name="config">
      <bean class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
        <property name="algorithm" value="PBEWithMD5AndDES" />
        <property name="passwordEnvName" value="FUSE_ENCRYPTION_PASSWORD" />
      </bean>
    </property>
  </enc:encryptor>
</enc:property-placeholder>

<jaas:config name="karaf" rank="1">
  <jaas:module className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule" flags="required">
    initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
    debug=true
    connectionURL=${ldap.url}
    connectionUsername=cn=mqbroker,ou=Services,ou=system,dc=fusesource,dc=com
    connectionPassword=${ldap.password}
    connectionProtocol=
    authentication=simple
    userRoleName=cn
    userBase = ou=User,ou=ActiveMQ,ou=system,dc=fusesource,dc=com
    userSearchMatching=(uid={0})
    userSearchSubtree=true
    roleBase = ou=Group,ou=ActiveMQ,ou=system,dc=fusesource,dc=com
    roleName=cn
    roleSearchMatching=(member:=uid={1})
    roleSearchSubtree=true
  </jaas:module>
</jaas:config>
</blueprint>

```

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

Installing the Jasypt features

By default, Fuse ESB Enterprise does not have the Jasypt encryption libraries installed. In order to use encrypted property placeholders, you will need to install the `jasypt-encryption` feature using Fuse MQ Enterprise's **features:install** command as shown in [Example 7.13 on page 68](#).

Example 7.13. Installing the Jasypt Feature

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

Configuring Roles for the Administrative Protocols

Overview

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

Administration protocols

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

- SSH (remote console login)
  - JMX management
-

Default role

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

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

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

Changing the remote console's role

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

```
sshRole=admin
```

Changing the JMX role

To override the default role for JMX add a `jmxRole` property to the `etc/org.apache.karaf.management.cfg` file.

Chapter 8. Sending Commands to the Broker

Fuse MQ Enterprise provides a number of commands that can be used to manage a broker, deploy new brokers, and report administrative details. You can send these commands to a broker using either the broker command console or the administration client.

Overview

The default mode for running a Fuse MQ Enterprise broker is to run in daemon mode. In this mode, the broker runs as a background process and you have no direct means for managing it or requesting status information. You can access a broker in daemon mode in the following ways:

- the Fuse MQ Enterprise administration client that can be used to send any of the console commands to a broker running in daemon mode
- a broker running in console mode can connect to a remote broker and be used to manage the remote broker
- Fuse MQ Enterprise includes a vanilla Apache Karaf shell that can connect to a remote broker and be used to manage the remote broker

If a broker is started in console mode, you can simply enter commands directly in the command console.

Running the administration client

The Fuse MQ Enterprise administration client is run using the **client** in `InstallDir/bin`. [Example 8.1 on page 71](#) shows the syntax for the command.

Example 8.1. Client Command

```
client [--help] [-a port] [-h host] [-u user] [-p password] [-v] [-r attempts] [-d delay] [commands]
```

[Table 8.1 on page 71](#) describes the command's arguments.

Table 8.1. Administration Client Arguments

Argument	Description
--help	Displays the help message.

Argument	Description
-a	Specifies the remote host's port.
-h	Specify the remote host's name.
-u	Specifies user name used to log into the broker.
-p	Specifies the password used to log into the broker.
-v	Use verbose output.
-r	Specifies the maximum number of attempts to establish a connection.
-d	Specifies, in seconds, the delay between retries. The default is 2 seconds.
<i>commands</i>	Specifies one or more commands to run. If no commands are specified, the client enters an interactive mode.

Using the broker console

The console provides commands that you can use to perform basic management of your Fuse MQ Enterprise environment, including managing destinations, connections and other administrative objects in the broker.

The console uses prefixes to group commands relating to the same functionality. For example commands related to configuration are prefixed **config:**, and logging-related commands are prefixed **log:**.

The console provides two levels of help:

- console help—list all of the commands along with a brief summary of the commands function
- command help—a detailed description of a command and its arguments

To access the console help you use the **help** command from the console prompt. It will display a grouped list of all the commands available in the console. Each command in the list will be followed by a description of the command as shown in [Example 8.2 on page 73](#).

Example 8.2. Console Help

```
FuseMQ:karaf@root> help
COMMANDS
    activemq:browse
    activemq:bstat
    activemq:list
    activemq:purge
    activemq:query
    admin:change-opts
        Changes the Java options of an existing container
instance.
    admin:change-rmi-registry-port
        Changes the RMI registry port (used by management
layer) of an
        existing container instance.
    ...
FuseMQ:karaf@root>
```

The help for each command includes the definition, the syntax, and the arguments and any options. To display the help for a command, type the command with the `--help` option. As shown in [Example 8.3 on page 73](#), entering `admin:start --help` displays the help for that command.

Example 8.3. Help for a Command

```
FuseMQ:karaf@root> admin:start --help
DESCRIPTION
    admin:start

    Starts an existing container instance.

SYNTAX
    admin:start [options] name

ARGUMENTS
    name                The name of the container instance

OPTIONS
    --help              Display this help message
    -o, --java-opts     Java options when launching the instance
```

```
FuseMQ:karaf@root>
```

Connecting a console to a remote broker

How you connect a command console to a broker on a remote machine depends on if the brokers are part of the same fabric. If the remote broker you want to command is a part of the same fabric as the broker whose command console you are using, then you can use the **fabric:container-connect** command to establish a connection to the remote broker.

The **fabric:container-connect** command has one required argument that specifies the name of the container to which a connection will be opened. You can also specify a command to be executed by the remote console connection. If you do not specify a command, you are presented with a prompt that will pass commands to the remote broker's console. For more details about **fabric:container-connect**, see [fabric:container-connect](#), in *Console Reference*.

If you are not using fabric, or the remote broker is not part of the same fabric as the broker whose command console you are using, you create a remote connection using the **ssh:ssh** command. The **ssh:ssh** command also only requires a single argument to establish the remote connection. In this case, it is the hostname, or IP address, of the machine on which the broker is running. If the remote broker is not using the default SSH port (8101), you will also need to specify the remote broker's SSH port using the `-p` flag. You can also specify a command to be executed by the remote console connection. If you do not specify a command, you are presented with a prompt that will pass commands to the remote broker's console. For more details about **ssh:ssh**, see [ssh:ssh](#), in *Console Reference*.

To disconnect from the remote console, you use the **logout** command or press Control+D.

Starting a basic console

Fuse MQ Enterprise includes a **shell** command that will open a vanilla command console without starting a broker instance. You can use this command console to connect to remote brokers in the same way as you would a broker's command console.

Available commands

The remote client can execute any of the broker's console commands. For a complete list of commands see the [Console Reference](#).

Chapter 9. Deploying a New Broker

In most large messaging environments there will be multiple brokers deployed. This may be for load management, high availability, or other business reasons. Using standalone brokers this requires manually installing and configuring multiple instances of Fuse MQ Enterprise. Using a fabric, however, you can deploy multiple brokers from a single location and easily reuse large portions of the configuration.

Deploying a Standalone Broker	76
Deploying a New Broker into a Fabric	77

When deploying multiple brokers, you need to decide how you want to manage the brokers:

- as a collection of standalone brokers
- a fabric of brokers

All of the advanced networking features such as fail over, network of brokers, load balancing, and master/slave are available regardless of how you choose to manage your broker deployment. The difference is in what is required to set up and maintain the deployment.

Using a collection of standalone brokers requires that you install, configure, and maintain each broker separately. If you have three brokers, you will need to manually install Fuse MQ Enterprise on three machines and configure each installation separately. This can be cumbersome and error prone particularly when configuring a network of brokers. When issues arise or you need to update your deployment, you will have to make the changes on each machine individually.

If you brokers are deployed into a fabric, you can perform the installation and configuration of all the brokers in the deployment from a central location. In addition, using a fabric simplifies the configuration process and makes it less error prone. Fabric provides tooling for auto-configuring failover clusters, networks of brokers, and master/slave clusters. In addition, it also makes it possible to place all of the common configuration into a single profile that all of the brokers share. When issues arise or you need to update your deployment, having your brokers in a fabric allows you to do incremental roll outs and provides a means for quickly rolling back any changes.

Deploying a Standalone Broker

Overview

Deploying a new standalone broker involves installing Fuse MQ Enterprise on a new machine and modifying its configuration as needed. You will need to do this for all of the additional brokers in your deployment.

Procedure

To deploy a new standalone broker:

1. Install Fuse MQ Enterprise onto the target system as described in [Installation Guide](#).
2. Modify the new installation's configuration for your environment as described in ["Editing a Broker's Configuration" on page 17](#).

You will need to repeat this process for each standalone broker you want to deploy.

More information

For more information on configuring brokers to work together see:

- [Using Networks of Brokers](#)
- [Fault Tolerant Messaging](#)

Deploying a New Broker into a Fabric

Overview

Deploying a new broker instance into a fabric involves creating a new broker profile and deploying it to a Fabric Container. The fabric infrastructure simplifies these tasks by:

- allowing you to do them from a remote location
- providing tools that assist in automatically configuring fail over clusters, networks of brokers, and master/slave clusters.
- allowing you to reuse parts of existing profiles to ensure consistency
- providing tooling to do rolling updates
- providing tooling to roll back changes when needed

From the Fuse MQ Enterprise console you can use the **fabric:mq-create** to create new broker profiles and new containers for the brokers. You can also use Fuse Management Console to perform to create the profiles and assign them to containers.

Procedure

To deploy a new broker into a fabric:

1. Create a template Fuse MQ Enterprise XML configuration file in a location that is accessible to the container.
2. In the command console, use the **fabric:import** command to upload the your XML configuration template to the Fabric Ensemble as shown in [Example 9.1 on page 77](#).

Example 9.1. Uploading a Template to a Fabric Ensemble

```
FuseESB:karaf@root> fabric:import -t /fabric/configs/versions/version/profiles/mq-base/configFile configFile
```

version must match the version of the new profile you will create for the new broker.

3. Use the **fabric:mq-create** command to create a profile for the new broker and assign it to a container.
  - To deploy the new broker into an existing container use the command shown in [Example 9.2 on page 78](#)

Example 9.2. Creating a New Broker in an Existing Container

```
FuseESB:karaf@root> fabric:mq-create --assign-container containerName --config configFile profileName
```

This will create a new broker profile that inherits from the `mq-base` profile, but uses your XML configuration template, and deploy it to the specified container.

- To deploy the new broker into an new container use the command shown in [Example 9.3 on page 78](#)

Example 9.3. Creating a New Broker in a New Container

```
FuseESB:karaf@root> fabric:mq-create --create-container containerName --config configFile profileName
```

This will create a new broker profile that inherits from the `mq-base` profile, but uses your XML configuration template, create a new container named `containerName`, and deploy the broker profile to it.



Note

The new container will be a child of the container from which you execute the **fabric:mq-create** command.



Tip

You can add network configuration settings to the profile as well. See [fabric:mq-create](#) in *Console Reference*.

4. Use the **fabric:profile-edit** command shown in [Example 9.4 on page 78](#) to set the required properties.

Example 9.4. Editing a Broker Profile

```
FuseESB:karaf@root> fabric:profile-edit --pid org.fusesource.mq.fabric.server-profileName/property=value  
profileName
```

The properties that need to be set will depend on the properties you specified using property place holders in the template XML configuration and the broker's network settings.

For information on using **fabric:profile-edit** see [fabric:profile-edit](#) in *Console Reference*.



Tip

Fuse Management Console makes this process easier by providing a Web based UI.

More information

For more information on configuring brokers to work together see:

- [Using Networks of Brokers](#)
- [Fault Tolerant Messaging](#)

For more information on using Fuse Management Console see the [Fuse Management Console Documentation](#)¹.

---

¹ <http://fusesource.com/documentation/fuse-management-console-documentation/>

Chapter 10. Shutting Down a Broker

Brokers can be shutdown from either the machine on which they are running or remotely from a different machine.

Shutting Down a Local Broker	82
Shutting Down a Broker Remotely	83



Important

If the broker is running in console mode it can only be shutdown locally.

Shutting Down a Local Broker

Overview

The method used to stop a broker running on the machine you logged into depends on the mode in which the broker is running. If it is running in console mode, you use one of the console commands to shut down the broker. If it is running in daemon mode, the broker doesn't have a command console. So, you need to use one of the utility commands supplied with Fuse MQ Enterprise.

Stopping the broker from console mode

If you launched Fuse MQ Enterprise by running **fusemq**, you shut it down using the **shutdown -f** command as shown in [Example 10.1 on page 82](#).

Example 10.1. Using the Console's Shutdown Command

```
FuseMQ:karaf@root> shutdown -f
FuseMQ:karaf@root>
logout

[Process completed]
```



Tip

CTRL+D will also shutdown the broker.

Stopping a broker running in daemon mode

If you launched Fuse MQ Enterprise by running the **start** command, log in to the machine where the broker is running and run the **stop** command in the broker installation's `bin` folder.



Tip

You can stop a broker running in daemon mode remotely. See ["Shutting Down a Broker Remotely" on page 83](#).

Shutting Down a Broker Remotely

Overview

For many use cases logging into the machine running a broker instance is impractical. In those cases, you need a way to stop a broker from a remote machine. Fuse MQ Enterprise offers a number of ways to accomplish this task:

- using the **stop** command—the stop command does not require starting an instance of the broker
- using a remote console connection—a broker's console can be used to remotely shutdown a broker on another machine
- using a fabric member's console—brokers that are part of a fabric can stop members of their fabric
- using Fuse Management Console—brokers that are part of a fabric can be stopped using an Fuse Management Console connected to their fabric

For more information see the [Fuse Management Console Documentation](#)¹.

Using the stop command

You can stop a remote instance without starting up Fuse ESB Enterprise on your local host by running the **stop** command in the `InstallDir/bin` directory. The commands syntax is shown in [Example 10.2 on page 83](#).

Example 10.2. Stop Command Syntax

```
stop [-a port] {-h hostname} {-u username} {-p password}
```

`-a port`

Specifies the SSH port of the remote instance. The default is 8101.

`-h hostname`

Specifies the hostname of the machine on which the remote instance is running.

`-u username`

Specifies the username used to connect to the remote broker.

¹ <http://fusesource.com/docs/fmc>



Tip

The default username for a broker is `karaf`.

-p *password*

Specifies the password used to connect to the remote broker.



Tip

The default password for a broker is `karaf`.

[Example 10.3 on page 84](#) shows how to stop a remote broker on a machine named `NEBrokerHost2`.

Example 10.3. Stopping a Remote Broker

```
bin/stop -u karaf -p karaf -h NEBrokerHost2
```

Using a remote console

Fuse MQ Enterprise's console can be connected to a remote broker using the **ssh:ssh** command. Once the console is connected to the remote broker, you can shut it down by running the **osgi:shutdown** command.

Example 10.4 on page 84 shows the command sequence for using a remote console connection to shutdown a broker running on a machine named `NWBrokerHost`.

Example 10.4. Shutting Down a Broker using a Remote Console Connection

```
FuseMQ:karaf@root> ssh -l karaf -P karaf NWBrokerHost
```

Fuse MQ (7.0.0.fuse-beta-040)
<http://fusesource.org/mq/>

```
Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or 'osgi:shutdown' to shutdown Fuse MQ.
```

```
FuseMQ:karaf@root> osgi:shutdown
Confirm: shutdown instance root (yes/no):
yes
FuseMQ:karaf@root> FuseESB:karaf@root>
```



Important

Pressing Control+D when connected to a remote broker closes the remote connection and returns you to the local shell.

Shutting down remote brokers in a fabric

If the broker you want to shutdown is part of a fabric, you can shut it down from any of the brokers in the fabric using the **fabric:container-stop** console command. **fabric:container-stop** takes the name of the fabric container hosting the broker as an argument. The command can be run either from a broker in console mode or using the broker's administration client.

[Example 10.5 on page 85](#) shows how to use the administration client to shutdown a broker running in a container named `fabric-broker3`.

Example 10.5. Shutting Down a Broker in a Fabric

```
./bin/client fabric-broker3 fabric:container-stop
```

For more information see [fabric:container-stop](#), in *Console Reference*.

Chapter 11. Connecting a Broker to a Fabric

If you want a standalone broker to become part of a fabric you can either create the fabric from the standalone broker or join the broker to an existing fabric. Once a broker is added to a fabric, it reverts to an empty fabric container.

Joining a Broker to a Fabric	88
Joining a Fabric as a Managed Container	89
Joining a Fabric as a Non-Managed Container	93
Creating a New Fabric	95

When Fuse MQ Enterprise is installed it is set up to run a standalone broker. If you want the broker to become part of a fabric you have two options:

- [join on page 88](#) an existing fabric

This option is useful if you have an existing fabric and simply want to add the broker to it. You can add the broker as a fully managed container in which the broker's existing configuration is wiped out and replaced with a profile from the fabric's repository. You can also add the broker as an unmanaged container that retains all of its configuration, but can be discovered through the fabric's ensemble.



Tip

This option can also be used to move a broker from one fabric to another.

- [create on page 95](#) a new fabric

This option converts the standalone broker into a Fabric Server. You will then need to either add a new broker to the fabric, or assign a broker profile to the server.

Joining a Broker to a Fabric

Joining a Fabric as a Managed Container	89
Joining a Fabric as a Non-Managed Container	93

Any standalone broker can be joined to an existing fabric using the **fabric:join**. You need to supply the URL of one of the Fuse Servers in the fabric and the standalone broker is added to the fabric. The broker can join the fabric as either a managed container or a non-managed container:

- A [managed container on page 89](#) is a full member of the fabric and is managed by a Fabric Agent. The agent configures the container based on information provided by the fabric's ensemble. The ensemble knows which profiles are associated with the container and the agent determines what to install based on the contents of the profiles.
- A [non-managed container on page 93](#) is not managed by a Fabric Agent. Its configuration remains intact after it joins the fabric and is controlled as if the broker were a standalone broker. Joining the fabric in this manner registers the broker with the fabric's ensemble and allows clients to locate the broker using the fabric's discovery mechanism.

Joining a Fabric as a Managed Container

Overview

When a broker joins a fabric as a managed container, it stops being a broker because the default behavior of the **fabric:join** command is to wipe out the container's configuration and replace it with the `fabric` profile. To start up as broker, you need to ensure that the fabric has a profile with the proper configuration and associate it with the container. The **fabric:join** command's `-p` argument allows you to specify a profile to install into the container once the agent is installed.

Becoming a managed container

Several things happen when a broker joins a fabric as a managed container:

1. The broker installs the required Fuse Fabric bundles to interact with the fabric's ensemble.
2. The broker contacts the specified Fabric Server and initiates the joining process.
3. The Fabric Server registers the container with the fabric's ensemble.

This adds the container's information to the fabric's registry using the container name as the key. Because the container is a managed container, the registry creates an empty entry for the container and only includes the information it needs to manage the container.



Warning

If the container being added to the fabric has the same name as a container already registered with the fabric, both containers will be reset and will always share the same configuration.

4. The Fabric Server, as a delegate of the fabric's ensemble, takes control of the container.

This process clears the container and resets its configuration to a default state.

5. The ensemble installs the `fabric` profile into the container.

This loads the Fabric Agent into the container to facilitate the management of the container.

6.
  - If the `-p` argument is passed the **fabric:join**, the container's agent installs the artifacts and configuration settings from the specified profile.
  - The container sits empty and ready for you to associate one or more profiles with it.
7. The agent monitors the container and the ensemble for changes.

If the state of the container changes, the agent updates the ensemble's registry entry. If the ensemble has updates for the container, such as a new profile being associated with the container or a change to one of the profiles already associated with the container, the agent updates the container.

Remaining a message broker

Joining a fabric as a managed container converts a standalone broker into a vanilla fabric container. It can take on any characteristics that are required. If you want it to join the fabric and continue to function as a message broker, then you must ensure that the fabric being joined has a profile that will configure the container to be a message broker and assign that profile to the container.

The default set of profiles installed with Fuse Fabric include an `mq` profile that loads a default message broker. You can also create a new profile for your brokers as described in ["Editing a Broker's Configuration in a Fabric" on page 24](#).

The easiest way to assign the profile is to do so when executing the **fabric:join** by using the `-p` flag. The specified profile will be assigned to the container as soon as the agent is installed. If you would rather do it in two steps, you can assign the profile using the **fabric:container-change-profile** command or Fuse Management Console.

Procedure

To join a broker to a fabric and have it start up as a broker:

1. Create a profile for your broker.

For details on creating a broker profile see ["Editing a Broker's Configuration in a Fabric" on page 24](#).

2. Get the URL for one of the Fabric Servers in the existing fabric.

The URL of a Server has the following format:

```
HostName:IPPort
```

For example, given a fabric registry agent running on the host, `myhost`, the URL would be `myhost:2181`. The IP port, 2181, is the default IP port used by a Fabric Server and is usually the correct value to use. If you are in any doubt about which URL to use you can discover the URLs of the Fabric Servers as follows:

1. Connect to the command console of one of the containers in the fabric.
2. Enter the following sequence of console commands:

```
FuseMQ:karaf@root> config:edit org.fusesource.fabric.zookeeper
er
FuseMQ:karaf@root> config:proplist
    service.pid = org.fusesource.fabric.zookeeper
    zookeeper.url = myhostA:2181,myhostB:2181,myhostC:2181,my
hostC:2182,myhostC:2183
    fabric.zookeeper.pid = org.fusesource.fabric.zookeeper
FuseMQ:karaf@root> config:cancel
```

The `zookeeper.url` property holds a comma-separated list of Fabric Server URLs. You can use any one of these URLs to join the fabric.

3. Connect to the standalone broker's command console.
4. Enter the following command:

```
FuseMQ:karaf@root> fabric:join -p brokerProfile fabricURL brokerName
```

You need to provide values for:

- *brokerProfile*—the name of the profile you created in [Step 1 on page 90](#)
- *fabricURL*—the Fabric Server URL you obtained in [Step 2 on page 90](#)
- *brokerName*—the name the broker will use to register with the fabric



Important

The name should be unique among the containers in the fabric.

For details on **fabric:join** see [fabric:join](#) in *Console Reference*.

Joining a Fabric as a Non-Managed Container

Overview

When a broker joins a fabric as a non-managed container, it continues being a standalone broker because a Fabric Agent does not take control of the container. The agent only registers the broker with the fabric's ensemble and keeps the registry entries for it up to date. This enables consumers to discover the broker using the fabric's discovery mechanism described in ["Fuse Fabric Discovery Agent"](#) in *Using Networks of Brokers*.

Becoming a non-managed container

Several things happen when a broker joins a fabric as a non-managed container:

1. The broker installs the required Fuse Fabric bundles to interact with the fabric's ensemble.
2. The broker contacts the specified Fabric Server and initiates the joining process.
3. The Fabric Server registers the container with the fabric's ensemble.

This adds the broker's information to the fabric's registry using the broker's name as the key. Because the broker is non-managed, the registry creates an entry containing information for all the message destinations hosted by the broker.



Warning

If the container being added to the fabric has the same name as a container already registered with the fabric, both containers will be reset and will always share the same configuration.

4. The agent monitors the broker and updates the updates the ensemble's registry entry as needed.
-

Procedure

To join a broker to a fabric a non-managed container:

1. Get the URL for one of the Fabric Servers in the existing fabric.

The URL of a Server has the following format:

```
HostName:IPPort
```

For example, given a fabric registry agent running on the host, `myhost`, the URL would be `myhost:2181`. The IP port, 2181, is the default IP port used by a Fabric Server and is usually the correct value to use. If you are in any doubt about which URL to use you can discover the URLs of the Fabric Servers as follows:

1. Connect to the command console of one of the containers in the fabric.
2. Enter the following sequence of console commands:

```
FuseMQ:karaf@root> config:edit org.fusesource.fabric.zookeeper
er
FuseMQ:karaf@root> config:proplist
  service.pid = org.fusesource.fabric.zookeeper
  zookeeper.url = myhostA:2181,myhostB:2181,myhostC:2181,my
hostC:2182,myhostC:2183
  fabric.zookeeper.pid = org.fusesource.fabric.zookeeper
FuseMQ:karaf@root> config:cancel
```

The `zookeeper.url` property holds a comma-separated list of Fabric Server URLs. You can use any one of these URLs to join the fabric.

2. Connect to the standalone broker's command console.
3. Enter the following command:

```
FuseMQ:karaf@root> fabric:join -n fabricURL brokerName
```

You need to provide values for:

- `-n`—specifies that the broker will **not** be managed by the fabric
- `fabricURL`—the Fabric Server URL you obtained in [Step 1 on page 93](#)
- `brokerName`—the name the broker will use to register with the fabric



Important

The name should be unique among the containers in the fabric.

For details on **`fabric:join`** see [fabric:join](#) in *Console Reference*.

Creating a New Fabric

Overview

If there is no existing fabric for a broker to join, you will need to create one from the standalone broker. There are two options for how to do this:

- The recommended approach is to create the fabric such that the broker is converted into a managed Fabric Server. The container will revert to being a vanilla broker, so you will need to either create a new broker with the desired configuration or deploy an updated broker profile into the Fabric Server.

Ensuring that the new broker is configured properly requires creating a broker profile with the proper settings. This can be done once the new fabric is created using the **fabric:mq-create** command. The **fabric:mq-create** command can also create the new broker.

- The alternative approach is to create the fabric such that the broker becomes a non-managed Fabric Server. The broker becomes a Fabric Server that is fully capable of being part of a fabric's ensemble. However, the broker retains control over its own configuration. It will continue to be managed as if it were a standalone broker and will not respond to fabric commands that attempt to alter its configuration.

Becoming a new fabric

Several things happen when a fabric is created from a standalone broker:

1. The broker installs the required Fuse Fabric bundles to become a Fabric Server.
2. The Fabric Server starts up the ensemble process.
3. A new ensemble containing an empty configuration registry and a runtime registry with a single entry for the Fabric Server is created.
4. The Fabric Server delegates control over its configuration to the ensemble.



Note

If the fabric is created with a non-managed server, this step is skipped.

5. The ensemble associates the profiles required to be a Fabric Server with the container.
6. Unless the `--no-import` flag is used, the ensemble imports a set of profiles.
  - If the `--import-dir` flag is used, the profiles are imported from the specified location.
  - If no location is specified the default set of profiles is imported from `InstallDir/fabric/import`.
7. The ensemble installs the default `mq` profile to the container.

This starts a default broker instance in the container.



Note

If the fabric is created with a non-managed server, this step is skipped.

8. The ensemble waits for commands to update the fabric.

Procedure

To create a new fabric from a standalone broker:

1. Delete the standalone broker configuration(s) from your installation's `/etc` folder.

The configuration is stored in
`/etc/org.fusesource.mq.fabric.server-configName.cfg`.

2. Connect to the standalone broker's command console.
3. Remove any brokers running in the container.
  - a. Use **config:list** to locate any PIDs using the form `org.fusesource.mq.fabric.server.ID`.
  - b. Use **config:delete** to delete all of the PIDs using the form `org.fusesource.mq.fabric.server.ID`.

4. Enter the following command:

```
FuseMQ:karaf@root> fabric:create
```



Tip

If you want to import a predefined set of profiles, use the **-p** *import-dir* option to specify the set of profiles to import.

For details on **fabric:create** see [fabric:create](#) in *Console Reference*.

5. Add a new broker to the fabric as described in ["Deploying a New Broker into a Fabric"](#) on page 77.

Chapter 12. Using Logging

The broker's log contains information about all of the critical events that occur in the broker. You can configure the granularity of the logged messages to provide the required amount of detail.

Logging Configuration	100
Viewing the Log	103

Fuse MQ Enterprise uses the *OPS4j Pax Logging* system. Pax Logging is an open source OSGi logging service that extends the standard OSGi logging service to make it more appropriate for use in enterprise applications. It uses Apache Log4j as the back-end logging service. Pax Logging has its own API, but it also supports the following APIs:

- Apache Log4j
- Apache Commons Logging
- SLF4J
- Java Util Logging

Logging Configuration

Overview

The logging system is configured by a combination of two OSGi Admin PIDs and one configuration file:

- `etc/system.properties`—the configuration file that sets the logging level during the broker's boot process. The file contains a single property, `org.ops4j.pax.logging.DefaultServiceLog.level`, that is set to `ERROR` by default.
- `org.ops4j.pax.logging`—the PID used to configure the logging back end service. It sets the logging levels for all of the defined loggers and defines the appenders used to generate log output. It uses standard Log4j configuration. By default, it sets the root logger's level to `INFO` and defines two appenders: one for the console and one for the log file.



Tip

The console's appender is disabled by default. To enable it, add `log4j.appender.stdout.append=true` to the configuration. For example, to enable the console appender in a broker, you would use the following commands:

```
FuseMQ:karaf@root> config:edit org.ops4j.pax.logging
FuseMQ:karaf@root> config:propappend log4j.appender.stdout.append true
FuseMQ:karaf@root> config:update
```

- `org.apache.karaf.log.cfg`—configures the output of the **log** console commands.

The most common configuration changes you will make are changing the logging levels, changing the threshold for which an appender writes out log messages, and activating per bundle logging.

Changing the log levels

The default logging configuration sets the logging levels so that the log file will provide enough information to monitor the behavior of the runtime and provide clues about what caused a problem. However, the default configuration will not provide enough information to debug most problems.

The most useful logger to change when trying to debug an issue with Fuse MQ Enterprise is the root logger. You will want to set its logging level to generate more fine grained messages. To do so you change the value of the `org.ops4j.pax.logging` PID's `log4j.rootLogger` property so that the logging level is one of the following:

- TRACE
- DEBUG
- INFO
- WARN
- ERROR
- FATAL
- NONE

[Example 12.1 on page 101](#) shows the commands for setting the root loggers log level in a standalone broker.

Example 12.1. Changing Logging Levels

```
FuseMQ:karaf@root> config:edit org.ops4j.pax.logging
FuseMQ:karaf@root> config:propset log4j.rootLogger "DEBUG, out,
osgi:VmLogAppender"
FuseMQ:karaf@root> config:update
```

Changing the appenders' thresholds

When debugging a problem in Fuse MQ Enterprise you may want to limit the amount of logging information that is displayed on the console, but not the amount written to the log file. This is controlled by setting the thresholds for each of the appenders to a different level. Each appender can have a `log4j.appender.appendName.threshold` property that controls what level of messages are written to the appender. The appender threshold values are the same as the log level values.

[Example 12.2 on page 102](#) shows an example of setting the root logger to `DEBUG` but limiting the information displayed on the console to `WARN`.

Example 12.2. Changing the Log Information Displayed on the Console

```
FuseESB:karaf@root> config:edit org.ops4j.pax.logging
FuseMQ:karaf@root> config:propset log4j.rootLogger "DEBUG, out,
osgi:VmLogAppender"
FuseMQ:karaf@root> config:propappend log4j.appender.stdout.threshold
WARN
FuseMQ:karaf@root> config:update
```

Viewing the Log

Overview

There are three ways you can view the log:

- using a text editor
  - using the broker's, or a remote broker's, console
  - using the administration client
-

Viewing the log in a text editor

The log files are stored as simple text files in `InstallDir/data/log`. The main log file is `karaf.log`. If archiving is turned on, there may be archived log files also stored in the logging directory.

Log entries are listed in chronological order with the oldest entries first. The default output displays the following information:

- the time of the entry
  - the log level of the entry
  - the thread that generated the entry
  - the bundle that generated the entry
  - an informational message about the cause of the entry
-

Viewing the log with the console

The Fuse MQ Enterprise console provides the following commands for viewing the log:

- **log:display**—displays the most recent log entries

By default, the number of entries returned and the pattern of the output depends on the `size` and `pattern` properties in the `org.apache.karaf.log.cfg` file. You can override these using the `-p` and `-d` arguments.

- **log:display-exception**—displays the most recently logged exception
- **log:tail**—continuously display log entries

For more information on using these commands see ["Log Console Commands"](#) in *Console Reference*.

Viewing the log with the administration client

If you do not have a broker running in console mode, you can also use the administration client to invoke the broker's log displaying commands. For example, entering `client log:display` into a system terminal will display the most recent log entries for the local broker.

Chapter 13. Using JMX

Fuse MQ Enterprise is fully instrumented to provide statistics about its performance using JMX. You can monitor a broker using any JMX aware monitoring tool.

Configuring JMX	106
Statistics Collected by JMX	108
Managing the Broker with JMX	111

By default Fuse MQ Enterprise creates MBeans, loads them into the MBean server created by the JVM, and creates a dedicated JMX connector that provides a Fuse MQ Enterprise-specific view of the MBean server. The default settings are sufficient for simple deployments and make it easy to access the statistics and management operations provided by a broker. For more complex deployments you easily configure many aspects of how a broker configures itself for access through JMX. For example, you can change the JMX URI of the JMX connector created by the broker or force the broker to use the generic JMX connector created by the JVM.

By connecting a JMX aware management and monitoring tool to a broker's JMX connector, you can view detailed information about the broker. This information provides a good indication of broker health and potential problem areas. In addition to the collected statistics, Fuse MQ Enterprise's JMX interface provides a number of operations that make it easy to manage a broker instance. These include stopping a broker, starting and stopping network connectors, and managing destinations.

Configuring JMX

Overview

By default a broker is set up to allow for JMX management. It uses the JVM's MBean server and creates its own JMX connector at `service:jmx:rmi:///jndi/rmi://hostname:1099/jmxrmi`. If the default configuration does not meet the needs of the deployment environment, the broker provides configuration properties for customizing most aspects of its JMX behavior. For instance, you can completely disable JMX for a broker. You can also force the broker to create its own MBean server.

Enabling and disabling

By default JMX is enabled for a Fuse MQ Enterprise broker. To disable JMX entirely you simply set the `broker` element's `useJmx` attribute to `false`. This will stop the broker from exposing itself via JMX.

 Important

Disabling JMX will also disable the commands in the **activemq** shell.

Securing access to JMX

In a production environment it is advisable to secure the access to your brokers' management interfaces. To set up authentication To override the default role for JMX access add a `jmxRole` property to the `etc/org.apache.karaf.management.cfg` file.

Advanced configuration

If the default JMX behavior is not appropriate for your deployment environment, you can customize how the broker exposes its MBeans. To customize a broker's JMX configuration, you add a `managementContext` child element to the broker's `broker` element. The `managementContext` element uses a `managementContext` child to configure the broker. The attributes of the inner `managementContext` element specify the broker's JMX configuration.

[Table 13.1 on page 106](#) describes the configuration properties for controlling a broker's JMX behavior.

Table 13.1. Broker JMX Configuration Properties

Property	Default Value	Description
<code>useMBeanServer</code>	<code>true</code>	Specifies whether the broker will use the MBean server created by the JVM.

Property	Default Value	Description
		When set to <code>false</code> , the broker will create an MBean server.
<code>jmxDomainName</code>	<code>org.apache.activemq</code>	Specifies the JMX domain used by the broker's MBeans.
<code>createMBeanServer</code>	<code>true</code>	Specifies whether the broker creates an MBean server if none is found.
<code>createConnector</code>	<code>true</code> ^a	Specifies whether the broker creates a JMX connector for the MBean server. If this is set to <code>false</code> the broker will only be accessible using the JMX connector created by the JVM.
<code>connectorPort</code>	<code>1099</code>	Specifies the port number used by the JMX connector created by the broker.
<code>connectorHost</code>	<code>localhost</code>	Specifies the host used by the JMX connector and the RMI server.
<code>rmiServerPort</code>	<code>0</code>	Specifies the RMI server port. This setting is useful if port usage needs to be restricted behind a firewall.
<code>connectorPath</code>	<code>/jmxrmi</code>	Specifies the path under which the JMX connector will be registered.

^aThe default configuration template for the broker sets this property to `false` so that the broker uses the container's JMX connection.

[Example 13.1 on page 107](#) shows configuration for a broker that will only use the JVM's MBean server and will not create its own JMX connector.

Example 13.1. Configuring a Broker's JMX Connection

```
<broker ... >
...
<managementContext>
  <managementContext createMBeanServer="false"
    createConnector="false" />
</managementContext>
...
</broker>
```

Statistics Collected by JMX

Broker statistics

[Table 13.2 on page 108](#) describes the statistics collected for a broker.

Table 13.2. Broker JMX Statistics

Name	Description
BrokerId	Specifies the broker's unique ID.
BrokerName	Specifies the broker's name.
BrokerVersion	Specifies the version of the broker.
DataDirectory	Specifies the pathname of the broker's data directory.
TotalEnqueueCount	Specifies the total number of messages that have been sent to the broker.
TotalDequeueCount	Specifies the number of messages that have been acknowledged on the broker.
TotalConsumerCount	Specifies the number of message consumers subscribed to destinations on the broker.
TotalProducerCount	Specifies the number of message producers active on destinations on the broker.
TotalMessageCount	Specifies the number of unacknowledged messages on the broker.
MemoryLimit	Specifies the memory limit, in bytes, used for holding undelivered messages before paging to temporary storage.
MemoryPercentageUsed	Specifies the percentage of available memory in use.
StoreLimit	Specifies the disk space limit, in bytes, used for persistent messages before producers are blocked.
StorePercentageUsed	Specifies the percentage of the store space in use.
TempLimit	Specifies the disk space limit, in bytes, used for non-persistent messages and temporary data before producers are blocked.
TempPercentageUsed	Specifies the percentage of available temp space in use.

Destination statistics

[Table 13.3 on page 109](#) describes the statistics collected for a destination.

Table 13.3. Destination JMX Statistics

Name	Description
BlockedProducerWarningInterval	Specifies, in milliseconds, the interval between warnings issued when a producer is blocked from adding messages to the destination.
MemoryLimit	Specifies the memory limit, in bytes, used for holding undelivered messages before paging to temporary storage.
MemoryPercentageUsed	Specifies the percentage of available memory in use.
MaxPageSize	Specifies the maximum number of messages that can be paged into the destination.
CursorFull	Specifies if the cursor has reached its memory limit for paged messages.
CursorMemoryUsage	Specifies, in bytes, the amount of memory the cursor is using.
CursorPercentUsage	Specifies the percentage of the cursor's available memory is in use.
EnqueueCount	Specifies the number of messages that have been sent to the destination.
DequeueCount	Specifies the number of messages that have been acknowledged and removed from the destination.
DispatchCount	Specifies the number of messages that have been delivered to consumers, but not necessarily acknowledged by the consumer.
InFlightCount	Specifies the number of dispatched to, but not acknowledged by, consumers.
ExpiredCount	Specifies the number of messages that have expired in the destination.
ConsumerCount	Specifies the number of consumers that are subscribed to the destination.
QueueSize	Specifies the number of messages in the destination that are waiting to be consumed.
AverageEnqueueTime	Specifies the average amount of time, in milliseconds, that messages sat in the destination before being consumed.

Name	Description
MaxEnqueueTime	Specifies the longest amount of time, in milliseconds, that a message sat in the destination before being consumed.
MinEnqueueTime	Specifies the shortest amount of time, in milliseconds, that a message sat in the destination before being consumed.
MemoryUsagePortion	Specifies the portion of the broker's memory limit used by the destination.
ProducerCount	Specifies the number of producers connected to the destination.

Subscription statistics

[Table 13.4 on page 110](#) describes the statistics collected for a subscription.

Table 13.4. Connection JMX Statistics

Name	Description
EnqueueCounter	Counts the number of messages that matched the subscription.
DequeueCounter	Counts the number of messages were sent to and acknowledge by the client.
DispatchedQueueSize	Specifies the number of messages dispatched to the client and are awaiting acknowledgement.
DispatchedCounter	Counts the number of messages that have been sent to the client.
MessageCountAwaitingAcknowledge	Specifies the number of messages dispatched to the client and are awaiting acknowledgement.
Active	Specifies if the subscription is active.
PendingQueueSize	Specifies the number of messages pending delivery.
PrefetchSize	Specifies the number of messages to pre-fetch and dispatch to the client.
MaximumPendingMessageLimit	Specifies the maximum number of pending messages allowed.

Managing the Broker with JMX

Overview

The MBeans exposed by Fuse MQ Enterprise provide a number of operations for monitoring and managing a broker instance. You can access these operations through any tool that supports JMX.

Broker actions

[Table 13.5 on page 111](#) describes the operations exposed by the MBean for a broker.

Table 13.5. Broker MBean Operations

Operation	Description
<code>void start();</code>	Starts the broker. In reality this operation is not useful because you cannot access the MBeans if the broker is stopped.
<code>void stop();</code>	Forces a broker to shut down. There is no guarantee that all messages will be properly recorded in the persistent store.
<code>void stopGracefully(String queueName);</code>	Checks that all listed queues are empty before shutting down the broker.
<code>void enableStatistics();</code>	Activates the broker's statistics plug-in.
<code>void resetStatistics();</code>	Resets the data collected by the statistics plug-in.
<code>void disableStatistics();</code>	Deactivates the broker's statistics plug-in.
<code>String addConnector(String URI);</code>	Adds a transport connector to the broker and starts it listening for incoming client connections and returns the name of the connector.
<code>boolean removeConnector(String connectorName);</code>	Deactivates the specified transport connector and removes it from the broker.
<code>String addNetworkConnector(String URI);</code>	Adds a network connector to the specified broker and returns the name of the connector.
<code>boolean removeNetworkConnector(String connectorName);</code>	Deactivates the specified connector and removes it from the broker.
<code>void addTopic(String name);</code>	Adds a topic destination to the broker.
<code>void addQueue(String name);</code>	Adds a queue destination to the broker.

Operation	Description
<code>void removeTopic(String name);</code>	Removes the specified topic destination from the broker.
<code>void removeQueue(String name);</code>	Removes the specified queue destination from the broker.
<code>ObjectName createDurableSubscriber(String clientId, String subscriberId, String topicName, String selector);</code>	Creates a new durable subscriber.
<code>void destroyDurableSubscriber(String clientId, String subscriberId);</code>	Destroys a durable subscriber.
<code>void gc();</code>	Runs the JVM garbage cleaner.
<code>void terminateJVM(int exitCode);</code>	Shuts down the JVM.
<code>void reloadLog4jProperties();</code>	Reloads the logging configuration from <code>log4j.properties</code> .

Connector actions

[Table 13.6 on page 112](#) describes the operations exposed by the MBean for a transport connector.

Table 13.6. Connector MBean Operations

Operation	Description
<code>void start();</code>	Starts the transport connector so that it is ready to receive connections from clients.
<code>void stop();</code>	Closes the transport connection and disconnects all connected clients.
<code>int connectionCount();</code>	Returns the number of open connections using the connector.
<code>void enableStatistics();</code>	Enables statistics collection for the connector.
<code>void resetStatistics();</code>	Resets the statistics collected for the connector.

Operation	Description
<code>void disableStatistics();</code>	Deactivates the collection of statistics for the connector.

Network connector actions

[Table 13.7 on page 113](#) describes the operations exposed by the MBean for a network connector.

Table 13.7. Network Connector MBean Operations

Operation	Description
<code>void start();</code>	Starts the network connector so that it is ready to communicate with other brokers in a network of brokers.
<code>void stop();</code>	Closes the network connection and disconnects the broker from any brokers that used the network connector to form a network of brokers.

Queue actions

[Table 13.8 on page 113](#) describes the operations exposed by the MBean for a queue destination.

Table 13.8. Queue MBean Operations

Operation	Description
<code>CompositeData getMessage(String messageId);</code>	Returns the specified message from the queue without moving the message cursor.
<code>void purge();</code>	Deletes all of the messages from the queue.
<code>boolean removeMessage(String messageId);</code>	Deletes the specified message from the queue.
<code>int removeMatchingMessages(String selector);</code>	Deletes the messages matching the selector from the queue and returns the number of messages deleted.
<code>int removeMatchingMessages(String selector, int maxMessages);</code>	Deletes up to the maximum number of messages that match the selector and returns the number of messages deleted.
<code>boolean copyMessageTo(String messageId, String destination);</code>	Copies the specified message to a new destination.
<code>int copyMatchingMessagesTo(String selector, String destination);</code>	Copies the messages matching the selector and returns the number of messages copied.

Operation	Description
<code>int copyMatchingMessagesTo(String selector, String destination, int maxMessages);</code>	Copies up to the maximum number of messages that match the selector and returns the number of messages copied.
<code>boolean moveMessageTo(String messageId, String destination);</code>	Moves the specified message to a new destination.
<code>int moveMatchingMessagesTo(String selector, String destination);</code>	Moves the messages matching the selector and returns the number of messages moved.
<code>int moveMatchingMessagesTo(String selector, String destination, int maxMessages);</code>	Moves up to the maximum number of messages that match the selector and returns the number of messages moved.
<code>boolean retryMessage(String messageId);</code>	Moves the specified message back to its original destination.
<code>int cursorSize();</code>	Returns the number of messages available to be paged in by the cursor.
<code>boolean doesCursorHaveMessagesBuffered();</code>	Returns <code>true</code> if the cursor has buffered messages to be delivered.
<code>boolean doesCursorHaveSpace();</code>	Returns <code>true</code> if the cursor has memory space available.
<code>CompositeData[] browse();</code>	Returns all messages in the queue, without changing the cursor, as an array.
<code>CompositeData[] browse(String selector);</code>	Returns all messages in the queue that match the selector, without changing the cursor, as an array.
<code>TabularData browseAsTable(String selector);</code>	Returns all messages in the queue that match the selector, without changing the cursor, as a table.
<code>TabularData browseAsTable();</code>	Returns all messages in the queue, without changing the cursor, as a table.
<code>void resetStatistics();</code>	Resets the statistics collected for the queue.
<code>java.util.List browseMessages(String selector);</code>	Returns all messages in the queue that match the selector, without changing the cursor, as a list.
<code>java.util.List browseMessages();</code>	Returns all messages in the queue, without changing the cursor, as a list.
<code>String sendTextMessage(String body, String username, String password);</code>	Send a text message to a secure queue.

Operation	Description
<code>String sendTextMessage(String body);</code>	Send a text message to a queue.

Topic actions

[Table 13.9 on page 115](#) describes the operations exposed by the MBean for a topic destination.

Table 13.9. Topic MBean Operations

Operation	Description
<code>CompositeData[] browse();</code>	Returns all messages in the topic as an array.
<code>CompositeData[] browse(String selector);</code>	Returns all messages in the topic that match the selector as an array.
<code>TabularData browseAsTable(String selector);</code>	Returns all messages in the topic that match the selector as a table.
<code>TabularData browseAsTable();</code>	Returns all messages in the topic as a table.
<code>void resetStatistics();</code>	Resets the statistics collected for the queue.
<code>java.util.List browseMessages(String selector);</code>	Returns all messages in the topic that match the selector as a list.
<code>java.util.List browseMessages();</code>	Returns all messages in the topic as a list.
<code>String sendTextMessage(String body, String username, String password);</code>	Send a text message to a secure topic.
<code>String sendTextMessage(String body);</code>	Send a text message to a topic.

Subscription actions

[Table 13.10 on page 115](#) describes the operations exposed by the MBean for a durable subscription.

Table 13.10. Subscription MBean Operations

Operation	Description
<code>void destroy();</code>	Destroys the subscription.

Operation	Description
<code>CompositeData[] browse();</code>	Returns all messages waiting for the subscriber.
<code>TabularData browseAsTable();</code>	Returns all messages waiting for the subscriber.
<code>int cursorSize();</code>	Returns the number of messages available to be paged in by the cursor.
<code>boolean doesCursorHaveMessagesBuffered();</code>	Returns <code>true</code> if the cursor has buffered messages to be delivered.
<code>boolean doesCursorHaveSpace();</code>	Returns <code>true</code> if the cursor has memory space available.
<code>boolean isMatchingQueue(String queueName);</code>	Returns <code>true</code> if this subscription matches the given queue name.
<code>boolean isMatchingTopic(String topicName);</code>	Returns <code>true</code> if this subscription matches the given topic name.

Chapter 14. Applying Patches

Fuse MQ Enterprise supports incremental patching. Red Hat will supply you with easy to install patches that only make targeted changes to a deployed broker.

Patching a Standalone Broker	118
Patching a Broker in a Fabric	120

Incremental patching allows you apply targeted fixes to a broker without needing to reinstall an updated version of Fuse MQ Enterprise. It also allows you to easily back the patch out if it causes problems with your deployed applications.

Patches are ZIP files that contain the artifacts needed to update a targeted set of bundles in a container. The patch file includes a `.patch` file that lists the contained artifacts. The artifacts are typically one or more bundles. They can, however, include configuration files and feature descriptors.

You get a patch file in one of the following ways:

- Customer Support sends you a patch.
- Customer Support sends you a link to download a patch.

The process of applying a patch to a broker depends on how the broker is deployed:

- standalone—the broker's command console's **patch** shell has commands for managing the patching process
- fabric—patching a fabric requires applying the patch to a profile and then applying the profile to a broker

Fuse Management Console is the recommended way to patch brokers in a fabric.

Patching a Standalone Broker

Overview

Patching a standalone broker directs the broker to load the patch versions of artifacts instead of the non-patch versions. The **patch** shell provides commands to patches to the broker's environment, see which bundles are effected by applying the patch, apply the patch to the container, and back the patch out if needed.

To make sure that the a patch can be rolled back Fuse MQ Enterprise applies the patch in a non-destructive manner. The patching process does not overwrite the artifacts included in the original installation. The patched artifacts are placed in the broker's `system` folder. When the patch is applied, the broker's configuration is changed so that it points to the patched artifacts instead of the artifacts from the original installation. This makes it easy for the system to be restored to its original state or to selectively back out patches.



Important

Patches **do not** persist across installations. If you delete and reinstall a Fuse MQ Enterprise instance you will need to download the patches and reapply them.

Applying a patch

To apply a patch to a standalone broker:

1. Add the patch to the broker's environment using the **patch:add** command.

[Example 14.1 on page 118](#) shows the command for adding the patch contained in the patch file `patch.zip` from the local file system.

Example 14.1. Adding a Patch to a Broker's Environment

```
FuseMQ> patch:add file://patch.zip
```

This command copies the specified patch file to the broker's `system` folder and unpacks it.

2. Simulate installing the patch using the **patch:simulate** command.

This will generate a log of the changes that will be made to the broker when the patch is installed, but will not make any actual changes to the broker.



Tip

The **patch:list** command will display a list of all patches added to the broker's `system` folder.

3. Review the simulation log to understand the changes that will be made to the broker.
4. Apply the patch to the broker using the **patch:install** command.



Tip

The **patch:list** command will display a list of all patches added to the broker's `system` folder.

The broker will need to restart to apply the patch. If you are using a remote console, you will lose the connection to the broker. If you are using the broker's local console, it will automatically reconnect when the broker restarts.

Rolling back a patch

Occasionally a patch will not work or introduce new issues to a broker. In these cases you can easily back the patch out of the system and restore it pre-patch behavior using the **patch:rollback** command. As shown in [Example 14.2 on page 119](#), the command takes the name of patch to be backed out.

Example 14.2. Rolling Back a Patch

```
FuseMQ> patch:rollback patch1
```



Tip

The **patch:list** command will display a list of all patches added to the broker's `system` folder.

The broker will need to restart to rollback the patch. If you are using a remote console, you will lose the connection to the broker. If you are using the broker's local console, it will automatically reconnect when the broker restarts.

Patching a Broker in a Fabric

Overview

The bundles loaded by a broker in a fabric is controlled by the broker's Fabric Agent. The agent inspects the profiles applied to the broker to determine what bundles to load, and the version of each bundle, and then loads the specified version of each bundle for the broker.

A patch typically includes a new version of one or more bundles, so to apply the patch to broker in a fabric you need to update the profiles applied to it. This will cause the Fabric Agent to load the patched versions of the bundles.

Fuse Management Console is the recommended tool for patching brokers in a fabric. However, the command console's **fabric** shell also provides the commands needed to patch brokers running in a fabric.

Procedure

Patching a broker in a fabric involves:

1. Getting a patch file.
    - Customer Support sends you a patch.
    - Customer Support sends you a link to download a patch.
  2. Uploading one or more patch files to the fabric's Maven repository.
  3. Applying the patch(es) to a profile version.

This creates a new profile version that points to the new versions of the patched bundles and repositories.
  4. Migrate one or two brokers to the patched profile version to ensure that the patch does not introduce any new issues.
  5. After you are certain that the patch works, migrate the remaining brokers in the fabric to the patched version.
-

Using Fuse Management Console

Fuse Management Console is the easiest and most verbose method of patching brokers in a fabric. The Fuse Management Console **Patching** tab uploads patches to a fabric's Maven repository and applies the patch to a specified

profile version. You can then use Fuse Management Console to roll the patch out to all of the containers in the fabric.

Using the command console

The Fuse MQ Enterprise command console can also be used to patch brokers running in a fabric. To patch a fabric broker:

1. Upload the patch file to the fabric's Maven repository.
2. Create a new profile version to which the patch will be applied.
3. Modify all of the profiles in the new version which require the patch.
4. Use the **fabric:container-upgrade** command to roll the patch out to the brokers running in the fabric.

Appendix A. Required JARS

Overview

To simplify deploying Fuse MQ Enterprise it is recommended that you place the `activemq-all.jar` file on the broker's `CLASSPATH`. It contains all of the classes needed by a message broker. This is the default set up for a Fuse MQ Enterprise installation.

However, if you want more control over the JARs in the broker's `CLASSPATH` you can add the individual JARs. There are several JARs that are required. In addition, there are a few that are only needed when certain features are used.

Required JARs from Fuse MQ Enterprise

The following JARs are installed with Fuse MQ Enterprise and must be placed on the broker's `CLASSPATH`:

- `activemq-core.jar`
  - `activeio-core.jar`
  - `slf4j-api.jar`
-

JEE JARs

The JARs containing the JEE APIs are also required by the broker. These could be located in one of the following locations:

- the `jee.jar` from Oracle
- your JEE container's installation
- the Geronimo specs JARs:
  - `geronimo-spec-jms.jar`
  - `geronimo-spec-jta.jar`

- `geronimo-spec-j2ee-management.jar`

Feature specific JARs

If you want to use persistent messaging you will need to add JARs to the broker's `CLASSPATH` for the desired persistence manager.

For KahaDB you will need `kahadb.jar`.

For JDBC you will need the JARs for your database's JDBC driver.

Index

A

- Active, 110
- activemq.xml, 22
- administration client
  - running, 71
- administration console
  - port number, 36
  - securing, 37
  - url, 36
- Aries
  - namespaces, 65
  - placeholder extension, 66
- authentication, 63
- AverageEnqueueTime, 109

B

- BlockedProducerWarningInterval, 109
- broker
  - addConnector, 111
  - addNetworkConnector, 111
  - addQueue, 111
  - addTopic, 111
  - createDurableSubscriber, 112
  - deploying
    - fabric container, 77
  - destroyDurableSubscriber, 112
  - disableStatistics, 111
  - enableStatistics, 111
  - gc, 112
  - reloadLog4jProperties, 112
  - removeConnector, 111
  - removeNetworkConnector, 111
  - removeQueue, 112
  - removeTopic, 112
  - resetStatistics, 111
  - start, 111
  - stop, 111
  - stopGracefully, 111
  - terminateJVM, 112

- useJmx, 106
- BrokerId, 108
- BrokerName, 108
- BrokerVersion, 108

C

- client, 71
- command console
  - getting help, 72
  - remote access, 74
- config shell, 22
- config.properties, 100
- configuration
  - persistent identifier, 21
  - PID, 21
  - template, 19
- connection.password, 62
- connection.url, 61
- connection.username, 62
- connector
  - connectionCount, 112
  - disableStatistics, 113
  - enableStatistics, 112
  - resetStatistics, 112
  - start, 112
  - stop, 112
- connectorHost, 107
- connectorPath, 107
- connectorPort, 107
- console
  - config shell, 22
- console mode
  - starting, 51
  - stopping, 82
- ConsumerCount, 109
- createConnector, 107
- createMBeanServer, 107
- CursorFull, 109
- CursorMemoryUsage, 109
- CursorPercentUsage, 109

D

- daemon mode

- starting, 52
  - stopping, 82
- DataDirectory, 108
- deploying
  - standalone broker, 76
- DequeueCount, 109
- DequeueCounter, 110
- DispatchCount, 109
- DispatchedCounter, 110
- DispatchedQueueSize, 110

E

- embedded console
  - disabling, 41
- encryptor, 66
- EnqueueCount, 109
- EnqueueCounter, 110
- ExpiredCount, 109

F

- fabric
  - profiles, 24
  - starting a broker, 52
  - stopping a broker, 85
  - versions, 24
- fabric shell, 77
- fabric:container-change-profile, 90
- fabric:container-connect, 74
- fabric:container-start, 52
- fabric:container-stop, 85
- fabric:container-upgrade, 121
- fabric:create, 95
- fabric:join, 88
  - n, 93
  - p, 89
- fabric:mq-create, 77
- features:install, 68
- Fuse HQ, 14
- Fuse Management Console, 14
- fusemq, 51

I

- InFlightCount, 109
- initial.context.factory, 63

J

- JAAS
  - configuration syntax, 56
  - converting to blueprint, 58
  - namespace, 56
- jaas:config, 57
- jaas:module, 57
- Jasypt
  - configuration, 66
  - libraries, 68
  - namespaces, 65
- jasypt-encryption, 68
- jconsole, 14
- JMX
  - disabling, 106
  - roles, 69, 106
- jmxDomainName, 107
- joining a fabric
  - managed container, 89
  - non-managed container, 93

K

- KARAF_BASE, 44
- KARAF_DATA, 44
- KARAF_HOME, 44

L

- LDAP
  - authentication, 63
  - configuration, 61
  - connection.password, 62
  - connection.url, 61
  - connection.username, 62
  - enabling, 61
  - initial.context.factory, 63
  - properties, 61
  - role.base.dn, 62
  - role.filter, 62

- role.name.attribute, 63
- role.search.subtree, 63
- ssl, 63
- ssl.algorithm, 63
- ssl.keyalias, 63
- ssl.keystore, 63
- ssl.protocol, 63
- ssl.provider, 63
- ssl.truststore, 64
- user.base.dn, 62
- user.filter, 62
- user.search.subtree, 62
- LDAPLoginModule, 61
- logging
  - console commands, 103-104
  - viewing as text, 103
  - viewing in an editor, 103
  - viewing in the console, 103
  - viewing with the admin client, 104

M

- managed container, 89
- managementContext, 106
  - connectorHost, 107
  - connectorPath, 107
  - connectorPort, 107
  - createConnector, 107
  - createMBeanServer, 107
  - jmxDomainName, 107
  - rmiServerPort, 107
  - useMBeanServer, 106
- MaxEnqueueTime, 110
- MaximumPendingMessageLimit, 110
- MaxPageSize, 109
- MemoryLimit, 108-109
- MemoryPercentageUsed, 108-109
- MemoryUsagePortion, 110
- MessageCountAwaitingAcknowledge, 110
- MinEnqueueTime, 110
- mq-create, 77

N

- namespaces

- Aries, 65
- Jasypt, 65
- network connector
  - start, 113
  - stop, 113
- non-managed container, 93

O

- org.apache.karaf.log, 100
- org.ops4j.pax.logging, 100
- org.ops4j.pax.logging.DefaultServiceLog.level, 100
- osgi:shutdown, 84

P

- patch:add, 118
- patch:install, 118
- patch:list, 118-119
- patch:rollback, 119
- patch:simulate, 118
- patching
  - fabric
    - command console, 121
  - Fuse Management Console, 120
  - standalone, 118
  - rollback, 119
- PendingQueueSize, 110
- persistent identifier, 21
- PID, 21
- PrefetchSize, 110
- ProducerCount, 110
- profile
  - creating, 77
- properties
  - Apache Karaf placeholder extension, 66
  - Aries placeholder extension, 66
  - encrypted, 65
  - LDAP, 61
  - placeholder, 67
  - property-placeholder, 66

Q

- queue

- browse, 114
- browseAsTable, 114
- browseMessages, 114
- copyMatchingMessagesTo, 113-114
- copyMessageTo, 113
- cursorSize, 114
- doesCursorHaveMessagesBuffered, 114
- doesCursorHaveSpace, 114
- getMessage, 113
- moveMatchingMessagesTo, 114
- moveMessageTo, 114
- purge, 113
- removeMatchingMessages, 113
- removeMessage, 113
- resetStatistics, 114
- retryMessage, 114
- sendTextMessage, 114-115
- QueueSize, 109

R

- remote console
  - roles, 69
- rmiServerPort, 107
- role.base.dn, 62
- role.filter, 62
- role.name.attribute, 63
- role.search.subtree, 63
- roles
  - default, 69
  - JMX, 69, 106
  - LDAP configuration, 61
  - remote console, 69
- routine tasks, 13
- routing console
  - securing, 37

S

- service wrapper
  - classpath, 45
  - JMX configuration, 46
  - JVM properties, 45
  - logging, 46
- shell, 74

- shutdown, 82
- ssh:ssh, 74, 84
- ssl, 63
  - ssl.algorithm, 63
  - ssl.keyalias, 63
  - ssl.keystore, 63
  - ssl.protocol, 63
  - ssl.provider, 63
  - ssl.truststore, 64
- SslSelectChannelConnector, 38
- standalone broker
  - configuration template, 22
  - deploying, 76
  - runtime configuration, 22
- start, 52
- stop, 82
- StoreLimit, 108
- StorePercentageUsed, 108
- subscription
  - browse, 116
  - browseAsTable, 116
  - cursorSize, 116
  - destory, 115
  - doesCursorHaveMessagesBuffered, 116
  - doesCursorHaveSpace, 116
  - isMatchingQueue, 116
  - isMatchingTopic, 116
- system service
  - Redhat, 48
  - Ubuntu, 48
  - Windows, 48

T

- TempLimit, 108
- TempPercentageUsed, 108
- tooling, 14
- topic
  - browse, 115
  - browseAsTable, 115
  - browseMessages, 115
  - resetStatistics, 115
  - sendTextMessage, 115
- TotalConsumerCount, 108

TotalDequeueCount, 108
TotalEnqueueCount, 108
TotalMessageCount, 108
TotalProducerCount, 108

U

useJmx, 106
useMBeanServer, 106
user.base.dn, 62
user.filter, 62
user.search.subtree, 62

V

VisualVM, 14

W

Web console
    accessing, 36
    basic authentication, 38
    clusters, 42
    port number, 36
    roles, 38
    securing, 37
    SSL, 38
    user credentials, 38
webconsole.jms.url, 41-42
webconsole.jmx.password, 41
webconsole.jmx.url, 41-42
webconsole.jmx.user, 41

