Red Hat AMQ 6.3

Managing and Monitoring a Broker

Administrative tasks made simple

Last Updated: 2020-10-23

Red Hat AMQ 6.3 Managing and Monitoring a Broker

Administrative tasks made simple

JBoss A-MQ Docs Team
Content Services
fuse-docs-support@redhat.com

Legal Notice

Abstract

Red Hat AMQ provides many tools to ensure that it is running at peak performance.

Table of Contents

CHAPTER 1. INTRODUCTION 4
OVERVIEW 4
ROUTINE TASKS 4
TROUBLESHOOTING 4
TOOLS 5

CHAPTER 2. EDITING A BROKER'S CONFIGURATION 6
2.1. INTRODUCTION TO BROKER CONFIGURATION 6
2.2. UNDERSTANDING THE RED HAT AMQ CONFIGURATION MODEL 6
2.3. EDITING A STANDALONE BROKER'S CONFIGURATION 8
2.4. MODIFYING A RUNNING STANDALONE BROKER'S XML CONFIGURATION 11
2.5. JVM CONFIGURATION OPTIONS. 16

CHAPTER 3. SECURITY BASICS 18
3.1. SECURITY OVERVIEW 18
3.2. BASIC SECURITY CONFIGURATION 19
3.3. ENABLE PASSWORD ENCRYPTION FOR NON-FABRIC ENVIRONMENT IN A-MQ 20
3.4. SETTING UP SSL FOR A-MQ 21
3.5. ENABLE BROKER-TO-BROKER AUTHENTICATION IN A-MQ 23
3.6. DISABLING BROKER SECURITY 23

CHAPTER 4. SECURING A STANDALONE RED HAT AMQ CONTAINER 25
4.1. DEFINING JAAS REALMS 25
4.2. ENABLE LDAP AUTHENTICATION IN THE OSGI CONTAINER 28
4.3. USING ENCRYPTED PROPERTY PLACEHOLDERS 29

CHAPTER 5. SECURING FABRIC CONTAINERS 34
DEFAULT AUTHENTICATION SYSTEM 34
MANAGING USERS 34
OBFUSCATING STORED PASSWORDS 34
ENABLING LDAP AUTHENTICATION 35

CHAPTER 6. INSTALLING RED HAT AMQ AS A SERVICE 36
6.1. OVERVIEW 36
6.2. RUNNING AMQ AS A SERVICE 36
6.3. CUSTOMIZING KARAF-SERVICE.SH UTILITY 36
6.4. SYSTEMD 37
6.5. SYSV 38
6.6. SOLARIS SMF 38
6.7. WINDOWS 38

CHAPTER 7. STARTING A BROKER 40
OVERVIEW 40
STARTING IN CONSOLE MODE 40
STARTING IN DAEMON MODE 41
STARTING A BROKER IN A FABRIC 41

CHAPTER 8. SENDING COMMANDS TO THE BROKER 43
OVERVIEW 43
RUNNING THE ADMINISTRATION CLIENT 43
USING THE BROKER CONSOLE 44
CONNECTING A CONSOLE TO A REMOTE BROKER 45
STARTING A BASIC CONSOLE 45

AVAILABLE COMMANDS 45

CHAPTER 9. DEPLOYING A NEW BROKER 46
9.1. TYPE OF DEPLOYMENT 46
9.2. DEPLOYING A STANDALONE BROKER 46

CHAPTER 10. ACTIVEMQ BROKERS AND CLUSTERS 48
10.1. CREATING A SINGLE BROKER INSTANCE 48
10.2. CONNECTING TO A BROKER 50
10.3. TOPOLOGIES 50
10.4. ALTERNATIVE MASTER-SLAVE CLUSTER 56
10.5. BROKER CONFIGURATION 58

CHAPTER 11. SHUTTING DOWN A BROKER 68
11.1. SHUTTING DOWN A LOCAL BROKER 68
11.2. SHUTTING DOWN A BROKER REMOTELY 68

CHAPTER 12. ADDING CLIENT CONNECTION POINTS 72
12.1. OVERVIEW OF TRANSPORT CONNECTORS 72
12.2. ADDING A TRANSPORT CONNECTOR TO A STANDALONE BROKER 72

CHAPTER 13. ADDING A QUEUE OR A TOPIC 74
AUTOMATIC DESTINATION CREATION 74
RESTRICTING DESTINATION CREATION 74

CHAPTER 14. USING LOGGING 75
14.1. OVERVIEW OF LOGGING 75
14.2. LOGGING CONFIGURATION 75
14.3. VIEWING THE LOG 77
14.4. CHANGE LOGGING LEVEL AT RUNTIME USING JCONSOLE 78

CHAPTER 15. USING JMX 79
15.1. INTRODUCTION TO JMX 79
15.2. CONFIGURING JMX 79
15.3. STATISTICS COLLECTED BY JMX 81
15.4. MANAGING THE BROKER WITH JMX 84

CHAPTER 16. APPLYING PATCHES 91
16.1. INTRODUCTION TO PATCHING 91
16.2. FINDING THE RIGHT PATCHES TO APPLY 91
16.3. INSTALLING A ROLLUP PATCH AS A NEW INSTALLATION 93
16.4. PATCHING A STANDALONE CONTAINER 93
16.5. PATCHING STANDALONE APACHE ACTIVEMQ 98
16.6. PATCHING A FABRIC CONTAINER WITH A ROLLUP PATCH 99
16.7. PATCHING A FABRIC CONTAINER WITH AN INCREMENTAL PATCH 106

APPENDIX A. REQUIRED JARS 109
OVERVIEW 109
REQUIRED JARS 109
JEE JARS 109
PERSISTENT MESSAGING JARS 109

INDEX 110

CHAPTER 1. INTRODUCTION

Abstract

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 Red Hat AMQ 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 Red Hat AMQ.

The following tools are included with AMQ:

- administration client—a command line tool that can be used to manage a broker and do rudimentary metric reporting
- console mode—a runtime mode that presents you with a custom console that provides a number of administrative options

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

- management console—a browser based console for viewing, monitoring, and deploying a group of distributed brokers
- JBoss Operations Network—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—a visual tool integrating several command line JDK tools and lightweight profiling capabilities

CHAPTER 2. EDITING A BROKER'S CONFIGURATION

Abstract

Red Hat AMQ 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.

2.1. INTRODUCTION TO BROKER CONFIGURATION

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 the 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.

2.2. UNDERSTANDING THE RED HAT AMQ CONFIGURATION MODEL

Abstract

The broker configuration is comprised of an XML template file that provides the framework for how a broker instance is configured, a default OSGi persistent identifier, and one or more OSGi persistent identifiers created by the OSGi Admin service. The container uses the template file to seed the configuration into the broker's runtime. The properties stored in the OSGi persistent identifiers replace any property placeholders left in the template. This allows the OSGi Admin service to update the broker's configuration on the fly.

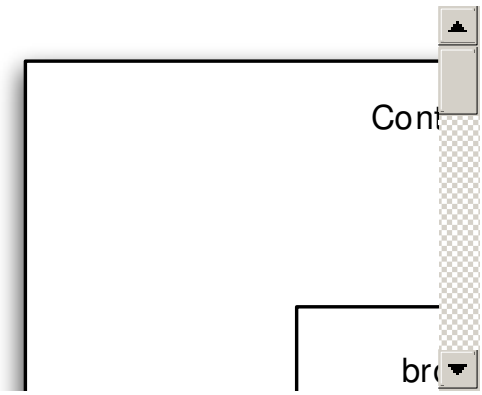
Overview

One of the weaknesses of the Apache ActiveMQ configuration model is that any changes require a broker restart. Red Hat AMQ 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 AMQ 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, “Red Hat AMQ Configuration System” , 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. Red Hat AMQ Configuration System



Configuration templates

The AMQ configuration template is an XML file that is based on the Apache ActiveMQ configuration file. The main differences between an Apache ActiveMQ and a AMQ 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
- 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, “Adding Property Placeholder Support to Red Hat AMQ Configuration”.

Example 2.1. Adding Property Placeholder Support to Red Hat AMQ 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="io.fabric8.mq.fabric.ConfigurationProperties"/>
  </property>
</bean>

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

The configuration template shown in Example 2.2, “Configuration with Property Placeholders” 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 AMQ instance have the prefix **io.fabric8.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.

2.3. EDITING A STANDALONE BROKER'S CONFIGURATION

Abstract

A standalone Red Hat AMQ message broker's configuration can be edited by directly modifying the configuration template and using the command console commands.

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/io.fabric8.mq.fabric.server-broker.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.

Splitting activemq.xml into multiple files

For complex broker configurations, you might prefer to split the **etc/activemq.xml** file into multiple XML files. You can do this using standard XML entities, declared in a *DTD internal subset*. For example, say you have an **etc/activemq.xml** file with the following outline:

```
<beans ... >
  ...
  <broker xmlns="http://activemq.apache.org/schema/core"
          brokerName="${broker-name}"
          dataDirectory="${data}"
          start="false" restartAllowed="false">

    <destinationPolicy>
      <policyMap>
        <policyEntries>
          <policyEntry topic=">" producerFlowControl="true">
            <pendingMessageLimitStrategy>
              <constantPendingMessageLimitStrategy limit="1000"/>
            </pendingMessageLimitStrategy>
          </policyEntry>
          <policyEntry queue=">" producerFlowControl="true" memoryLimit="1mb">
          </policyEntry>
        </policyEntries>
      </policyMap>
    </destinationPolicy>

    <!-- Rest of the broker configuration -->
    ...
  </broker>
</beans>
```

In this example, we assume you want to store the **destinationPolicy** element in a separate file. First of all, create a new file, **etc/destination-policy.xml**, to store the **destinationPolicy** element, with the following content:

```
<destinationPolicy>
  <policyMap>
    <policyEntries>
      <policyEntry topic=">" producerFlowControl="true">
        <pendingMessageLimitStrategy>
          <constantPendingMessageLimitStrategy limit="1000"/>
        </pendingMessageLimitStrategy>
      </policyEntry>
      <policyEntry queue=">" producerFlowControl="true" memoryLimit="1mb">
      </policyEntry>
    </policyEntries>
  </policyMap>
</destinationPolicy>
```

You can then reference and include the contents of the **etc/destination-policy.xml** file in your **etc/activemq.xml** file by editing **activemq.xml**, as follows:

```
<!DOCTYPE beans [
<!ENTITY destinationpolicy SYSTEM "file:etc/destination-policy.xml">
]>
<beans ... >
    ...
    <broker xmlns="http://activemq.apache.org/schema/core"
            brokerName="${broker-name}"
            dataDirectory="${data}"
            start="false" restartAllowed="false">

 &destinationpolicy;

        <!-- Rest of the broker configuration -->
        ...
    </broker>
</beans>
```

Where the destinationPolicy element has now been replaced by the **&destinationpolicy;** entity reference.

If you need to specify the absolute location of the **destination-policy.xml** file, use the URL format, **file:///path/to/file**. For example, to reference the absolute location, **/var/destination-policy.xml**, you would use the following **DOCTYPE** declaration at the start of the file:

```
<!DOCTYPE beans [
<!ENTITY destinationpolicy SYSTEM "file:///var/destination-policy.xml">
]>
...
```

Format of the DOCTYPE declaration

The recommended format of the **DOCTYPE** declaration to use with the **etc/activemq.xml** file is as follows:

```
<!DOCTYPE RootElement [
<!ENTITY EntityName SYSTEM "URL">
]>
```

```
...
```

Note the following points about this format:

RootElement

This *must* always match the name of the root element in the current file. In the case of **activemq.xml**, the root element is **beans**.

EntityName

The name of the entity you are defining with this ENTITY declaration. In the main part of the current XML file, you can insert the contents of this entity using the entity reference, **&*EntityName*;**.

URL

To store the contents of the entity in a file, you must reference the file using the **file:** scheme. Because of the way that ActiveMQ processes the XML file, it is not guaranteed to work, if you leave out the **file:** prefix. Relative paths have the format **file:path/to/file** and absolute paths have the format **file:///path/to/file**.

Editing the OSGi properties

The initial values for all of the OSGi properties configuring the broker are specified in the **etc/io.fabric8.mq.fabric.server-broker.cfg** file. You can edit these values using the command console's **config** shell. The PID for these values are **io.fabric8.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 Fuse Management 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

2.4. MODIFYING A RUNNING STANDALONE BROKER'S XML CONFIGURATION

Abstract

A select set of properties in a standalone Red Hat AMQ message broker's **.xml** configuration file can be modified, saved, then applied while the broker is running. This dynamic runtime configuration feature is useful when you cannot disrupt the operation of existing producers or consumers with a broker restart.

IMPORTANT

Take care when using this dynamic runtime configuration feature in production environments as only the xml is validated, and changes to the broker's configuration take effect according to the specified time interval.

Overview

You can edit a running broker's **.xml** configuration file (default is **etc/activemq.xml**) directly using an external text or xml editor. Once the edits are saved, the runtime configuration plugin, which monitors the broker's **.xml** configuration file, applies any detected runtime-supported changes to the running broker. These changes persist through broker restarts.

You can dynamically change only a select set of properties by editing the broker's **.xml** configuration file:

- network connectors—add a network connector to a broker or modify the attributes of an existing one
- virtual destinations—add a virtual destination to a broker or modify the attributes of an existing one
- destination policy—add a subset of <policyEntry> attributes
- authorization roles—add or modify roles that define read/write/admin access to queues and topics.

Prerequisites

- Disable configuration monitoring by the OSGi service factory

  You need to prevent the OSGi service factory from restarting the broker when it detects a change in the broker's configuration. To do so, you edit the *installDir*/**etc/io.fabric8.mq.fabric.server-broker.cfg** file to add the line config.check=false.

  IMPORTANT

  If you fail to disable the OSGi service factory, it will override the **runtimeConfigurationPlugin** and restart the broker when it detects a change.

  If the broker is stopped, you can edit this file directly using an external text or xml editor. If the broker is running, you must use the appropriate **config:** shell commands to edit this file.

- Enable dynamic runtime configuration

  To enable dynamic runtime configuration, you must set two values in the broker's **.xml** configuration file:

  - In the **<broker.../>** element, add **start="false"**; for example:

```
<broker xmlns="http://activemq.apache.org//schema/core" ... start="false".../>
```

This setting prevents Spring from starting the broker when the spring context is loaded. If Spring starts the broker, the broker will not know the location of the resource that created it, leaving the runtime configuration plugin with nothing to monitor.

- In the <plugins> element, add **<runtimeConfigurationPlugin checkPeriod="1000">** to enable automated runtime configuration; for example:

```
<plugins>
  <runtimeConfigurationPlugin checkPeriod="1000" />
</plugins>
```

The runtime configuration plugin monitors the broker's **.xml** configuration file at intervals of **checkPeriod** and applies only the runtime-supported changes that it detects to the running broker. Modifications made to the attributes of other properties in the broker's **.xml** configuration file are ignored until the next broker restart.

NOTE

The unit of value for **checkPeriod** is milliseconds. The default is **0**, which disables checking for changes. Using the default, you must manually trigger updates via JMX.

Dynamically updating network connectors

To dynamically update the broker's network connectors, you add a network connector or modify attributes in an existing network connector in the <networkConnectors> section of the broker's **.xml** configuration file.

For example:

```
<networkConnectors>
  <networkConnector uri="static:(tcp://localhost:5555)" networkTTL="1" name="one" ... />
</networkConnectors>
```

Dynamically updating virtual destinations

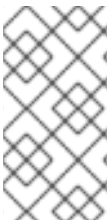
To dynamically update the broker's virtual destinations, you add a virtual destination or modify attributes in an existing virtual topic in the <destinationInterceptors> section of the broker's **.xml** configuration file.

For example:

```
<destinationInterceptors>
  <virtualDestinationInterceptor>
    <virtualDestinations>
      <virtualTopic name="B.>" selector="false" />
    </virtualDestinations>
  </virtualDestinationInterceptor>
</destinationInterceptors>
```

NOTE

Changes take effect the next time a new consumer destination is added, not at the runtime configuration plugin's **checkPeriod** interval.



NOTE

Out-of-the-box, virtual topics are enabled by default in the broker, without explicit configuration in its **.xml** configuration file. The first time you add a virtual destination, you must add the entire <destinationInterceptors> section to the broker's **.xml** configuration file. Doing so replaces the broker's default <destinationInterceptors> configuration.

Dynamically updating the destination policy

To dynamically update the broker's virtual destination policy, you edit the <destinationInterceptors> section in the broker's **.xml** configuration file.

Table 2.1 lists the runtime-changeable attributes of the **<policyEntry>** element, which apply to queues and topics.

Table 2.1. Dynamically changeable <policyEntry> attributes

Attribute	Type	Queues	Topics
allConsumersBeforeDispatchStarts	boolean	Y	N
alwaysRetroactive	boolean	Y	N
advisoryForConsumed	boolean	Y	N
advisoryForDelivery	boolean	Y	N
advisoryForDiscardingMessages	boolean	Y	N
advisoryForFastProducers	boolean	Y	N
advisoryForSlowConsumers	boolean	Y	N
advisoryWhenFull	boolean	Y	N
blockedProducerWarningInterval	long	Y	N
consumersBeforeDispatchStarts	int	Y	N
cursorMemoryHighWaterMark	int	Y	N
doOptimizeMessageStore	boolean	Y	N
gcIsInactiveDestinations	boolean	Y	N

Attribute	Type	Queues	Topics
gcWithNetworkConsumers	boolean	Y	N
inactiveTimeoutBeforeGC	long	Y	N
lazyDispatch	boolean	Y	Y
maxBrowsePageSize	int	Y	N
maxExpirePageSize	int	Y	N
maxPageSize	int	Y	N
memoryLimit	string	Y	Y
minimumMessageSize	long	Y	N
optimizedDispatch	boolean	Y	N
optimizeMessageStoreInFlightLimit	int	Y	N
producerFlowControl	boolean	Y	N
reduceMemoryFootprint	boolean	Y	N
sendAdvisoryIfNoConsumers	boolean	Y	N
storeUsageHighWaterMark	int	Y	N
strictOrderDispatch	boolean	Y	N
timeBeforeDispatchStarts	int	Y	N
useConsumerPriority	boolean	Y	N

Destination policies to control paging

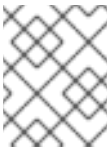
The following destination policies control message paging (the number of messages that are pulled into memory from the message store, each time the memory is emptied):

maxPageSize

The maximum number of messages paged into memory for sending to a destination.

maxBrowsePageSize

The maximum number of messages paged into memory for browsing a queue.



NOTE

The number of messages paged in for browsing cannot exceed the destination's **memoryLimit** setting.

maxExpirePageSize

The maximum number of messages paged into memory to check for expired messages.

Dynamically updating authorization roles

To dynamically add authorization roles for accessing the broker's queues and topics, you:

- add the authorization plugin to the **<plugins>** section of the broker's **.xml** configuration file
- configure the authorization plugin's **<map>** element

For example:

```
<plugins>
 <runtimeConfigurationPlugin checkPeriod="1000" />
 <authorizationPlugin>
  <map>
   <authorizationMap>
    <authorizationEntries>
     <authorizationEntry queue=">" read="admins" write="admins" admin="admins" />
     <authorizationEntry queue="USERS.>" read="users" write="users" admin="users" />

     <authorizationEntry topic=">" read="admins" write="admins" admin="admins" />
     <authorizationEntry topic="USERS.>" read="users" write="users" admin="users" />
     ...
```

2.5. JVM CONFIGURATION OPTIONS.

Abstract

Various settings for the JVM can be configured prior to startup. To do this, edit the **bin/setenv** file. The **setenv** file is used as part of the start-up routine, so for any changes to be picked up they have to be made before JBoss A-MQ is started.

Setting Java Options

Java Options can be set using the **/bin/setenv** file. Use this file to set a number of Java Options, such as JAVA_MIN_MEM, JAVA_MAX_MEM, JAVA_PERM_MEM, JAVA_MAX_PERM_MEM. These are the default options. Other Java Options can be set using the EXTRA_JAVA_OPTS variable.

For example, to allocate minimum memory for the JVM use

```
JAVA_MIN_MEM=512M # Minimum memory for the JVM
```

. To set a Java option other than the defaults, use

```
EXTRA_JAVA_OPTS="Java option"
```

. For example,

```
EXTRA_JAVA_OPTS="-XX:+UseG1GC"
```

.

CHAPTER 3. SECURITY BASICS

Abstract

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

3.1. SECURITY OVERVIEW

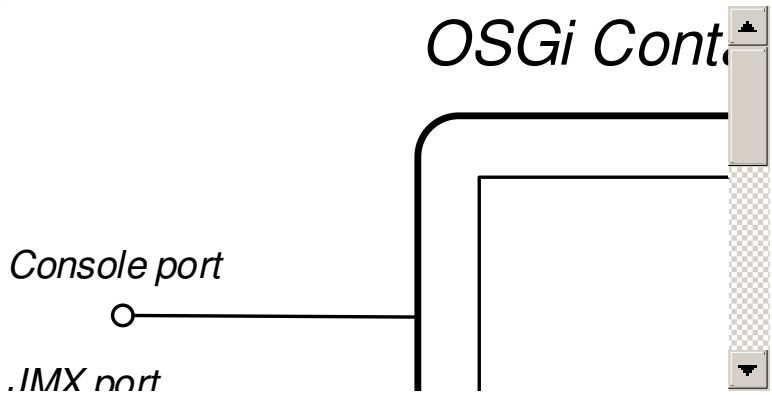
Overview

The Red Hat AMQ 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 container

Figure 3.1, "Ports Exposed by the Red Hat AMQ Container" shows the ports exposed by the AMQ container by default.

Figure 3.1. Ports Exposed by the Red Hat AMQ 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 hosts the Fuse Management Console.

Authentication and authorization system

Red Hat AMQ 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. AMQ's command console provides commands to configure the JAAS system.

3.2. BASIC SECURITY CONFIGURATION

Overview

The default security settings block access to a broker's remote ports. If you want to access the Red Hat AMQ 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 Fuse Management Console
- assign roles to each of the remote ports to limit access
- strengthen the credentials needed to access the remote console

WARNING

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

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,Administrator
```

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

Do not define a numeric username with a leading zero. Such usernames will always cause a login attempt to fail. This is because the Karaf shell, which the console uses, drops leading zeros when the input appears to be a number. For example:

```
JBossA-MQ:karaf@root> echo 0123
123
JBossA-MQ:karaf@root> echo 00.123
0.123
JBossA-MQ:karaf@root>
```

NOTE

You can also grant privileges to a user through user groups, instead of listing the roles directly.

WARNING

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

Role-based access control

The AMQ container supports role-based access control, which regulates access through the JMX protocol, the Karaf command console, and the Fuse Management console. When assigning roles to users, you can choose from the set of standard roles, which provide the levels of access described in Table 3.1, “Standard Roles for Access Control” .

Table 3.1. Standard Roles for Access Control

Roles	Description
Monitor, **Operator**, **Maintainer**	Grants read-only access to the container.
Deployer, **Auditor**	Grants read-write access at the appropriate level for ordinary users, who want to deploy and run applications. But blocks access to sensitive container configuration settings.
Administrator, **SuperUser**	Grants unrestricted access to the container.

For more details about role-based access control, see section "Role-Based Access Control" in "Security Guide".

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 the *Security Guide*.

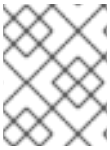
3.3. ENABLE PASSWORD ENCRYPTION FOR NON-FABRIC ENVIRONMENT IN A-MQ

Red Hat JBoss A-MQ provides a set of options for enabling password encryption. To protect the passwords, you must set the file permissions of the **users.properties** file so that it can be read only by administrators. To provide additional protection, you can also encrypt the stored passwords using a message digest algorithm.

To enable the password encryption feature using the MD algorithm, follow the below instructions:

- Edit the **InstallDir/etc/org.apache.karaf.jaas.cfg** file.
- For example, the following settings would enable basic encryption using the MD5 message digest algorithm:

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



NOTE

The encryption settings in the **org.apache.karaf.jaas.cfg** file are applied only to the default karaf realm in a standalone container.

See Also: section "Using Encrypted Property Placeholders" in "Security Guide"

3.4. SETTING UP SSL FOR A-MQ

ActiveMQ includes key and trust stores that reference a dummy self signed certificate.

To install and configure SSL support for A-MQ, you need to create a keystore file to store the server's private key and self-signed certificate and uncomment the **SSL HTTP/1.1 Connector entry in conf/server.xml**.

NOTE

When you create a broker certificate and trust stores for your installation, either overwrite the values in the **conf** directory or delete the existing dummy key and trust stores so they do not interfere.

Starting the Broker with SSL

To start the broker, use the **>javax.net.ssl.keyStore** and **javax.net.ssl.keyStorePassword** system properties

1. Set the **SSL_OPTS** environment variable so that it knows to use the broker keystore. **<export SSL_OPTS = -Djavax.net.ssl.keyStore=/path/to/broker.ks -Djavax.net.ssl.keyStorePassword=password**

   Alternately, you can set the system properties in the broker configuration file.

To configure the security context in the broker configuration file, follow the instructions below:

- In the **conf/activemq.xml**, edit the attributes in the **sslContext** element.
- Set the values for KeyStore, Key StorePassword, truststore, trustStorePassword.

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

keyStore

equivalent to setting **javax.net.ssl.keyStore**

keyStorePassword

equivalent to setting **javax.net.ssl.keyStorePassword**

keyStoreType

equivalent to setting **javax.net.ssl.keyStoreType**

keyStoreAlgorithm

defaults to JKS

trustStore

equivalent to setting **javax.net.ssl.trustStore**

trustStorePassword

equivalent to setting **javax.net.ssl.trustStorePassword**

trustStoreType

equivalent to setting **javax.net.ssl.trustStoreType**

Verifying Client Certificates

To verify client certificates, follow the below instructions:

- Export the client's certificate to share it with the broker. **keytool -export -alias client -keystore client.ks -file client_cert**
- Create a truststore for the broker and import the client's certificate. This ensures that the broker trusts the client.

```
keytool -import -alias client -keystore broker.ts -file client_cert
```

- Add **javax.net.ssl.trustStore** system property to **SSL_OPTS Djavax.net.ssl.trustStore=/path/to/broker.ts**

- Instruct ActiveMQ to require client authentication by setting the following in **activemq.xml**.

```
<transportConnectors>
  <transportConnector name="ssl" uri="ssl://localhost:61617?needClientAuth=true"/>
</transportConnectors>
```

3.5. ENABLE BROKER-TO-BROKER AUTHENTICATION IN A-MQ

To enable authentication between 2 brokers, for example Broker A and Broker B, where Broker A is configured to perform authentication, you can configure Broker B to log on to Broker A by setting the **userName** attribute and the password attribute in the **networkConnector** element.

To configure the network connector follow the below instructions:

- Assuming that Broker A is configured to connect to Broker B. Configure the Broker A's networkConnector element with username/password credentials as shown:

- For example, the following settings would enable basic encryption using the MD5 message digest algorithm:

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

  Here Broker A's authentication plug-in checks for Broker A's username. For example, if Broker A has its authentication configured by a simpleAuthenticationPlugin element, Broker A's username must appears in this element.

The encryption settings in the **org.apache.karaf.jaas.cfg** file are applied only to the default karaf realm in a standalone container.

3.6. 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 Red Hat AMQ in standalone mode (that is, running in an OSGi container, but not using Fuse Fabric). In your installation of Red Hat AMQ, 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 AMQ container using the start script the broker will run with unsecured ports.

CHAPTER 4. SECURING A STANDALONE RED HAT AMQ CONTAINER

Abstract

The Red Hat AMQ 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. Red Hat AMQ runs in an OSGi container that uses the Java Authentication and Authorization Service(JAAS) to perform authorization. Changing the authorization scheme for the container involves defining a new JAAS realm and deploying it into the container.

4.1. DEFINING JAAS REALMS

Overview

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

Namespace

The **jaas:config** element is defined in the **http://karaf.apache.org/xmlns/jaas/v1.0.0** namespace. When defining a JAAS realm you will need to include the line shown in Example 4.1, "JAAS Blueprint Namespace".

Example 4.1. 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 4.2, "Defining a JAAS Realm in Blueprint XML".

Example 4.2. Defining a JAAS Realm in Blueprint XML

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

  <jaas:config name="JaasRealmName"
               [rank="IntegerRank"]>
    <jaas:module className="LoginModuleClassName"
                 [flags="[required|requisite|sufficient|optional]"]>
      Property=Value
      ...
    </jaas:module>
```

```
        ...
        <!-- Can optionally define multiple modules -->
        ...
    </jaas:config>

</blueprint>
```

The elements are used as follows:

jaas:config

Defines the JAAS realm. It has the following attributes:

- **name**—specifies the name of the JAAS realm.
- **rank**—specifies an optional rank for resolving naming conflicts between JAAS realms . When two or more JAAS realms are registered under the same name, the OSGi container always picks the realm instance with the highest rank. If you decide to override the default realm, **karaf**, you should specify a **rank** of **100** or more, so that it overrides all of the previously installed **karaf** realms (in the context of Fabric, you need to override the default **ZookeeperLoginModule**, which has a rank of **99**).

jaas:module

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

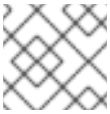
- **className**—the fully-qualified class name of a JAAS login module. The specified class must be available from the bundle classloader.
- **flags**—determines what happens upon success or failure of the login operation. Table 4.1, "Flags for Defining a JAAS Module" describes the valid values.

Table 4.1. 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

Red Hat AMQ uses the same properties as a standard Java login configuration file, however Red Hat AMQ requires that they are specified slightly differently. To see how the Red Hat AMQ approach to defining JAAS realms compares with the standard Java login configuration file approach, consider how to convert the login configuration shown in Example 4.3, “Standard JAAS Properties”, which defines the **PropertiesLogin** realm using the Red Hat AMQ properties login module class, **PropertiesLoginModule**:

Example 4.3. 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 4.4, “Blueprint JAAS Properties”.

Example 4.4. Blueprint JAAS Properties

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

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

</blueprint>
```

IMPORTANT

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

Example

Red Hat AMQ also provides an adapter that enables you to store JAAS authentication data in an X.500 server. Example 4.5, “Configuring a JAAS Realm” defines the **LDAPLogin** realm to use Red Hat AMQ's **LDAPLoginModule** class, which connects to the LDAP server located at ldap://localhost:10389.

Example 4.5. Configuring a JAAS Realm

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

  <jaas:config name="LDAPLogin" rank="200">
    <jaas:module flags="required"
      className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule">
        initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
        connection.username=uid=admin,ou=system
        connection.password=secret
        connection.protocol=
        connection.url = ldap://localhost:10389
        user.base.dn = ou=users,ou=system
        user.filter = (uid=%u)
        user.search.subtree = true
        role.base.dn = ou=users,ou=system
        role.filter = (uid=%u)
        role.name.attribute = ou
        role.search.subtree = true
        authentication = simple
    </jaas:module>
  </jaas:config>
</blueprint>
```

For a detailed description and example of using the LDAP login module, see ???.

4.2. ENABLE LDAP AUTHENTICATION IN THE OSGI CONTAINER

Overview

You can configure the OSGi container to retrieve authentication data from an LDAP directory server. The exact configuration depends on the particular directory server implementation, and on the organization of the directory information tree.

References

Detailed documentation on LDAP authentication is provided in the Security Guide, as follows:

- *LDAP Tutorial*—is provided in chapter "LDAP Authentication Tutorial" in "Security Guide" .
- *LDAPLoginModule options*—are described in detail in section "JAAS LDAP Login Module" in "Security Guide".
- *cachedLDAPAuthorizationMap options*—are described in detail in section "Cached LDAP Authorization Plug-In" in "Security Guide".

4.3. USING ENCRYPTED PROPERTY PLACEHOLDERS

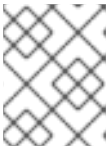
Overview

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. This feature is supported both in Blueprint XML files and in Spring XML files.

How to use encrypted property placeholders

To use encrypted property placeholders in a Blueprint XML file or in a Spring XML file, perform the following steps:

1. Download and install Jasypt, to gain access to the Jasypt **listAlgorithms.sh**, **encrypt.sh** and **decrypt.sh** command-line tools.

   NOTE

   When installing the Jasypt command-line tools, don't forget to enable execute permissions on the script files, by running **chmod u+x *ScriptName*.sh**.

2. Choose a master password and an encryption algorithm. To discover which algorithms are supported in your current Java environment, run the **listAlgorithms.sh** Jasypt command-line tool, as follows:

   ```
   ./listAlgorithms.sh
   DIGEST ALGORITHMS:   [MD2, MD5, SHA, SHA-256, SHA-384, SHA-512]

   PBE ALGORITHMS:      [PBEWITHMD5ANDDES, PBEWITHMD5ANDTRIPLEDES,
   PBEWITHSHA1ANDDESEDE, PBEWITHSHA1ANDRC2_40]
   ```

   On Windows platforms, the script is **listAlgorithms.bat**. AMQ uses **PBEWithMD5AndDES** by default.

3. Use the Jasypt encrypt command-line tool to encrypt your sensitive configuration values (for example, passwords for use in configuration files). For example, the following command encrypts the ***PlaintextVal*** value, using the specified algorithm and master password ***MasterPass***:

   ```
   ./encrypt.sh input="PlaintextVal" algorithm=PBEWithMD5AndDES password=MasterPass
   ```

4. Create a properties file with encrypted values. For example, suppose you wanted to store some LDAP credentials. You could create a file, **etc/ldap.properties**, with the following contents:

   Example 4.6. Property File with an Encrypted Property

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

The encrypted property values (as generated in the previous step) are identified by wrapping in the **ENC()** function.

5. *(Blueprint XML only)* Add the requisite namespaces to your Blueprint XML 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 4.7, "Encrypted Property Namespaces" shows a Blueprint file with the requisite namespaces.

   Example 4.7. 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>
   ```

6. Configure the location of the properties file for the property placeholder and configure the Jasypt encryption algorithm .

   - **Blueprint XML**

     Example 4.8, "Jasypt Blueprint Configuration" shows how to configure the **ext:property-placeholder** element to read properties from the **etc/ldap.properties** file. The **enc:property-placeholder** element configures Jasypt to use the **PBEWithMD5AndDES** encryption algorithm and to read the master password from the **JASYPT_ENCRYPTION_PASSWORD** environment variable.

     Example 4.8. 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:etc/ldap.properties</location>
       </ext:property-placeholder>

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

```
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>
...
</blueprint>
```

- Spring XML

  Example 4.9, "Jasypt Spring Configuration" shows how to configure Jasypt to use the **PBEWithMD5AndDES** encryption algorithm and to read the master password from the **JASYPT_ENCRYPTION_PASSWORD** environment variable.

  The **EncryptablePropertyPlaceholderConfigurer** bean is configured to read properties from the **etc/ldap.properties** file and to read properties from the **io.fabric8.mq.fabric.ConfigurationProperties** class (which defines the **karaf.base** property, for example).

  Example 4.9. Jasypt Spring Configuration

```
<bean id="environmentVariablesConfiguration"
class="org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig">
    <property name="algorithm" value="PBEWithMD5AndDES" />
    <property name="passwordEnvName"
value="JASYPT_ENCRYPTION_PASSWORD" />
</bean>

<bean id="configurationEncryptor"
class="org.jasypt.encryption.pbe.StandardPBEStringEncryptor">
    <property name="config" ref="environmentVariablesConfiguration" />
</bean>

<bean id="propertyConfigurer"
class="org.jasypt.spring31.properties.EncryptablePropertyPlaceholderConfigurer">
    <constructor-arg ref="configurationEncryptor" />
    <property name="location" value="file:${karaf.base}/etc/ldap.properties"/>
    <property name="properties">
        <bean class="io.fabric8.mq.fabric.ConfigurationProperties"/>
    </property>
</bean>
```

7. Use the placeholders in your configuration file. The placeholders you use for encrypted properties are the same as you use for regular properties. Use the syntax **${*prop.name*}**.

8. Make sure that the **jasypt-encryption** feature is installed in the container. If necessary, install the **jasypt-encryption** feature with the following console command:

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

9. Shut down the container, by entering the following command:

```
JBossFuse:karaf@root> shutdown
```

10. Carefully restart the container and deploy your secure application, as follows:

    1. Open a command window (first command window) and enter the following commands to start the AMQ container in the background:

    ```
    export JASYPT_ENCRYPTION_PASSWORD="your super secret master pass phrase"
    ./bin/start
    ```

    2. Open a second command window and start the client utility, to connect to the container running in the background:

    ```
    ./bin/client -u Username -p Password
    ```

    Where ***Username*** and ***Password*** are valid JAAS user credentials for logging on to the container console.

    3. In the second command window, use the console to install your secure application that uses encrypted property placeholders. Check that the application has launched successfully (for example, using the **osgi:list** command to check its status).

    4. After the secure application has started up, go back to the first command window and unset the **JASYPT_ENCRYPTION_PASSWORD** environment variable.

IMPORTANT

Unsetting the **JASYPT_ENCRYPTION_PASSWORD** environment variable ensures there will be minimum risk of exposing the master password. The Jasypt library retains the master password in encrypted form in memory.

Blueprint XML example

Example 4.10, "Jasypt Example in Blueprint XML" shows an example of an LDAP JAAS realm configured in Blueprint XML, using Jasypt encrypted property placeholders.

Example 4.10. Jasypt Example in Blueprint XML

```
<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>

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

```
  <jaas:config name="karaf" rank="200">
    <jaas:module className="org.apache.karaf.jaas.modules.ldap.LDAPLoginModule"
flags="required">
      initialContextFactory=com.sun.jndi.ldap.LdapCtxFactory
      debug=true
       connectionURL=${ldap.url}
       connectionUsername=cn=mqbroker,ou=Services,ou=system,dc=jbossfuse,dc=com
       connectionPassword=${ldap.password}
       connectionProtocol=
       authentication=simple
       userRoleName=cn
       userBase = ou=User,ou=ActiveMQ,ou=system,dc=jbossfuse,dc=com
       userSearchMatching=(uid={0})
       userSearchSubtree=true
       roleBase = ou=Group,ou=ActiveMQ,ou=system,dc=jbossfuse,dc=com
       roleName=cn
       roleSearchMatching= (member:=uid={1})
       roleSearchSubtree=true
    </jaas:module>
  </jaas:config>

</blueprint>
```

The **${ldap.password}** placeholder is replaced with the decrypted value of the **ldap.password** property from the **etc/ldap.properties** properties file.

CHAPTER 5. SECURING FABRIC CONTAINERS

Abstract

By default, fabric containers uses text-based username/password authentication. Setting up a more robust access control system involves creating and deploying a new JAAS realm to the containers in the fabric.

DEFAULT AUTHENTICATION SYSTEM

By default, Fabric uses a simple text-based authentication system (implemented by the JAAS login module, **io.fabric8.jaas.ZookeeperLoginModule**). This system allows you to define user accounts and assign passwords and roles to the users. Out of the box, the user credentials are stored in the Fabric registry, unencrypted.

MANAGING USERS

You can manage users in the default authentication system using the **jaas:*** family of console commands. First of all you need to attach the **jaas:*** commands to the **ZookeeperLoginModule** login module, as follows:

```
JBossFuse:karaf@root> jaas:realms
Index Realm              Module Class
    1 karaf              org.apache.karaf.jaas.modules.properties.PropertiesLoginModule
    2 karaf              org.apache.karaf.jaas.modules.publickey.PublickeyLoginModule
    3 karaf              io.fabric8.jaas.ZookeeperLoginModule
JBossFuse:karaf@root> jaas:manage --index 3
```

Which attaches the **jaas:*** commands to the **ZookeeperLoginModule** login module. You can then add users and roles, using the **jaas:useradd** and **jaas:roleadd** commands. Finally, when you are finished editing the user data, you must commit the changes by entering the **jaas:update** command, as follows:

```
JBossFuse:karaf@root> jaas:update
```

Alternatively, you can abort the pending changes by entering **jaas:cancel**.

OBFUSCATING STORED PASSWORDS

By default, the JAAS **ZookeeperLoginModule** stores passwords in plain text. You can provide additional protection to passwords by storing them in an obfuscated format. This can be done by adding the appropriate configuration properties to the **io.fabric8.jaas** PID and ensuring that they are applied to *all* of the containers in the fabric.

For more details, see section "Encrypting Stored Passwords" in "Security Guide" .

NOTE

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

ENABLING LDAP AUTHENTICATION

Fabric supports LDAP authentication (implemented by the Apache Karaf **LDAPLoginModule**), which you can enable by adding the requisite configuration to the default profile.

For details of how to enable LDAP authentication in a fabric, see chapter "LDAP Authentication Tutorial" in "Security Guide".

CHAPTER 6. INSTALLING RED HAT AMQ AS A SERVICE

Abstract

This chapter provides information on how you can start the Red Hat AMQ instance as a system service by using the templates.

NOTE

Before following these instructions, you must install Red Hat JBoss A-MQ 6.3.0 Roll Up 1.

6.1. OVERVIEW

By using the Service Script templates, you can run a AMQ instance with the help of operating system specific init scripts. You can find these templates under the **bin/contrib** directory.

6.2. RUNNING AMQ AS A SERVICE

The **karaf-service.sh** utility helps you to customize the templates. This utility will automatically identify the operating system and the default init system and generates ready to use init scripts. You can also customize the scripts to adapt them to its environment, by setting JAVA_HOME and few other environment variables.

The generated scripts are composed of two files:

1. the init script
2. the init configuration file

6.3. CUSTOMIZING KARAF-SERVICE.SH UTILITY

You can customize the **karaf-service.sh** utility, by defining an environment variable or by passing command line options:

Table 6.1.

Command Line Option	Environment Variable	Description
-k	KARAF_SERVICE_PATH	Karaf installation path
-d	KARAF_SERVICE_DATA	Karaf data path (default to **\${KARAF_SERVICE_PATH}/data**)
-c	KARAF_SERVICE_CONF	Karaf configuration file (default to **\${KARAF_SERVICE_PATH/etc/\${KARAF_SERVICE_NAME}.conf**)

Command Line Option	Environment Variable	Description
-t	KARAF_SERVICE_ETC	Karaf etc path (default to **\${KARAF_SERVICE_PATH/etc}**)
-p	KARAF_SERVICE_PIDFILE	Karaf pid path (default to **\${KARAF_SERVICE_DATA}/\${KARAF_SERVICE_NAME}.pid)**)
-n	KARAF_SERVICE_NAME	Karaf service name (default karaf)
-e	KARAF_ENV	Karaf environment variable
-u	KARAF_SERVICE_USER	Karaf user
-g	KARAF_SERVICE_GROUP	Karaf group (default **\${KARAF_SERVICE_USER**)
-l	KARAF_SERVICE_LOG	Karaf console log (default to **\${KARAF_SERVICE_DATA}/log/\${KARAF_SERVICE_NAME}-console.log**)
-f	KARAF_SERVICE_TEMPLATE	Template file to use
-x	KARAF_SERVICE_EXECUTABLE	Karaf executable name (defaul karaf support daemon and stop commands)

```
CONF_TEMPLATE="karaf-service-template.conf"
SYSTEMD_TEMPLATE="karaf-service-template.systemd"
SYSTEMD_TEMPLATE_INSTANCES="karaf-service-template.systemd-instances"
INIT_TEMPLATE="karaf-service-template.init"
INIT_REDHAT_TEMPLATE="karaf-service-template.init-redhat"
INIT_DEBIAN_TEMPLATE="karaf-service-template.init-debian"
SOLARIS_SMF_TEMPLATE="karaf-service-template.solaris-smf"
```

6.4. SYSTEMD

When the ***karaf-service.sh*** utility identifies Systemd, it generates three files:

- a systemd unit file to manage the root Apache Karaf container
- a systemd environment file with variables used by the root Apache Karaf container
- a systemd template unit file to manage Apache Karaf child containers

Here is an example:

```
$ ./karaf-service.sh -k /opt/karaf-4 -n karaf-4
Writing service file "/opt/karaf-4/bin/contrib/karaf-4.service"
Writing service configuration file "/opt/karaf-4/etc/karaf-4.conf"
Writing service file "/opt/karaf-4/bin/contrib/karaf-4@.service"
$ cp /opt/karaf-4/bin/contrib/karaf-4.service /etc/systemd/system
$ cp /opt/karaf-4/bin/contrib/karaf-4@.service /etc/systemd/system
$ systemctl enable karaf-4.service
```

6.5. SYSV

When the ***karaf-service.sh*** utility identifies a SysV system, it generates two files:

- an init script to manage the root Apache Karaf container
- an environment file with variables used by the root Apache Karaf container

Here is an example:

```
$ ./karaf-service.sh -k /opt/karaf-4 -n karaf-4
Writing service file "/opt/karaf-4/bin/contrib/karaf-4"
Writing service configuration file "/opt/karaf-4/etc/karaf-4.conf"
$ ln -s /opt/karaf-4/bin/contrib/karaf-4 /etc/init.d/
$ chkconfig karaf-4 on
```

NOTE

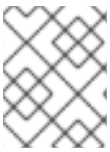
To enable the service startup upon boot, Refer your operating system init guide.

6.6. SOLARIS SMF

When the karaf-service.sh utility identifies a Solaris operating system, it generates a single file.

Here is an example:

```
$ ./karaf-service.sh -k /opt/karaf-4 -n karaf-4
Writing service file "/opt/karaf-4/bin/contrib/karaf-4.xml"
$ svccfg validate /opt/karaf-4/bin/contrib/karaf-4.xml
$ svccfg import /opt/karaf-4/bin/contrib/karaf-4.xml
```



NOTE

The generated SMF descriptor is defined as transient, so that you can execute the start method only once.

6.7. WINDOWS

Installation of Apache Karaf as windows service is supported through winsw.

To install Apache Karaf as windows service, perform the following:

- Rename the **karaf-service-win.exe** file to **karaf-4.exe** file.
- Rename the **karaf-service-win.xml** file to **karaf-4.xml** file.
- Customize the service descriptor as per your requirements.
- Use the service executable to install, start and stop the service.

Here is an example:

```
C:\opt\apache-karaf-4\bin\contrib> karaf-4.exe install
C:\opt\apache-karaf-4\bin\contrib> karaf-4.exe start
```

CHAPTER 7. STARTING A BROKER

Abstract

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

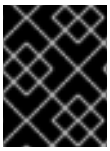
A broker can be run in one of two modes:

- 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 manged 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/io.fabric8.mq.fabric.server-broker.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 7.1, "Start up Commands for Console Mode".

Table 7.1. Start up Commands for Console Mode

Windows	**bin\amq.bat**
Linux/UNIX	**bin/amq**

If the server starts up correctly you should see something similar to Example 7.1, "Broker Console" on the console.

Example 7.1. Broker Console

```
    _ ____                        __  __  ____
   | | _ \                /\      |  \/  |/ __ \
   | | |_) | ___  ___ ___    /  \_____| \  / | | | |
_  | | _ < / _ \/ __/ __|  / /\ \_____| |\/| | | | |
| |__| | |_) | (_) \__ \__ \ / ____ \    | | | |__| |
```

```
\____/|____/ \___/|___/___/ /_/   \_\   |_| |_|\___\_\

 AMQ (6.3.0.redhat-187)
 http://www.redhat.com/products/jbossenterprisemiddleware/amq/

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.

Open a browser to http://localhost:8181 to access the management console

Hit '<ctrl-d>' or 'osgi:shutdown' to shutdown AMQ.


JBossA-MQ:karaf@root>
```

NOTE

Since version AMQ 6.2.1, launching in console mode creates two processes: the parent process **./bin/karaf**, which is executing the Karaf console; and the child process, which is executing the Karaf server in a **java** JVM. The shutdown behaviour remains the same as before, however. That is, you can shut down the server from the console using either Ctrl-D or **osgi:shutdown**, which kills both processes.

STARTING IN DAEMON MODE

Launching a broker in daemon mode runs Red Hat AMQ in the background without a console. To launch a broker in daemon mode, change to *InstallDir* and run one of the commands in Table 7.2, “Start up Commands for Daemon Mode”.

Table 7.2. Start up Commands for Daemon Mode

Windows	**bin\start.bat**
Linux/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 an 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 woul duse the command shown in Example 7.2, “Starting a Broker in a Fabric”.

  Example 7.2. Starting a Broker in a Fabric

  ```
  JBossA-MQ:karaf@root> fabric:container-start fabric-broker3
  ```

- 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 Red Hat AMQ's **bin** folder. Example 7.3, "Starting a Broker in a Fabric with the Administration Client" shows how to use the remote client to start remote broker in the fabric.

  Example 7.3. Starting a Broker in a Fabric with the Administration Client

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

- using the management console

  The management console can start and stop any of the brokers in the fabric it manages from a Web based console.

  For more information see *Using the Management Console* .

CHAPTER 8. SENDING COMMANDS TO THE BROKER

Abstract

Red Hat AMQ 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 Red Hat AMQ 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 AMQ 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
- Red Hat AMQ 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 AMQ administration client is run using the **client** in ***InstallDir*/bin**. Example 8.1, “Client Command” 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, “Administration Client Arguments” describes the command's arguments.

Table 8.1. Administration Client Arguments

Argument	Description
--help	Displays the help message.
-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.

Argument	Description
-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.
commands	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 AMQ 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, "Console Help".

Example 8.2. Console Help

```
JBossA-MQ: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.
    ...
JBossA-MQ: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, "Help for a Command", entering **admin:start --help** displays the help for that command.

Example 8.3. Help for a Command

```
JBossA-MQ: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
> JBossA-MQ: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..

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.

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

STARTING A BASIC CONSOLE

Red Hat AMQ 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

Abstract

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 Red Hat AMQ. Using a fabric, however, you can deploy multiple brokers from a single location and easily reuse large portions of the configuration.

9.1. TYPE OF DEPLOYMENT

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 Red Hat AMQ 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.

9.2. DEPLOYING A STANDALONE BROKER

Abstract

Deploying standalone brokers requires manually installing and configuring multiple instances of Red Hat AMQ.

Overview

Deploying a new standalone broker involves installing Red Hat AMQ 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 AMQ onto the target system as described in the *Installation Guide*.

2. Modify the new installation's configuration for your environment as described in Chapter 2, *Editing a Broker's Configuration*.

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*

CHAPTER 10. ACTIVEMQ BROKERS AND CLUSTERS

Abstract

Fabric provides predefined profiles for deploying a single (unclustered) broker and, in addition, you can use the powerful **fabric:mq-create** command to create and deploy clusters of brokers.

10.1. CREATING A SINGLE BROKER INSTANCE

MQ profiles

The following profiles are important for creating broker instances:

mq-base

An abstract profile, which defines some important properties and resources for the broker, but should never be used directly to instantiate a broker.

mq-default

A basic single broker, which inherits most of its properties from the **mq-base** profile.

To examine the properties defined in these profiles, you can invoke the **fabric:profile-display** command, as follows:

```
JBossFuse:karaf@root> fabric:profile-display mq-default
...
JBossFuse:karaf@root> fabric:profile-display mq-base
...
```

Creating a new broker instance

A Fuse MQ broker is a Karaf container instance running a message broker profile. The profile defines the broker dependencies (through features) and the configuration for the broker. The simplest approach to creating a new broker is to use the provided **mq-default** profile.

For example, to create a new **mq-default** broker instance called **broker1**, enter the following console command:

```
JBossFuse:karaf@root> fabric:container-create-child --profile mq-default root broker1
Creating new instance on SSH port 8102 and RMI ports 1100/44445 at:
  /Users/jdoe/Downloads/jboss-fuse-6.3.0-254/instances/broker1
The following containers have been created successfully:
   Container: broker1.
```

This command creates a new container called **broker1** with a broker *of the same name* running on it.

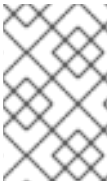
fabric:mq-create command

The **fabric:mq-create** command provides a shortcut to creating a broker, but with more flexibility, because it also creates a new profile. To create a new broker instance called **brokerx** using **fabric:mq-create**, enter the following console command:

```
JBossFuse:karaf@root> fabric:mq-create --create-container broker --replicas 1 brokerx
MQ profile mq-broker-default.brokerx ready
```

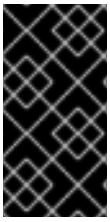
Just like the basic **fabric:container-create-child** command, **fabric:mq-create** creates a container called **broker1** and runs a broker instance on it. There are some differences, however:

- The new **broker1** container is implicitly created as a child of the current container,
- The new broker has its own profile, **mq-broker-default.brokerx**, which is based on the **mq-base** profile template,
- It is possible to edit the **mq-broker-default.brokerx** profile, to customize the configuration of this new broker.
- The **--replicas** option lets you specify the number of master/slave broker replicas (for more details, see Section 10.3.2, "Master-Slave Cluster"). In this example, we specify one replica (the default is two).



NOTE

The new profile gets the name **mq-broker-*Group.BrokerName*** by default. If you want the profile to have the same name as the broker (which was the default in AMQ version 6.0), you can specify the profile name explicitly using the **--profile** option.



IMPORTANT

The new broker is created with SSL enabled by default. The initial certificates and passwords created by default are not secure, however, and *must* be replaced by custom certificates and passwords. See the section called "Customizing the SSL keystore.jks and truststore.jks file" for details of how to do this.

Starting a broker on an existing container

The **fabric:mq-create** command can be used to deploy brokers on existing containers. Consider the following example, which creates a new Fuse MQ broker in two steps:

```
JBossFuse:karaf@root> fabric:container-create-child root broker1
Creating new instance on SSH port 8102 and RMI ports 1100/44445 at:
   /Users/jdoe/Downloads/jboss-fuse-6.3.0-254/instances/broker1
The following containers have been created successfully:
   broker1.

JBossFuse:karaf@root> fabric:mq-create --assign-container broker1 brokerx
MQ profile mq-broker-default.brokerx ready
```

The preceding example firstly creates a default child container, and then creates and deploys the new **mq-broker-default.brokerx** profile to the container, by invoking **fabric:mq-create** with the **--assign-container** option. Of course, instead of deploying to a local child container (as in this example), we could assign the broker to an SSH container.

Broker groups

Brokers created using the **fabric:mq-create** command are always registered with a specific *broker group*. If you do not specify the group name explicitly at the time you create the broker, the broker gets registered with the **default** group by default.

If you like, you can specify the group name explicitly using the **--group** option of the **fabric:mq-create** command. For example, to create a new broker that registers with the **west-coast** group, enter the following console command:

```
JBossFuse:karaf@root> fabric:mq-create --create-container broker --replicas 1 --group west-coast
brokery
MQ profile mq-broker-west-coast.brokery ready
```

If the **west-coast** group does not exist prior to running this command, it is automatically created by Fabric. Broker groups are important for defining clusters of brokers, providing the underlying mechanism for creating load-balancing clusters and master-slave clusters. For details, see Section 10.3, "Topologies".

10.2. CONNECTING TO A BROKER

Overview

This section describes how to connect a client to a broker. In order to connect to a JBoss MQ broker, you need to know its *group name*. Every MQ broker is associated with a group when it is created: if none is specified explicitly, it automatically gets associated with the **default** group.

Client URL

To connect to an MQ broker, the client must specify a discovery URL, in the following format:

```
discovery:(fabric:GroupName)
```

For example, to connect to a broker associated with the **default** group, the client would use the following URL:

```
discovery:(fabric:default)
```

The connection factory then looks for available brokers in the group and connects the client to one of them.

10.3. TOPOLOGIES

10.3.1. Load-Balancing Cluster

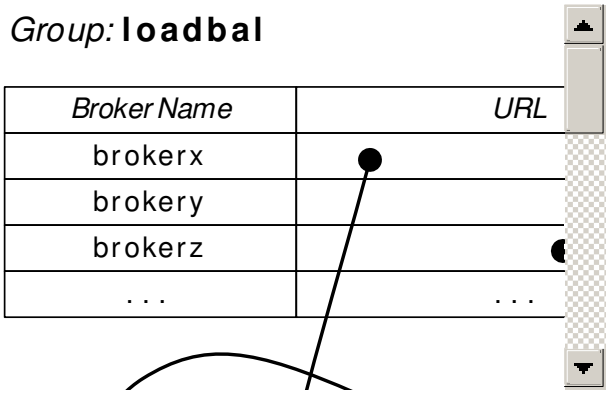
Overview

Fabric exploits the concept of broker groups to implement cluster functionality. To set up a load-balancing cluster, all of the brokers in the cluster should register with the same group name, but using *unique* broker names.

For example, Figure 10.1, "Load-Balancing Cluster" shows a load-balancing cluster with the group name, **loadbal**, and with three brokers registered in the group: **brokerx**, **brokery**, and **brokerz**. This topology is most useful in a scenario where producer is generating a heavy load but does not care if all the messages

are delivered to the consumer. In this scenario non persistent messages are used. This topology does not have shared storage. For scenarios where better guarantees regarding delivery of messages is required it is best to combine networks and master slave as defined in subsequent sections.

Figure 10.1. Load-Balancing Cluster



Create brokers in a load-balancing cluster

The basic rules for creating a load-balancing cluster of brokers are as follows:

- Choose a group name for the load-balancing cluster.
- Each broker in the cluster registers with the chosen group.
- Each broker must be identified by a *unique broker name*.
- Normally, each broker is deployed in a separate container.

For example, consider the cluster shown in Figure 10.1, "Load-Balancing Cluster". The group name is **loadbal** and the cluster consists of three broker instances with broker names: **brokerx**, **brokery**, and **brokerz**.

To create this cluster, perform the following steps:

1. First of all create some containers:

```
JBossFuse:karaf@root> container-create-child root broker 3
Creating new instance on SSH port 8102 and RMI ports 1100/44445 at:
    /Users/jdoe/Downloads/jboss-fuse-6.3.0.redhat-254/instances/broker2
Creating new instance on SSH port 8104 and RMI ports 1102/44447 at:
    /Users/jdoe/Downloads/jboss-fuse-6.3.0.redhat-254/instances/broker3
Creating new instance on SSH port 8103 and RMI ports 1101/44446 at:
    /Users/jdoe/Downloads/jboss-fuse-6.3.0.redhat-254/instances/broker1
The following containers have been created successfully:
 Container: broker2.
 Container: broker3.
 Container: broker1.
```

2. Wait until the containers are successfully provisioned. You can conveniently monitor them using the **watch** command, as follows:

```
JBossFuse:karaf@root> watch container-list
```

3. You can then assign broker profiles to each of the containers, using the **fabric:mq-create** command, as follows:

```
JBossFuse:karaf@root> mq-create --group loadbal --assign-container broker1 brokerx
MQ profile mq-broker-loadbal.brokerx ready

JBossFuse:karaf@root> mq-create --group loadbal --assign-container broker2 brokery
MQ profile mq-broker-loadbal.brokery ready

JBossFuse:karaf@root> mq-create --group loadbal --assign-container broker3 brokerz
MQ profile mq-broker-loadbal.brokerz ready
```

4. You can use the **fabric:profile-list** command to see the new profiles created for these brokers:

```
JBossFuse:karaf@root> profile-list --hidden
[id]                              [# containers] [parents]
...
mq-broker-loadbal.brokerx         1              mq-base
mq-broker-loadbal.brokery         1              mq-base
mq-broker-loadbal.brokerz         1              mq-base
mq-client-loadbal
...
```

5. You can use the **fabric:cluster-list** command to view the cluster configuration for this load balancing cluster:

```
JBossFuse:karaf@root> cluster-list
[cluster]    [masters] [slaves] [services]
...
amq/loadbal
  brokerx    broker1   -       tcp://MyLocalHost.61616 mqtt://MyLocalHost.61424
amqp://MyLocalHost.61426 stomp://MyLocalHost.61428
  brokery    broker2   -       tcp://MyLocalHost.61437 mqtt://MyLocalHost.61439
amqp://MyLocalHost.61441 stomp://MyLocalHost.61443
  brokerz    broker3   -       tcp://MyLocalHost.61453 mqtt://MyLocalHost.61455
amqp://MyLocalHost.61457 stomp://MyLocalHost.61459
```

Configure clients of a load-balancing cluster

To connect a client to a load-balancing cluster, use a URL of the form, **discovery:(fabric:*GroupName*)**, which automatically load balances the client across the available brokers in the cluster. For example, to connect a client to the **loadbal** cluster, you would use a URL like the following:

```
discovery:(fabric:loadbal)
```

For convenience, the **mq-create** command automatically generates a profile named **mq-client-*GroupName***, which provides an **ActiveMQConnectionFactory** instance in the registry. If you deploy this profile together with a Camel route that uses JMS endpoints, the Camel route will automatically find and use the **ActiveMQConnectionFactory** instance to connect to the broker cluster.

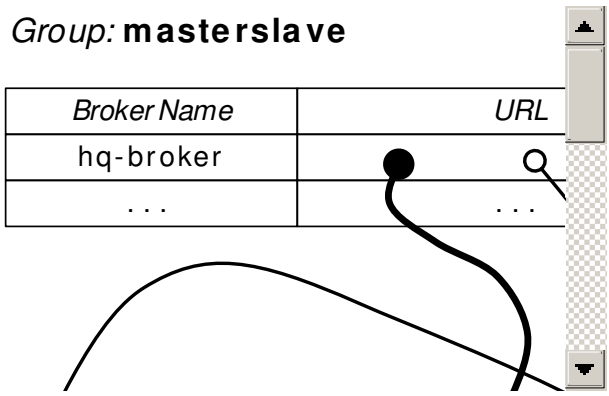
10.3.2. Master-Slave Cluster

Overview

In the master-slave pattern, multiple peer brokers provide the same service and all compete to be the master. Only *one* master can exist at a given time, while the rest remain on standby as slaves. If the master stops, the remaining brokers (slaves) compete to become the new master. If the broker containers are deployed across different machines or data centres, the result is a highly available broker.

For example, Figure 10.2, "Master-Slave Cluster" shows a master-slave cluster with the group name, **masterslave**, and three brokers that compete with each other to register as the broker, **hq-broker**. A broker becomes the master by acquiring a lock (where the lock implementation is provided by the underlying ZooKeeper registry). The other two brokers that fail to acquire the lock remain as slaves (but they continue trying to acquire the lock, at regular time intervals).

Figure 10.2. Master-Slave Cluster



Create brokers in a master-slave cluster

The basic rules for creating a master-slave cluster of brokers are as follows:

- Choose a group name for the master-slave cluster.
- Each broker in the cluster registers with the chosen group.
- Each broker must be identified by the *same* virtual broker name.
- Normally, each broker is deployed in a separate container.

For example, consider the cluster shown in Figure 10.2, "Master-Slave Cluster" . The group name is **masterslave** and the cluster consists of three broker instances, each with the *same* broker name: **hq-broker**. You can create this cluster by entering a single **fabric:mq-create** command, as follows:

```
JBossFuse:karaf@root> mq-create --create-container broker --replicas 3 --group masterslave hq-broker
```

Alternatively, if you have already created three containers, **broker1**, **broker2** and **broker3** (possibly running on separate machines), you can deploy a cluster of three brokers to the containers by entering the following command:

```
JBossFuse:karaf@root> mq-create --assign-container broker1,broker2,broker3 --group masterslave hq-broker
```

The first broker that starts becomes the master, while the others are slaves. When you stop the master, one of the slaves will take over and clients will reconnect. If brokers are persistent, you need to ensure that they all use the same store—for details of how to configure this, see the section called "Configuring persistent data".

Configure clients of a master-slave cluster

To connect a client to a master-slave cluster, use a URL of the form, **discovery:(fabric:*GroupName*)**, which automatically connects the client to the current master server. For example, to connect a client to the **masterslave** cluster, you would use a URL like the following:

```
discovery:(fabric:masterslave)
```

You can use the automatically generated client profile, **mq-client-masterslave**, to create sample clients (by referencing the corresponding **ActiveMQConnectionFactory** instance in the registry).

Locking mechanism

One benefit of this kind of master-slave architecture is that it does not depend on shared storage for locking, so it can be used even with non-persistent brokers. The broker group uses ZooKeeper to manage a shared distributed lock that controls ownership of the master status.

Re-using containers for multiple clusters

Fabric supports re-using the same containers for multiple master-slave clusters, which is a convenient way to economize on hardware resources. For example, given the three containers, **broker1**, **broker2**, and **broker3**, already running the **hq-broker** cluster, it is possible to reuse the *same* containers for another highly available broker cluster, **web-broker**. You can assign the **web-broker** profile to the existing containers with the following command:

```
mq-create --assign-container broker1,broker2,broker3 web-broker
```

This command assigns the new **web-broker** profile to the same containers already running **hq-broker**. Fabric automatically prevents two masters from running on the same container, so the master for **hq-broker** will run on a different container from the master for **web-broker**. This arrangement makes optimal use of the available resources.

Configuring persistent data

When you run a master-slave configuration with persistent brokers, it is important to specify where your store is located, because you need to be able to access it from multiple hosts. To support this scenario, the **fabric:mq-create** command enables you to specify the location of the data directory, as follows:

```
mq-create --assign-container broker1 --data /var/activemq/hq-broker hq-broker
```

The preceding command creates the **hq-broker** virtual broker, which uses the **/var/activemq/hq-broker** directory for the data (and store) location. You can then mount some shared storage to this path and share the storage amongst the brokers in the master-slave cluster.

10.3.3. Broker Networks

Overview

It is possible to combine broker clusters with broker networks, giving you a hybrid broker network that combines the benefits of broker clusters (for example, high availability) with the benefits of broker networks (managing the flow of messages between different geographical sites).

Broker networks

A *broker network* in AMQ is a form of federation where brokers are linked together using *network connectors*. This can be used as a way of forwarding messages between different geographical locations. Messages can be forwarded either *statically* (where specified categories of messages are always forwarded to a specific broker), or *dynamically* (where messages are forwarded only in response to a client that connects to a broker and subscribes to particular queues or topics).

For more details, see "Using Networks of Brokers" from the AMQ library.

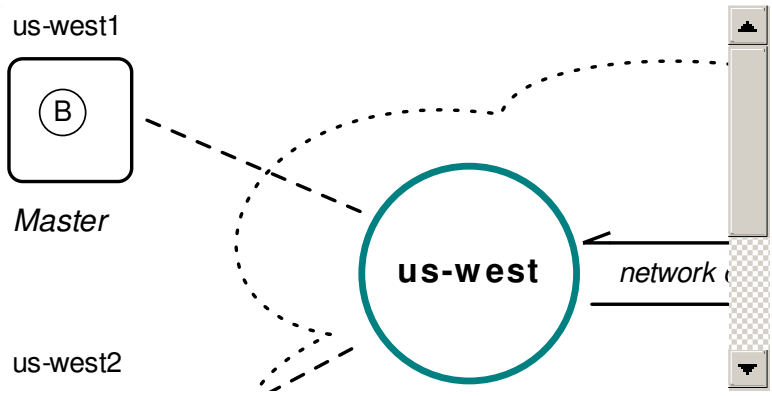
Creating network connectors

In the context of Fabric, network connectors can be created by passing the **--network** option to the **fabric:mq-create** command.

Example broker network

Consider the scenario shown in Figure 10.3, "Broker Network with Master-Slave Clusters".

Figure 10.3. Broker Network with Master-Slave Clusters



The figure shows two master-slave clusters:

- The first cluster has the group name, **us-west**, and provides high-availability with a master-slave cluster of two brokers, **us-west1** and **us-west2**.
- The second cluster has the group name, **us-east**, and provides high-availability with a master-slave cluster of two brokers, **us-east1** and **us-east2**.

Network connectors link the master brokers between each of the geographical locations (there are, in fact, two network connectors in this topology: from west to east and from east to west).

To create the pair of master-slave brokers for the **us-east** group (consisting of the two containers **us-east1** and **us-east2**), you would log on to a root container running in the US East location and enter a command like the following:

```
mq-create --group us-east --network us-west --networks-username User --networks-password Pass --create-container us-east us-east
```

Where the **--network** option specifies the name of the broker group you want to connect to, and the ***User*** and ***Pass*** are the credentials required to log on to the **us-west** broker cluster. By default, the **fabric:mq-create** command creates a master/slave pair of brokers.

And to create the pair of master-slave brokers for the **us-west** group (consisting of the two containers **us-west1** and **us-west2**), you would log on to a root container running in the US West location and enter a command like the following:

```
mq-create --group us-west --network us-east --networks-username User --networks-password Pass --create-container us-west us-west
```

Where ***User*** and ***Pass*** are the credentials required to log on to the **us-east** broker cluster.

NOTE

In a real scenario, you would probably first create the containers on separate machines and then assign brokers to the containers, using the **--assign-container** option in place of **--create-container**.

Connecting to the example broker network

At the US East location, any clients that need to connect to the broker network should use the following client URL:

```
discovery:(fabric:us-east)
```

And at the US West location, any clients that need to connect to the broker network should use the following client URL:

```
discovery:(fabric:us-west)
```

Any messages that need to be propagated between locations, from US East to US West (or from US West to US East), are transmitted over the broker network through one of the network connectors.

10.4. ALTERNATIVE MASTER-SLAVE CLUSTER

Why use an alternative master-slave cluster?

The standard master-slave cluster in Fabric uses Apache Zookeeper to manage the locking mechanism: in order to be promoted to master, a broker connects to a Fabric server and attempts to acquire the lock on a particular entry in the Zookeeper registry. If the master broker *loses* connectivity to the Fabric ensemble, it automatically becomes dormant (and ceases to accept incoming messages). A potentially undesirable side effect of this behaviour is that when you perform maintenance on the Fabric ensemble (for example, by shutting down one of the Fabric servers), you will find that the broker cluster shuts down as well.

In some deployment scenarios, therefore, you might get better up times and more reliable broker performance by disabling the Zookeeper locking mechanism (which Fabric employs by default) and using an alternative locking mechanism instead.

Alternative locking mechanism

The Apache ActiveMQ persistence layer supports alternative locking mechanisms which can be used to enable a master-slave broker cluster. In order to use an alternative locking mechanism, you need to make at least the following basic configuration changes:

1. Disable the default Zookeeper locking mechanism (which can be done by setting **standalone=true** in the broker's **io.fabric8.mq.fabric.server-*BrokerName*** PID).

2. Enable the shared file system master/slave locking mechanism in the KahaDB persistence layer (see section "Shared File System Master/Slave" in "Fault Tolerant Messaging").

NOTE

In fact, the KahaDB locking mechanism is usually enabled by default. This does not cause any problems with Fabric, because it operates at a completely different level from the Zookeeper locking mechanism. The Zookeeper coordination and locking works at the broker level to coordinate the broker start. The KahaDB lock coordinates the persistence adapter start.

standalone property

The **standalone** property belongs to the **io.fabric8.mq.fabric.server-*BrokerName*** PID and is normally used for a *non-Fabric* broker deployment (for example, it is set to **true** in the **etc/io.fabric8.mq.fabric.server-broker.cfg** file). By setting this property to **true**, you instruct the broker to stop using the discovery and coordination services provided by Fabric (but it is still possible to deploy the broker in a Fabric container). One consequence of this is that the broker stops using the Zookeeper locking mechanism. But this setting has other side effects as well.

Side effects of setting standalone=true

Setting the property, **standalone=true**, on a broker deployed in Fabric has the following effects:

- Fabric no longer coordinates the locks for the brokers (hence, the broker's persistence adapter needs to be configured as shared file system master/slave instead).
- The broker no longer uses the **ZookeeperLoginModule** for authentication and falls back to using the **PropertiesLoginModule** instead. This requires users to be stored in the **etc/users.properties** file or added to the **PropertiesLoginModule** JAAS Realm in the container where the broker is running for the brokers to continue to accept connections
- Fabric discovery of brokers no longer works (which affects client configuration).

Configuring brokers in the cluster

Brokers in the cluster must be configured as follows:

1. Set the property, **standalone=true**, in each broker's **io.fabric8.mq.fabric.server-*BrokerName*** PID. For example, given a broker with the broker name, **brokerx**, which is configured by the profile, **mq-broker-default.brokerx**, you could set the **standalone** property to **true** using the following console command:

   ```
   profile-edit --pid io.fabric8.mq.fabric.server-brokerx/standalone=true mq-broker-default.brokerx
   ```

2. To customize the broker's configuration settings further, you need to create a unique copy of the broker configuration file in the broker's own profile (instead of inheriting the broker configuration file from the base profile, **mq-base**). If you have not already done so, follow the instructions in the section called “Customizing the broker configuration file” to create a custom broker configuration file for each of the broker's in the cluster.

3. Configure each broker's KahaDB persistence adapter to use the shared file system locking mechanism. For this you must customize each broker configuration file, adding or modifying (as appropriate) the following XML snippet:

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

```
    <persistenceAdapter>
      <kahaDB directory="/sharedFileSystem/sharedBrokerData" lockKeepAlivePeriod="2000">
        <locker>
          <shared-file-locker lockAcquireSleepInterval="10000" />
        </locker>
      </kahaDB>
    </persistenceAdapter>
    ...
  </broker>
```

You can edit this profile resource either though the Fuse Management Console, through the Git configuration approach (see Section 10.5, "Broker Configuration"), or using the **fabric:profile-edit** command.

NOTE

For more details about configuring brokers, see Section 10.5, "Broker Configuration".

Configuring authentication data

When you set **standalone=true** on a broker, it can no longer use the default **ZookeeperLoginModule** authentication mechanism and falls back on the **PropertiesLoginModule**. This implies that you must populate authentication data in the **etc/users.properties** file *on each of the hosts* where a broker is running. Each line of this file takes an entry in the following format:

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

Where each entry consists of ***Username*** and ***Password*** credentials and a list of one or more roles, ***Role1***, ***Role2***,....

IMPORTANT

Using such a decentralized approach to authentication in a distributed system such as Fabric is potentially problematic. For example, if you move a broker from one host to another, the authentication data would *not* automatically become available on the new host. You should, therefore, carefully consider the impact this might have on your administrative procedures.

Configuring a client

Clients of the alternative master-slave cluster cannot use Fabric discovery to connect to the cluster. This makes the client configuration slightly less flexible, because you cannot abstract away the broker locations. In this scenario, it is necessary to list the host locations explicitly in the client connection URL.

For example, to connect to a shared file system master-slave cluster that consists of three brokers, you could use a connection URL like the following:

```
failover:(tcp://broker1:61616,tcp://broker2:61616,tcp://broker3:61616)
```

10.5. BROKER CONFIGURATION

Overview

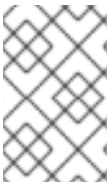
The examples presented so far have demonstrated how to create brokers with default configuration settings. In practice, you will usually need to customize the broker configurations and this can be done by editing the properties of the corresponding Fabric profiles.

Setting OSGi Config Admin properties

Many of the broker configuration settings can be altered by editing OSGi Config Admin properties (which are organized into collections identified by a *persistent ID* or PID). For example, consider the **broker1** profile created by entering the following **fabric:mq-create** command:

```
fabric:mq-create --create-container broker --replicas 1 --network us-west brokerx
```

The preceding command creates the new profile, **mq-broker-default.brokerx**, and assigns this profile to the newly created **broker1** container.



NOTE

The new profile gets the name **mq-broker-*Group.BrokerName*** by default. If you want the profile to have the same name as the broker (which was the default in AMQ version 6.0), you can specify the profile name explicitly using the **--profile** option.

You can inspect the details of the **mq-broker-default.brokerx** profile using the **fabric:profile-display** command, as follows:

```
JBossFuse:karaf@root> profile-display mq-broker-default.brokerx
Profile id: mq-broker-default.brokerx
Version   : 1.0
Attributes:
    parents: mq-base
Containers:

Container settings
----------------------------

Configuration details
----------------------------
PID: io.fabric8.mq.fabric.server-brokerx
  connectors openwire mqtt amqp stomp
  data ${runtime.data}brokerx
  standby.pool default
  keystore.file profile:keystore.jks
  kind MasterSlave
  keystore.password mca^e.Xg
  broker-name brokerx
  ssl true
  truststore.password mca^e.Xg
  keystore.cn localhost
  keystore.url profile:keystore.jks
  truststore.file profile:truststore.jks
  config profile:ssl-broker.xml
  group default
  network us-west
```

```
Other resources
---------------------------
Resource: truststore.jks
Resource: keystore.jks
```

Associated with the **io.fabric8.mq.fabric.server-brokerx** PID are a variety of property settings, such as **network** and **group**. You can now modify the existing properties or add more properties to this PID to customize the broker configuration.

Modifying basic configuration properties

You can modify the basic configuration properties associated with the **io.fabric8.mq.fabric.server-brokerx** PID by invoking the **fabric:profile-edit** command, with the appropriate syntax for modifying PID properties.

For example, to change the value of the **network** property to **us-east**, enter the following console command:

```
profile-edit --pid io.fabric8.mq.fabric.server-brokerx/network=us-east mq-broker-default.brokerx
```

Customizing the SSL keystore.jks and truststore.jks file

When using a broker with SSL security, it is necessary to replace the default keystore files with your own custom versions. The following JKS resources are stored in the **mq-broker-default.brokerx** profile when SSL is enabled (which is the default case):

keystore.jks

A Java keystore file containing this broker's *own* X.509 certificate. The broker uses this certificate to identify itself to other brokers in the network. The password for this file is stored in the **io.fabric8.mq.fabric.server-brokerx/keystore.password** property.

truststore.jks

A Java truststore file containing one or more Certificate Authority (CA) certificates or other certificates, which are used to verify the certificates presented by other brokers during the SSL handshake. The password for this file is stored in the **io.fabric8.mq.fabric.server-brokerx/truststore.password** property.

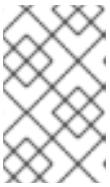
For replacing entire resource files in a profile, the easiest approach to take is to make a git clone of the profile data from the Fabric ensemble server (which also acts as a git server) and then use git to update the profile data. For more details about how to use git in Fabric, see ???.

For example, to customize the SSL settings for the **mq-broker-default.brokerx** profile, perform the following steps:

1. If you have not done so already, clone the git repository that stores all of the profile data in your Fabric. Enter a command like the following:

   ```
   git clone -b 1.0 http://Username:Password@localhost:8181/git/fabric
   cd fabric
   ```

   Where ***Username*** and ***Password*** are the credentials of a Fabric user with **Administrator** role and we assume that you are currently working with profiles in version **1.0** (which corresponds to the git branch named **1.0**).



NOTE

In this example, it is assumed that the fabric is set up to use the git cluster architecture (which is the default) and also that the Fabric server running on **localhost** is currently the *master* instance of the git cluster.

2. The **keystore.jks** file and the **truststore.jks** file can be found at the following locations in the git repository:

```
fabric/profiles/mq/broker/default.brokerx.profile/keystore.jks
fabric/profiles/mq/broker/default.brokerx.profile/truststore.jks
```

   Copy your custom versions of the **keystore.jks** file and **truststore.jks** file to these locations, over-writing the default versions of these files.

3. You also need to modify the corresponding passwords for the keystore and truststore. To modify the passwords, edit the following file in a text editor:

```
fabric/profiles/mq/broker/default.brokerx.profile/io.fabric8.mq.fabric.server-brokerx.properties
```

   Modify the **keystore.password** and **truststore.password** settings in this file, to specify the correct password values for your custom JKS files.

4. When you are finished modifying the profile configuration, commit and push the changes back to the Fabric server using git, as follows:

```
git commit -a -m "Put a description of your changes here!"
git push
```

5. For these SSL configuration changes to take effect, a restart of the affected broker (or brokers) is required. For example, assuming that the modified profile is deployed on the **broker** container, you would restart the **broker** container as follows:

```
fabric:container-stop broker
fabric:container-start broker
```

Customizing the broker configuration file

Another important aspect of broker configuration is the ActiveMQ broker configuration file, which is specified as a Spring XML file. There are two alternative versions of the broker configuration file: **ssl-broker.xml**, for an SSL-enabled broker; and **broker.xml**, for a non-SSL-enabled broker.

If you want to customize the broker configuration, it is recommended that you create a copy of the broker configuration file in your broker's own profile (instead of inheriting the broker configuration from the **mq-base** parent profile). The easiest way to make this kind of change is to use a git repository of profile data that has been cloned from a Fabric ensemble server.

For example, to customize the broker configuration for the **mq-broker-default.brokerx** profile, perform the following steps:

1. It is assumed that you have already cloned the git repository of profile data from the Fabric ensemble server (see the section called “Customizing the SSL keystore.jks and truststore.jks file”). Make sure that you have checked out the branch corresponding to the profile version that

you want to edit (which is assumed to be **1.0** here). It is also a good idea to do a git pull to ensure that your local git repository is up-to-date. In your git repository, enter the following git commands:

```
git checkout 1.0
git pull
```

2. The default broker configuration files are stored at the following location in the git repository:

```
fabric/profiles/mq/base.profile/ssl-broker.xml
fabric/profiles/mq/base.profile/broker.xml
```

Depending on whether your broker is configured with SSL or not, you should copy either the **ssl-broker.xml** file or the **broker.xml** file into your broker's profile. For example, assuming that your broker uses the **mq-broker-default.brokerx** profile and is configured to use SSL, you would copy the broker configuration as follows:

```
cp fabric/profiles/mq/base.profile/ssl-broker.xml
fabric/profiles/mq/broker/default.brokerx.profile/
```

3. You can now edit the copy of the broker configuration file, customizing the broker's Spring XML configuration as required.

4. When you are finished modifying the broker configuration, commit and push the changes back to the Fabric server using git, as follows:

```
git commit -a -m "Put a description of your changes here!"
git push
```

5. For the configuration changes to take effect, a restart of the affected broker (or brokers) is required. For example, assuming that the modified profile is deployed on the **broker** container, you would restart the **broker** container as follows:

```
fabric:container-stop broker
fabric:container-start broker
```

Additional broker configuration templates in mq-base

If you like, you can add extra broker configurations to the **mq-base** profile, which can then be used as templates for creating new brokers with the **fabric:mq-create** command. Additional template configurations must be added to the following location in the git repository:

```
fabric/profiles/mq/base.profile/
```

You can then reference one of the templates by supplying the **--config** option to the **fabric:mq-create** command.

For example, given that a new broker configuration, **mybrokertemplate.xml**, has just been installed:

```
fabric/profiles/mq/base.profile/mybrokertemplate.xml
```

You could use this custom **mybrokertemplate.xml** configuration template by invoking the **fabric:mq-create** command with the **--config** option, as follows:

```
fabric:mq-create --config mybrokertemplate.xml brokerx
```

The **--config** option assumes that the configuration file is stored in the current version of the **mq-base** profile, so you need to specify only the file name (that is, the full ZooKeeper path is not required).

Setting network connector properties

You can specify additional configuration for network connectors, where the property names have the form **network.*NetworkPropName***. For example, to add the setting, **network.bridgeTempDestinations=false**, to the PID for **brokerx** (which has the profile name, **mq-broker-default.brokerx**), enter the following console command:

```
profile-edit --pid io.fabric8.mq.fabric.server-brokerx/network.bridgeTempDestinations=false mq-broker-default.brokerx
```

The deployed broker dynamically detects the change to this property and updates the network connector on the fly.

Network connector properties by reflection

Fabric uses reflection to set network connector properties. That is, any PID property of the form **network.*OptionName*** can be used to set the corresponding ***OptionName*** property on the **org.apache.activemq.network.NetworkBridgeConfiguration** class. In particular, this implies you can set any of the following **network.*OptionName*** properties:

Property	Default	Description
name	**bridge**	Name of the network - for more than one network connector between the same two brokers, use different names
userName	*None*	Username for logging on to the remote broker port, if authentication is enabled.
password	*None*	Password for logging on to the remote broker port, if authentication is enabled.
dynamicOnly	**false**	If **true**, only activate a networked durable subscription when a corresponding durable subscription reactivates, by default they are activated on start-up.

Property	Default	Description
dispatchAsync	**true**	Determines how the network bridge sends messages to the local broker. If **true**, the network bridge sends messages asynchronously.
decreaseNetworkConsumerPriority	**false**	If **true**, starting at priority **-5**, decrease the priority for dispatching to a network Queue consumer the further away it is (in network hops) from the producer. If **false**, all network consumers use same default priority (that is, **0**) as local consumers.
consumerPriorityBase	**-5**	Sets the starting priority for consumers. This base value will be decremented by the length of the broker path when **decreaseNetworkConsumerPriority** is set.
networkTTL	**1**	The number of brokers in the network that messages and subscriptions can pass through (sets both **messageTTL** and **consumerTTL**)
messageTTL	**1**	The number of brokers in the network that messages can pass through.
consumerTTL	**1**	The number of brokers in the network that subscriptions can pass through (keep to 1 in a mesh).
conduitSubscriptions	**true**	Multiple consumers subscribing to the same destination are treated as one consumer by the network.
duplex	**false**	If **true**, a network connection is used both to produce *and* to consume messages. This is useful for hub and spoke scenarios, when the hub is behind a firewall, and so on.

Property	Default	Description
prefetchSize	**1000**	Sets the prefetch size on the network connector's consumer. It must be greater than **0**, because network consumers do not poll for messages
suppressDuplicateQueueSubscriptions	**false**	If **true**, duplicate subscriptions in the network that arise from network intermediaries are suppressed. For example, consider brokers **A**, **B**, and **C**, networked using multicast discovery. A consumer on **A** gives rise to a networked consumer on **B** and **C**. In addition, **C** networks to **B** (based on the network consumer from **A**) and **B** networks to **C**. When **true**, the network bridges between **C** and **B** (being duplicates of their existing network subscriptions to **A**) will be suppressed. Reducing the routing choices in this way provides determinism when producers or consumers migrate across the network as the potential for dead routes (stuck messages) are eliminated. The **networkTTL** value needs to match or exceed the broker count to require this intervention.
suppressDuplicateTopicSubscriptions	**true**	If **true**, duplicate network topic subscriptions (in a cyclic network) are suppressed.

Property	Default	Description
bridgeTempDestinations	**true**	Whether to broadcast advisory messages for temporary destinations created in the network of brokers. Temporary destinations are typically created for request-reply messages. Broadcasting the information about temp destinations is turned on by default, so that consumers of a request-reply message can be connected to another broker in the network and still send back the reply on the temporary destination specified in the **JMSReplyTo** header. In an application scenario where most or all of the messages use the request-reply pattern, this generates additional traffic on the broker network, because every message typically sets a unique **JMSReplyTo** address (which causes a new temp destination to be created and broadcasted with an advisory message in the network of brokers). If you disable this feature, this network traffic can be reduced, but in this case the producers and consumers of a request-reply message need to be connected to the same broker. Remote consumers (that is, connected through another broker in your network) will not be able to send the reply message, but instead will raise a **temp destination does not exist** exception.
alwaysSyncSend	**false**	If **true**, non-persistent messages are sent to the remote broker using request/reply semantics instead of oneway message semantics. This setting affects both persistent and non-persistent messages the same way.
staticBridge	**false**	If **true**, the broker does not respond dynamically to new consumers. It uses only **staticallyIncludedDestinations** to create demand subscriptions.
useCompression	**false**	Compresses the message body when sending it over the network.

Property	Default	Description
advisoryForFailedForward	**false**	If **true**, send an advisory message when the broker fails to forward the message to the temporary destination across the bridge.
useBrokerNamesAsIdSeed	**true**	Add the broker name as a prefix to connections and consumers created by the network bridge. It helps with visibility.
gcDestinationViews	**true**	If **true**, remove any MBeans for destinations that have not been used for a while.
gcSweepTime	**60000**	The period of inactivity in milliseconds, after which we remove MBeans.
checkDuplicateMessagesOnDuplex	**false**	If **true**, check for duplicates on the duplex connection.

CHAPTER 11. SHUTTING DOWN A BROKER

Abstract

Brokers can be shutdown from either the machine on which they are running or remotely from a different machine. *If the broker is running in console mode it can only be shutdown locally.*

11.1. SHUTTING DOWN A LOCAL BROKER

Abstract

Depending on how you started the local broker, you stop it using either a console command or command line tool.

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 Red Hat AMQ.

Stopping the broker from console mode

If you launched the broker by running **amq**, you shut it down using the **shutdown -f** command as shown in Example 11.1, “Using the Console's Shutdown Command” .

Example 11.1. Using the Console's Shutdown Command

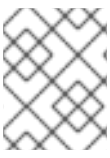
```
JBossA-MQ:karaf@root> shutdown -f
JBossA-MQ:karaf@root>
logout [Process completed]
```

NOTE

CTRL+**D** will also shutdown the broker.

Stopping a broker running in daemon mode

If you launched the broker 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.



NOTE

You can stop a broker running in daemon mode remotely. See Section 11.2, “Shutting Down a Broker Remotely”.

11.2. SHUTTING DOWN A BROKER REMOTELY

Abstract

You have a number of options for stopping a broker running on a remote machine. You can stop the broker using a console or without using a console. You can also step a broker remotely using the management console.

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. Red Hat AMQ 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 the management console—brokers that are part of a fabric can be stopped using a management console connected to their fabric

  For more information see *Using the Management Console* .

Using the stop command

You can stop a remote instance without starting up Red Hat AMQ on your local host by running the **stop** command in the ***InstallDir*/bin** directory. The commands syntax is shown in Example 11.2, "Stop Command Syntax".

Example 11.2. Stop Command Syntax

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

-a *port*

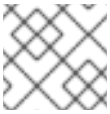
Specifies the SSH port of the remote instance. The default is 8101.

-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.

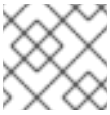


NOTE

The default username for a broker is **karaf**.

-p *password*

Specifies the password used to connect to the remote broker.



NOTE

The default password for a broker is **karaf**.

Example 11.3, "Stopping a Remote Broker" shows how to stop a remote broker on a machine named **NEBrokerHost2**.

Example 11.3. Stopping a Remote Broker

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

Using a remote console

Red Hat AMQ's console can be connected to a remote broker using the **ssh:ssh** command. The SSH port 8101 is used to access the desired remote console. Once the console is connected to the remote broker, you can shut it down by running the **osgi:shutdown** command. Example 11.4, "Shutting Down a Broker using a Remote Console Connection" shows the command sequence for using a remote console connection to shutdown a broker running on a machine named **NWBrokerHost**.

Example 11.4. Shutting Down a Broker using a Remote Console Connection

```
JBossA-MQ:karaf@root> ssh -l username -P password 8101 HOSTNAME
        _ _____ __ __ _____ | | _ \ /\ | \/ |/ __ \ | | |_) | ____ ____ ____ / \ _______| \ / | | | | _ | | _ < / _ \/ __/
__| / /\ \_______| |\/| | | | | | |__| | | |_) | (_) \__ \__ \ / _____ \ | | | | |__| | | \_____/|_____/ \____/|____/____/ /_/
\_\ |_| |_|\____\_\ AMQ (6.0.0.redhat-012)
http://www.redhat.com/products/jbossenterprisemiddleware/amq/ 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 AMQ.

JBossA-MQ:karaf@root> osgi:shutdown
Confirm: shutdown instance root (yes/no):
yes
JBossA-MQ:karaf@root> JBossA-MQ: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 11.5, "Shutting Down a Broker in a Fabric" shows how to use the administration client to shutdown a broker running in a container named **fabric-broker3**.

Example 11.5. Shutting Down a Broker in a Fabric

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

CHAPTER 12. ADDING CLIENT CONNECTION POINTS

Abstract

Message brokers must explicitly create connection points for clients. These connection points are called transport connectors. Red Hat AMQ supports a number of transport flavors to facilitate interoperability with the widest possible array of clients.

12.1. OVERVIEW OF TRANSPORT CONNECTORS

A message broker communicates with its clients using one or more ports. These ports are managed by the broker's configuration. There are two required components to add a client connection point to a broker:

- a **transportConnector** element in the XML configuration template that provides the details for the connection point
- an entry in the broker's **io.fabric8.mq.fabric.server.*id*** PID's connectors property to activate the connection point

The **transportConnector** element provides all of the details needed to create the connection point. This includes the type of transport being used, the host and port for the connection, and any transport properties needed. The connectors property is a space delimited list that specifies which transport connectors to activate.

Red Hat JBoss Fuse supports a number of different transport flavors. Each transport has its own set of strengths. For more information on the different transports see the *Client Connectivity Guide* and the *Connection Reference*.

12.2. ADDING A TRANSPORT CONNECTOR TO A STANDALONE BROKER

Adding a transport connector definition

To add a transport connector definition:

1. Open the broker's configuration template for editing.
2. Locate the **transportConnectors** element.
3. Add a **transportConnector** element as a child of the **transportConnectors** element.
4. Add a **name** attribute to the new **transportConnector** element.

   The **name** attribute specifies a unique identifier for the transport connector. It is used in the connectors property to identify the transport to be activated.
5. Add a **uri** attribute to the new **transportConnector** element.

   The **uri** attribute specifies the connection details used to instantiate the connector. Clients will use a similar URI to access the broker using this connector. For a complete list of the URIs see the *Connection Reference*.

6. Save the changes to the configuration template.

NOTE

The newly added transport connector is not available until it has been activated using the connectors property.

Activating a connector

To activate a transport connector in a standalone broker:

1. Connect to the broker using a command console.
2. Open the broker's **io.fabric8.mq.fabric.server.*id*** PID for editing using the **config:edit**command.

```
JBossAMQ:karaf> config:edit io.fabric8.mq.fabric.server.098765
```

NOTE

You can use the **config:list** command to find the *id* for the broker.

3. Verify the value of the connectors property using the **config:proplist** command.

```
JBossAMQ:karaf> config:proplist connector
```

4. Change the value of the connectors property using the **config:propset** command.

```
JBossAMQ:karaf> config:propset connector "connector1 connector2..."
```

connector1 specifies the name of a transport to activate. The value corresponds the value of the **transportConnector** element's **name** attribute.

5. Save the changes using the **config:update** command.

```
JBossAMQ:karaf> config:update
```

CHAPTER 13. ADDING A QUEUE OR A TOPIC

Abstract

Normally, you do not need to add any queues or topics explicitly, because the broker automatically creates destinations on the fly.

AUTOMATIC DESTINATION CREATION

By default, the broker automatically creates destinations on the fly. For example, when a JMS producer client tries to write a message to a non-existent queue, the broker automatically (and transparently) creates the requisite queue and puts the message on the queue. Consequently, administrators do not need to execute a command to create a new queue or a new topic on a broker.

RESTRICTING DESTINATION CREATION

In some applications, however, you might not want the broker to create destinations dynamically. In other words, you might want to restrict destination creation, so that only certain (privileged) users are allowed to create a new destination. If you need to, you can restrict destination creation by configuration of the broker's authorization plug-in. By restricting the **admin** role and not granting it to certain user groups, you can ensure that those user groups are *unable* to create new destinations on the fly.

The details of how to apply the **admin** role vary, depending on which authorization plug-in the broker uses. For full details about how to configure broker authorization, please consult the *Authorization* chapter of the *AMQ Security Guide*.

CHAPTER 14. USING LOGGING

Abstract

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.

14.1. OVERVIEW OF LOGGING

Red Hat AMQ 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

14.2. LOGGING CONFIGURATION

Abstract

To configure the logging of a broker, you need to edit the ops4j configuration and the broker's runtime 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.

NOTE

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:

```
JBossA-MQ:karaf@root> config:edit org.ops4j.pax.logging
JBossA-MQ:karaf@root> config:propappend log4j.appender.stdout.append
true
JBossA-MQ: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 Red Hat AMQ 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 14.1, “Changing Logging Levels” shows the commands for setting the root loggers log level in a standalone broker.

Example 14.1. Changing Logging Levels

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

Changing the appenders' thresholds

When debugging a problem in AMQ 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.*appenderName*.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 14.2, "Changing the Log Information Displayed on the Console" shows an example of setting the root logger to **DEBUG** but limiting the information displayed on the console to **WARN**.

Example 14.2. Changing the Log Information Displayed on the Console

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

14.3. VIEWING THE LOG

Abstract

You can view the log using your systems text display mechanisms, the Red Hat AMQ console, or the administration client.

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 AMQ 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

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.

14.4. CHANGE LOGGING LEVEL AT RUNTIME USING JCONSOLE

In standalone activemq you can change logging level through JMX at runtime. The logging level can be changed using the Log4JConfiguarion MBean which is accessible through JMX. JConsole, a part of JDK allows you the change MBean at runtime.

To change root logging level to DEBUG, follow these steps:

- Start activemq using **./bin/activemq start**.
- Open JConsole and connect to activemq. To connect to activemq use the **activemq.jar start** listed in local processes if you have launched JConsole on same machine as activemq. For connecting remotely you need to configure activemq. Refer to http://activemq.apache.org/jmx.html.
- In JConsole, click the **MBeans**tab.
- Navigate to **org.apache.activemq** -> **Broker** -> **localhost** -> **Log4JConfiguarion** -> **RootLogLevel** and set attribute value to **DEBUG**.

To change a particular logger use the ***setLogLevel*** on Log4JConfiguration MBean.

NOTE

Activemq allows accessing the attributes and operation of Log4jConfiguartion MBean through the client application, See http://docs.oracle.com/javase/tutorial/jmx/remote/custom.html.

CHAPTER 15. USING JMX

Abstract

Red Hat AMQ is fully instrumented to provide statistics about its performance using JMX. You can monitor a broker using any JMX aware monitoring tool.

15.1. INTRODUCTION TO JMX

By default Red Hat AMQ creates MBeans, loads them into the MBean server created by the JVM, and creates a dedicated JMX connector that provides a AMQ-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, AMQ'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.

15.2. CONFIGURING JMX

Abstract

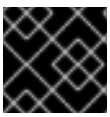
By default, brokers have JMX activated. However, a broker's JMX behavior is highly configurable. You can specify if JMX is used, if the broker uses a dedicated JMX connector, if the broker creates its own MBean server, and the JMX URL it uses.

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/karaf-*containerName*. 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 Red Hat AMQ 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 15.1, "Broker JMX Configuration Properties" describes the configuration properties for controlling a broker's JMX behavior.

Table 15.1. Broker JMX Configuration Properties

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

Property	Default Value	Description
connectorPath	**/jmxrmi**	Specifies the path under which the JMX connector will be registered.
suppressMBean	*empty*	Specifies a comma-separated list of MBean name patterns to ignore. For example: **endpoint=dynamicProducer, endpoint=Consumer,connectionName=*,destinationName=ActiveMQ.Advisory.***

[a] The default configuration template for the broker sets this property to **false** so that the broker uses the container's JMX connection.

Example 15.1, "Configuring a Broker's JMX Connection" shows configuration for a broker that will only use the JVM's MBean server and will not create its own JMX connector.

Example 15.1. Configuring a Broker's JMX Connection

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

15.3. STATISTICS COLLECTED BY JMX

Broker statistics

Table 15.2, "Broker JMX Statistics" describes the statistics collected for a broker.

Table 15.2. 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.

Name	Description
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 15.3, “Destination JMX Statistics” describes the statistics collected for a destination.

Table 15.3. 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.

Name	Description
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.
MaxEnqueueTime	Specifies the longest amount of time, in milliseconds, that a message sat in the destination before being consumed.

Name	Description
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 15.4, "Connection JMX Statistics" describes the statistics collected for a subscription.

Table 15.4. 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.

15.4. MANAGING THE BROKER WITH JMX

Abstract

All of the exposed MBeans have operations that allow you to perform management tasks on the broker. These operations allow you to stop a broker, start and stop network connectors, create and destroy destinations, and create and destroy subscriptions. The MBeans also provide operations for browsing destinations and passing test messages to destinations.

Overview

The MBeans exposed by Red Hat AMQ 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 15.5, "Broker MBean Operations" describes the operations exposed by the MBean for a broker.

Table 15.5. Broker MBean Operations

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

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

Connector actions

Table 15.6, "Connector MBean Operations" describes the operations exposed by the MBean for a transport connector.

Table 15.6. Connector MBean Operations

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

Operation	Description
void disableStatistics();	Deactivates the collection of statistics for the connector.

Network connector actions

Table 15.7, "Network Connector MBean Operations" describes the operations exposed by the MBean for a network connector.

Table 15.7. Network Connector MBean Operations

Operation	Description
void start();	Starts the network connector so that it is ready to communicate with other brokers in a network of brokers.
void stop();	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 15.8, "Queue MBean Operations" describes the operations exposed by the MBean for a queue destination.

Table 15.8. Queue MBean Operations

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

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

Operation	Description
java.util.List browseMessages(String selector);	Returns all messages in the queue that match the selector, without changing the cursor, as a list.
java.util.List browseMessages();	Returns all messages in the queue, without changing the cursor, as a list.
String sendTextMessage(String body, String username, String password);	Send a text message to a secure queue.
String sendTextMessage(String body);	Send a text message to a queue.

Topic actions

Table 15.9, "Topic MBean Operations" describes the operations exposed by the MBean for a topic destination.

Table 15.9. Topic MBean Operations

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

Subscription actions

Table 15.10, "Subscription MBean Operations" describes the operations exposed by the MBean for a durable subscription.

Table 15.10. Subscription MBean Operations

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

CHAPTER 16. APPLYING PATCHES

Abstract

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

16.1. INTRODUCTION TO PATCHING

Patching enables you apply fixes to a broker without needing to reinstall an updated version of Red Hat AMQ. It also allows you to back out the patch, 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 artifacts are typically one or more bundles. They can, however, include configuration files and feature descriptors.

In addition, since AMQ 6.2.1, a full distribution can be used as a rollup patch. This enables you to upgrade all aspects of an existing AMQ distribution—including installed features, bundles and configurations—while preserving custom configuration changes.

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.
- Download a patch directly from the Red Hat customer portal.

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.

16.2. FINDING THE RIGHT PATCHES TO APPLY

Abstract

This section explains how to find the patches for a specific version of AMQ on the Red Hat Customer Portal and how to figure out which patches to apply, and in what order.

Locate the patches on the customer portal

If you have a subscription for AMQ, you can download the latest patches directly from the Red Hat Customer Portal. Locate the patches as follows:

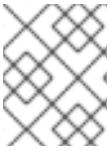
1. Login to the Red Hat Customer Portal using your customer account. This account *must* be associated with an appropriate Red Hat software subscription, otherwise you will not be able to see the patch downloads for AMQ.
2. Navigate to the customer portal Software Downloads page.

3. In the **Product** dropdown menu, select the appropriate product (for example, **A-MQ** or **Fuse**), and then select the version, 6.3, from the **Version** dropdown menu. A table of downloads now appears, which has three tabs: **Releases**, **Patches**, and **Security Advisories**.

4. Click the **Releases** tab to view the GA product releases.

5. Click the **Patches** tab the rollup patches, and the regular incremental patches (with no security-related fixes).

6. Click the **Security Advisories** tab to view the incremental patches with security-related fixes.



NOTE

To see the *complete* set of patches, you must look under the **Releases** tab, the **Patches** tab *and* the **Security Advisories** tab.

Types of patch

The following types of patch can be made available for download:

- Rollup patches
- Incremental patches

Rollup patches

A rollup patch is a cumulative patch that incorporates *all* of the fixes from the preceding patches. Moreover, each rollup patch is regression tested and establishes a new baseline for the application of future patches.

Since AMQ 6.2.1, a rollup patch file is dual-purpose, as follows:

- Each rollup patch file is a complete new build of the official target distribution. This means you can unzip the rollup patch file to obtain a completely new installation of AMQ, just as if it was a fresh download of the product (which, in fact, it is). See Section 16.3, "Installing a Rollup Patch as a New Installation".
- You can also treat the rollup patch as a regular patch, using it to upgrade an existing installation. That is, you can provide the rollup patch file as an argument to the standalone patch console commands (for example, **patch:add** and **patch:install**) or the Fabric patch console command, **patch:fabric-install**.

Incremental patches

Incremental patches are patches released either directly after GA or after a rollup patch, and they are intended to be applied on top of the corresponding build of AMQ. The main purpose of an incremental patch is to update some of the bundles in an existing distribution.

Which patches are needed to update the GA product to the latest patch level?

To figure out which patches are needed to update the GA product to the latest patch level, you need to pay attention to the type of patches that have been released so far:

1. If the only patches released so far are patches with GA baseline (Patch 1, Patch 2, and so on), apply the *latest* of these patches directly to the GA product.

2. If a rollup patch has been released and no patches have been released after the latest rollup patch, simply apply the latest rollup patch to the GA product.
3. If the latest patch is a patch with a rollup baseline, you must apply two patches to the GA product, as follows:
    a. Apply the latest rollup patch, and then
    b. Apply the latest patch with a rollup baseline.

Which patches to apply, if you only want to install regression-tested patches?

If you prefer to install only patches that have been regression tested, install the latest rollup patch.

16.3. INSTALLING A ROLLUP PATCH AS A NEW INSTALLATION

A rollup patch is a new build

Since AMQ 6.2.1, a rollup patch file is a complete new build of the official target distribution. In other words, it is just like the original GA distribution, except that it includes later build artifacts.

Installing the new build

To install a new build, corresponding to a rollup patch level, perform the following steps:

1. Identify which rollup patch you need to install and download it from the Customer Portal. For more details, see Section 16.2, “Finding the Right Patches to Apply” .
2. Unzip the rollup patch file to a convenient location, just as you would with a regular GA distribution. This is your new installation of AMQ.

Comparison with patch process

Compared with the conventional patch process, installing a new build has the following advantages and limitations:

- This approach is only for creating a completely new installation of AMQ. If your existing installation already has a lot of custom configuration, this might not be the most convenient approach to use.
- The new build includes only the artifacts and configuration for the new patch level. There is thus no concept of *rolling back* to the GA version.
- If you create a new fabric (using **fabric:create**), the default fabric profiles are already at the new patch level (same as the standalone container). It is therefore not necessary to patch the fabric.

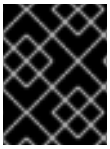
16.4. PATCHING A STANDALONE CONTAINER

Abstract

You apply patches to a standalone container using the command console's **patch** shell. You can apply and roll back patches as needed.

Overview

When patching a standalone container, you can apply either an incremental patch or a rollup patch. There are very significant differences between the two kinds of patch and the way they are applied. Although the same commands are used in both cases, the internal processes are different (the patch commands auto-detect the patch type).



IMPORTANT

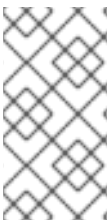
The instructions in this section apply only to AMQ versions 6.2.1 and later, which support the new patching mechanism.

Incremental patch

An incremental patch is used mainly to update the *bundle JAR*s in the container. This type of patch is suitable for delivering hot fixes to the AMQ installation, but it has its limitations. An incremental patch:

- Updates bundle JARs.
- Patches only the current container instance (under the **data**/ directory). Hence, patches are *not* preserved after deleting a container instance.
- Updates any feature dependencies installed in the current container instance, but does not update the feature files themselves.
- Might update some configuration files, but is *not* suitable for updating most configuration files.
- Supports patch rollback.
- After applying the patch, and creating a new fabric using **fabric:create**, the new Fabric reverts to the unpatched configuration.

After applying an incremental patch to a standalone container, meta-data about the patch is written to the **etc**/**startup.properties** and **etc**/**overrides.properties** files. With these files, the Karaf installation is able to persist the patch even after deleting the root container instance (that is, after removing the root container's **data**/ directory).



NOTE

Removing the **data**/**cache** directory uninstalls any bundles, features, or feature repositories that were installed into the container using Karaf console commands. However, any patches that have been applied will remain installed, as long as the **etc**/**startup.properties** and **etc**/**overrides.properties** files are preserved.

Rollup patch

A rollup patch can make updates to *any* installation files including *bundle JARs* and *static files* (including, for example, configuration files under the **etc**/ directory). A rollup patch:

- Updates any files, including bundle JARs, configuration files, and any static files.
- Patches both the current container instance (under the **data**/ directory) and the underlying installation. Hence, patches are preserved after deleting a container instance.

- Updates all of the files related to Karaf features, including the features repository files and the features themselves. Hence, any features installed after the rollup patch will reference the correct patched dependencies.
- If necessary, updates configuration files (for example, files under **etc**/), automatically merging any configuration changes you have made with the configuration changes made by the patch. If merge conflicts occur, see the patch log for details of how they are handled.
- Tracks *all* of the changes made to the installation (including to static files), so that it is possible to roll back the patch.

NOTE

The rollup patching mechanism uses an internal git repository (located under **patches/.management/history**) to track the changes made.

- After applying the patch, and creating a new fabric using **fabric:create**, the new Fabric is created with the patched configuration.

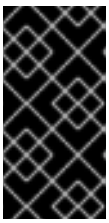
Patching the patch mechanism

(Recommended, if applicable) If there is no patch management package corresponding to the rollup patch you are about to install, then you can skip this procedure and install the rollup patch directly.

From time to time, important changes and improvements are made to the patch mechanism. In order to pick up these improvements, we recommend that you patch the patch mechanism to a higher level *before* upgrading AMQ with a rollup patch. If you were to upgrade straight to the latest rollup patch version of AMQ, the improved patch mechanism would become available *after* you completed the upgrade. But at that stage, it would be too late to benefit from the improvements in the patch mechanism.

To circumvent this bootstrap problem, the improved patch mechanism is made available as a separate download, so that you can patch the patch mechanism itself, *before* you upgrade to the new patch level. To patch the patch mechanism, proceed as follows:

1. Download the appropriate patch management package. From the AMQ 6.3.0 Software Downloads page, select a package named **Red Hat AMQ 6.3.0 Rollup *N* on Karaf Update Installer**, where *N* is the number of the particular rollup patch you are about to install.



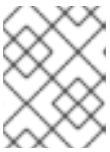
IMPORTANT

The rollup number, *N*, of the downloaded patch management package *must* match the rollup number of the rollup patch you are about to install. For some rollup patches, there is no corresponding patch management package, in which case you can skip directly to the instructions for installing the rollup patch.

2. Install the patch management package, **patch-management-for-amq-630-*TargetVersion*.zip**, on top of your 6.3.0 installation. Use an archive utility to extract the contents on top of the existing Karaf container installation (installing files under the **system**/ and **patches**/ subdirectories).

IMPORTANT

Ensure the credentials you are using to unzip the file are the same credentials used to install Fuse; otherwise, an error may occur since Fuse will not have access to the patched file.



NOTE

It does not matter whether the container is running or not when you extract these files.

3. Start the container, if it is not already running.

4. Uninstall the existing patch commands from the container as follows. Remove the patch features as follows:

```
JBossFuse:karaf@root> features:uninstall patch patch-core
```

But this is not sufficient to remove all of the patch bundles. Check for any remaining patch bundles as follows:

```
JBossFuse:karaf@root> list -t 0 -l | grep patch

[   1] [Active     ] [            ] [       ] [    2] mvn:io.fabric8.patch/patch-management/1.2.0.redhat-
630187
```

You can remove this system bundle, as follows:

```
JBossFuse:karaf@root> uninstall 1
You are about to access system bundle 1.  Do you wish to continue (yes/no): yes
JBossFuse:karaf@root> list -t 0 -l | grep patch
```

Finally, you should remove the features URL for the old patch mechanism version, as follows:

```
JBossFuse:karaf@root> features:listurl | grep patch
  true    mvn:io.fabric8.patch/patch-features/1.2.0.redhat-630187/xml/features
JBossFuse:karaf@root> features:removeurl mvn:io.fabric8.patch/patch-features/1.2.0.redhat-
630187/xml/features
```

Check the version of **patch-features** that you have, because it might be different from **1.2.0.redhat-630187**.

5. Install the new patch commands as follows. Add the features URL for the new patch commands, as follows:

```
JBossFuse:karaf@root> features:addurl mvn:io.fabric8.patch/patch-features/1.2.0.redhat-
630xxx/xml/features
```

Where you must replace **1.2.0.redhat-630xxx** with the actual build version of the patch commands you are installing (for example, the build version **xxx** can be taken from the last three digits of the ***TargetVersion*** in the downloaded patch management package file name).

Install the new patch features, as follows:

```
JBossFuse:karaf@root> features:install patch-core patch
```

Check that the requisite patch bundles are now installed:

```
JBossFuse:karaf@root> list -t 0 -l | grep patch
[ 265] [Active     ] [            ] [       ] [   80] mvn:io.fabric8.patch/patch-core/1.2.0.redhat-630xxx
[ 266] [Active     ] [            ] [       ] [    2] mvn:io.fabric8.patch/patch-management/1.2.0.redhat-630xxx
[ 267] [Active     ] [            ] [       ] [   80] mvn:io.fabric8.patch/patch-commands/1.2.0.redhat-630xxx
```

6. Restart the container (the **patch:add** command and the other patch commands will not be available in the console shell until you perform a restart).

Applying a patch

To apply a patch to a standalone container:

1. Make a full backup of your AMQ installation before attempting to apply the patch.
2. *(Rollup patch only)* Before applying the rollup patch to your container, you *must* patch the patch mechanism, as described in the section called “Patching the patch mechanism” .
3. *(Rollup patch only)* Remove the **lib/endorsed/org.apache.karaf.exception-2.4.0.redhat-630xxx.jar** file (where the build number, **xxx**, depends on the build being patched).
4. *(Incremental patch only)* Before you proceed to install the patch, make sure to read the text of the **README** file that comes with the patch, as there might be additional *manual steps* required to install a particular patch.
5. Start the container, if it is not already running. If the container is running in the background (or remotely), connect to the container using the SSH console client, **/bin/client**.
6. Add the patch to the container's environment by invoking the **patch:add** command. For example, to add the ***patch*.zip** patch file:

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

7. Simulate installing the patch by invoking the **patch:simulate** command.

   This generates a log of the changes that will be made to the container when the patch is installed, but will not make any actual changes to the container. Review the simulation log to understand the changes that will be made to the container.

8. Invoke the **patch:list** command to display a list of added patches. In this list, the entries under the **[name]** heading are patch IDs. For example:

```
 patch:list
[name]                      [installed]   [description]
jboss-a-mq-6.3.0.redhat-329  false
```

   Ensure that the container has fully started before you try to perform the next step. In some cases, the container must restart before you can apply a patch, for example, if static files are patched. In these cases, the container restarts automatically.

9. Apply a patch to the container by invoking the **patch:install** command and specifying the patch ID for the patch that you want to apply. For example:

```
patch:install jboss-a-mq-6.3.0.redhat-329
```

10. Validate the patch, by searching for one of the patched artifacts. For example, if you had just upgraded AMQ 6.2.1 to the patch with build number **621423**, you could search for bundles with this build number, as follows:

```
AMQ:karaf@root> osgi:list -s -t 0 | grep -i 630187
[   6] [Active     ] [            ] [       ] [   10] org.apache.felix.configadmin    (1.2.0.redhat-630187)
```

After applying a rollup patch, you also see the new version and build number in the Welcome banner when you restart the container.

Rolling back a patch

Occasionally a patch will not work or will introduce new issues to a container. In these cases, you can easily back the patch out of the system and restore it to pre-patch behaviour using the **patch:rollback** command, as follows:

1. Invoke the **patch:list** command to obtain the patch ID, ***PatchID***, of the most recently installed patch.

2. Invoke the **patch:rollback** command, as follows:

```
patch:rollback PatchID
```

In some cases the container will need to restart to roll back the patch. In these cases, the container restarts automatically. Due to the highly dynamic nature of the OSGi runtime, during the restart you might see some occasional errors related to incompatible classes. These are related to OSGi services that have just started or stopped. These errors can be safely ignored.

Adding features to an incrementally patched container

Since AMQ 6.1, it is possible to add Karaf features to an already patched standalone container without performing any special steps.

16.5. PATCHING STANDALONE APACHE ACTIVEMQ

Abstract

AMQ provides a standalone distribution of Apache ActiveMQ (that is, Apache ActiveMQ without the Apache Karaf container) under the ***InstallDir*/extras** directory. Patching the standalone Apache ActiveMQ is a manual process, requiring you to copy some library files.

Patch files

The first step in patching a standalone Apache ActiveMQ instance is to figure out what patches need to be applied. When it comes to determining which patches to apply, the same principles apply as for patching the container.

See Section 16.2, "Finding the Right Patches to Apply" for details of how to work out which patches to apply and download the relevant patches.

Apache ActiveMQ install directory

For the following patching instructions, it is assumed that you have already extracted the standalone Apache ActiveMQ distribution from the **extras/apache-activemq-5.11.0.redhat-630187.zip** file and installed standalone Apache ActiveMQ into the ***ApacheActiveMQInstall*** directory.

How to apply a patch to standalone Apache ActiveMQ

To apply a patch (or patches) to a standalone Apache ActiveMQ instance, perform the following steps:

1. After determining which patches to apply, download the relevant patches from the Customer Portal, as described in Section 16.2, "Finding the Right Patches to Apply" .
2. Stop the ActiveMQ broker, if it is running.
3. Make a backup copy of the original standalone Apache ActiveMQ **lib** directory, ***ApacheActiveMQInstall*/lib**
4. Starting with the first patch file, use an archive utility to open the downloaded patch (**.zip**) file, and extract the patch to a convenient temporary location, ***ExtractedPatch***.
5. The patched library files for the standalone Apache ActiveMQ instance are located in the following subdirectory of the patch:

   ```
   ExtractedPatch/apache-activemq-5.11.0.redhat-630187/lib
   ```

   Copy the complete contents of this directory to the standalone Apache ActiveMQ **lib** directory, ***ApacheActiveMQInstall*/lib**.
6. Delete the older versions of the patched library files in ***ApacheActiveMQInstall*/lib**. Only *one* version of each library should be present in the lib directory, and it should be the patched version.

   For example, if you found two versions of the **activemq-broker** JAR file present in the **lib** directory after copying the patch libraries:

   ```
   activemq-broker-5.9.0.redhat-610379.jar
   activemq-broker-5.9.0.redhat-611423.jar
   ```

   You would delete the older version, **activemq-broker-5.9.0.redhat-610379.jar**.
7. If you need to install a second patch on top of the first, repeat steps 4, 5, and 6, for the second patch.
8. Restart the ActiveMQ broker.

16.6. PATCHING A FABRIC CONTAINER WITH A ROLLUP PATCH

Abstract

Follow the procedures described in this section to patch a Fabric container with a *rollup patch*.

Overview

A rollup patch updates *bundle JARs*, other *Maven artifacts*, *libraries*, and *static files* in a Fabric. The following aspects of the fabric are affected:

- Distribution of patched artifacts
- Profiles
- Configuration of the underlying container

Root container

Throughout this section, we refer to a *root container*, which is just a container chosen from the Fabric ensemble. Throughout the patching procedure, you invoke the **patch:*** commands from the console of the root container. If you are planning to distribute patch artifacts through the Maven proxy, it is convenient to choose the root container to be the ensemble container that is currently the master of the Maven proxy cluster (see ???). This would ensure that patch artifacts can immediately be downloaded by other containers in the cluster.

Distribution of patch artifacts

When patching an entire fabric of containers, you need to consider how the patch artifacts are distributed to the containers in the fabric. You can adopt one of the following approaches:

- *Through the Maven proxy* (default approach)—when you add a rollup patch to your root container (using the **patch:add** command), the patch artifacts are installed into the root container's **system**/ directory, whose directory structure is laid out like a Maven repository. The root container can then serve up these patch artifacts to remote containers by behaving as a Maven proxy, enabling remote containers to download the required Maven artifacts (this process is managed by the Fabric agent running on each Fabric container). Alternatively, if you have installed the rollup patch to a container that is *not* hosting the Maven proxy, you can ensure that the patch artifacts are uploaded to the Maven proxy by invoking the **patch:fabric-install** command with the **--upload** option.

  There is a limitation to the Maven proxy approach, however, if the Fabric ensemble consists of multiple containers. In this case, it can happen that the Maven proxy fails over to a different ensemble container (not the original root container). This can result in the patch artifacts suddenly becoming unavailable to other containers in the fabric. If this occurs during the patching procedure, it will cause problems.

  NOTE

  Containers that are added to an ensemble do *not* automatically deploy the Maven proxy. To enable the Maven proxy, make sure that the **fabric** profile is deployed in the container.

  For more details, see chapter "Fabric Maven Proxies" in "Fabric Guide" .

- *Through a local repository* (recommended approach)—to overcome the limitations of the Maven proxy approach, we recommend that you make the patch artifacts available directly to all of the containers in the Fabric by setting up a *local repository* on the file system. Assuming that you have a networked file system, all containers will be able to access the patch artifacts directly.

  For example, you might set up a local repository of patch artifacts, as follows:

1. Given a rollup patch file, extract the contents of the **system**/ directory from the rollup patch file into the **repositories**/ subdirectory of a local Maven repository (which could be ~/**.m2**/**repositories** or any other location).

2. Configure the Fabric agent and the Maven proxy to pick up artifacts from the local repository by editing the current version of the **default** profile, as follows:

```
profile-edit --append --pid
io.fabric8.agent/org.ops4j.pax.url.mvn.defaultRepositories="file:///PathToRepository"
default
```

Replace ***PathToRepository*** by the actual location of the local repository on your file system.

NOTE

Make sure that you make the edits to the **default** profile for all relevant profile versions. If some of your containers are using a non-default profile version, repeat the **profile-edit** commands while specifying the profile version explicitly as the last parameter.

Profiles

The rollup patching process updates all of the standard profiles, so that they reference the patched dependencies. Any custom profiles that you created yourself remain unaffected by these updates. However, in cases where you have already made some changes directly to the *standard profiles* (such as **default**, **fabric**, **karaf**, and so on), the patching mechanism attempts to merge your changes with the changes introduced by the patch.

IMPORTANT

In the case where you have modified standard profiles, it is recommended that you verify your custom changes are preserved after patching. This is particularly important with respect to any changes made to the location of Maven repositories (which are usually configured in the **default** profile).

Configuration of the underlying container

If required, the rollup patching mechanism is capable of patching the underlying container (that is, files located under **etc**/, **lib**/, and so on). When a Fabric container is upgraded to a patched version (for example, using the **fabric:container-upgrade** command), the container's Fabric agent checks whether the underlying container must be patched. If yes, the Fabric agent triggers the patching mechanism to update the underlying container. Moreover, if certain critical files are updated (for example, **lib**/**karaf.jar**), the container status changes to **requires full restart** after the container is upgraded. This status indicates that a full *manual* restart is required (an automatic restart is not possible in this case).

io.fabric.version in the default profile

The **io.fabric.version** resource in the **default** profile plays a key role in the patching mechanism. This resource defines the version and build of AMQ and of all of its main components. When upgrading (or rolling back) a Fabric container to a new version, the Fabric agent checks the version and build of AMQ as defined in the **io.fabric.version** resource. If the AMQ version changes between the original profile version and the upgraded profile version, the Fabric agent knows that an upgrade of the underlying container is required when upgrading to this profile version.

Patching the patch mechanism

(Recommended, if applicable) If there is no patch management package corresponding to the rollup patch you are about to install, then you can skip this procedure and install the rollup patch directly.

From time to time, important changes and improvements are made to the patch mechanism. In order to pick up these improvements, we recommend that you patch the patch mechanism to a higher level *before* upgrading AMQ with a rollup patch. If you were to upgrade straight to the latest rollup patch version of AMQ, the improved patch mechanism would become available *after* you completed the upgrade. But at that stage, it would be too late to benefit from the improvements in the patch mechanism.

To circumvent this bootstrap problem, the improved patch mechanism is made available as a separate download, so that you can patch the patch mechanism itself, before you upgrade to the new patch level. To patch the patch mechanism, proceed as follows:

1. Download the appropriate patch management package. From the AMQ 6.3.0 Software Downloads page, select a package named **Red Hat AMQ 6.3.0 Rollup *N* on Karaf Update Installer**, where *N* is the number of the particular rollup patch you are about to install.

IMPORTANT

The rollup number, *N*, of the downloaded patch management package *must* match the rollup number of the rollup patch you are about to install. For some rollup patches, there is no corresponding patch management package, in which case you can skip directly to the instructions for installing the rollup patch.

2. Extract the contents of the patch management package, **patch-management-for-amq-630-*TargetVersion*.zip**, on top of the root container (that is, on top of the Fabric container that will be used to perform the remainder of the patching tasks). Use an archive utility to extract the contents on top of the root container installation, merging the contents of the archive **system/** and **patches/** directories with the container **system/** and **patches/** subdirectories.

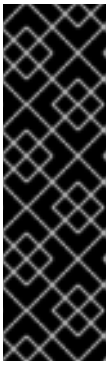
NOTE

It does not matter whether the root container is running when you extract these files.

3. Start the root container, if it is not already running.

4. Create a new version, using the **fabric:version-create** command (where we assume that the current profile version is **1.0**):

```
JBossFuse:karaf@root> fabric:version-create --parent 1.0 1.0.1
Created version: 1.0.1 as copy of: 1.0
```



IMPORTANT

Version names are important! The tooling sorts version names based on the numeric version string, according to *major.minor* numbering, to determine the version on which to base a new one. You can safely add a text description to a version name as long as you append it to the end of the generated default name like this: **1.3 [.description]**. If you abandon the default naming convention and use a textual name instead (for example, Patch051312), the next version you create will be based, not on the last version (Patch051312), but on the highest-numbered version determined by dot syntax.

5. Update the **patch** property in the **io.fabric8.version** PID in the version **1.0.1** of the **default** profile, by entering the following Karaf console command:

```
profile-edit --pid io.fabric8.version/patch=1.2.0.redhat-630xxx default 1.0.1
```

   Where you must replace **1.2.0.redhat-630xxx** with the actual build version of the patch commands you are installing (for example, the build version **xxx** can be taken from the last three digits of the ***TargetVersion*** in the downloaded patch management package file name).

6. Upgrade the root container to use the new patching mechanism, as follows:

```
container-upgrade 1.0.1 root
```

7. Likewise, for *all* other containers in your fabric that need to be patched (SSH, child, and so on), provision them with the new patching mechanism by upgrading them to profile version **1.0.1**. For example:

```
container-upgrade 1.0.1 container1 container2 container3
```

8. After completing the container-upgrade, if **patch** commands are unavailable or if the console issues a prompt that a container restart is necessary, then restart the upgraded containers to complete the upgrade process.

Applying a rollup patch

To apply a rollup patch to a Fabric container:

1. Before applying the rollup patch to your fabric, you *must* patch the patch mechanism, as described in the section called “Patching the patch mechanism” .

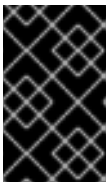
2. For every top-level container (that is, any container that is not a child container), perform these steps, one container at a time:

   1. In the corresponding Karaf installation, remove the **lib/endorsed/org.apache.karaf.exception-2.4.0.redhat-630xxx.jar** file (where the build number, **xxx**, depends on the build being patched).

   2. Restart the container.

3. Add the patch to the root container's environment using the **patch:add** command. For example, to add the ***patch*.zip** patch file:

```
JBossFuse:karaf@root> patch:add file://patch.zip
[name]          [installed] [description]
PatchID         false       Description
```

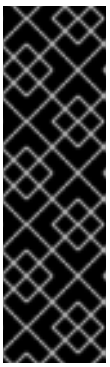


IMPORTANT

If you have decided to use a local repository to distribute the patch artifacts *(recommended)*, set up the local repository now—see the section called "Distribution of patch artifacts".

4. Create a new version, using the **fabric:version-create** command:

```
JBossFuse:karaf@root> fabric:version-create 1.1
Created version: 1.1 as copy of: 1.0.1
```

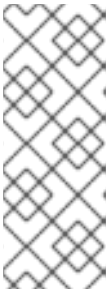


IMPORTANT

Version names are important! The tooling sorts version names based on the numeric version string, according to *major.minor* numbering, to determine the version on which to base a new one. You can safely add a text description to a version name as long as you append it to the end of the generated default name like this: **1.3[.description]**. If you abandon the default naming convention and use a textual name instead (for example, Patch051312), the next version you create will be based, not on the last version (Patch051312), but on the highest-numbered version determined by dot syntax.

5. Apply the patch to the new version, using the **patch:fabric-install** command. Note that in order to run this command you *must* provide the credentials, ***Username*** and ***Password***, of a user with **Administrator** privileges. For example, to apply the ***PatchID*** patch to version **1.1**:

```
patch:fabric-install --username Username --password Password --upload --version 1.1
PatchID
```



NOTE

When you invoke the **patch:fabric-install** command with the **--upload** option, Fabric looks up the ZooKeeper registry to discover the URL of the currently active Maven proxy, and uploads all of the patch artifacts to this URL. Using this approach it is possible to make the patch artifacts available through the Maven proxy, even if the container you are currently logged into is not hosting the Maven proxy.

6. Delete the old bundle overrides created by the old hot fix patch by modifying the parent profiles of the profile default and removing the old hot fix patch profile as being a parent of the default profile. For example,

```
JBossFuse:karaf@root> fabric:profile-display --version 1.X default
Attributes:
parents: acls patch-jboss-fuse-6.2.1.redhat-186-12-r7hf10
JBossFuse:karaf@root> fabric:profile-change-parents --version 1.X default acls
```

NOTE

The parent **patch-jboss-fuse-6.2.1.redhat-186-12-r7hf10** is only visible if a hot fix patch was installed previously. The name of the parent patch is different based on the hot fix patch.

The above commands shows that default profile has two parents:

- acls - standard and must be present.
- patch-jboss-fuse-6.2.1.redhat-186-12-r7hf10 - a profile that represents hotfix patch.

7. Synchronize the patch information across the fabric, to ensure that the profile changes in version **1.1** are propagated to all containers in the fabric (particularly remote SSH containers). Enter the following console command:

```
patch:fabric-synchronize
```

8. Upgrade each existing container in the fabric using the **fabric:container-upgrade** command (but leaving the root container, where you installed the patch, until last). For example, to upgrade a container named **remote**, enter the following command:

```
JBossFuse:karaf@root> fabric:container-upgrade 1.1 remote
Upgraded container remote from version 1.0.1 to 1.1
```

At this point, not only does the Fabric agent download and install the patched bundles into the specified container, but *the agent also applies the patch to the underlying container* (updating any static files in the container, if necessary). If necessary, the agent will then restart the target container automatically or set the container status to **requires full restart** (if an automatic restart is not possible), so that any changes made to the static files are applied to the running container.

IMPORTANT

It is recommended that you upgrade only one or two containers to the patched profile version, to ensure that the patch does not introduce any new issues.

9. If the current status of the upgraded container is **requires full restart**, you must now use one of the standard mechanisms to stop and restart the container manually. In some cases, it will be possible to do this using Fabric commands from the console of the root container.

For example, you could stop the **remote** container as follows:

```
fabric:container-stop remote
```

And restart the **remote** container as follows:

```
fabric:container-start remote
```

10. Upgrade the root container last (that is, the container that you originally installed the patch on):

```
fabric:container-upgrade 1.1 root
```

11. *(Windows only)* If the root container status has changed to **requires full restart** and it is running on a Windows operating system, you must first shut down all of the root container's child containers (if any) before manually restarting the root container.

    For example, if the root container has three child containers, **child1**, **child2**, and **child3**, you would first shut them down, as follows:

    ```
    fabric:container-stop child1 child2 child3
    ```

    You can then shut down the root container with the **shutdown** command:

    ```
    shutdown
    ```

Rolling back a rollup patch

To roll back a rollup patch on a Fabric container, use the **fabric:container-rollback** command. For example, assuming that **1.0** is an unpatched profile version, you can roll the **remote** container back to the unpatched version **1.0** as follows:

```
fabric:container-rollback 1.0 remote
```

At this point, not only does the Fabric agent roll back the installed profiles to an earlier version, but *the agent also rolls back the patch on the underlying container* (restoring any static files to the state they were in before the patch was applied, if necessary). If necessary, the agent will then restart the target container automatically or set the container status to **requires full restart** (if an automatic restart is not possible), so that any changes made to the static files are applied to the running container.

16.7. PATCHING A FABRIC CONTAINER WITH AN INCREMENTAL PATCH

Abstract

Follow the procedures described in this section to patch a Fabric container with an *incremental patch*.

Overview

An incremental patch makes updates only to the *bundle JARs* in a Fabric. The following aspects of the fabric are affected:

- Distribution of patched artifacts through Maven proxy
- Profiles

Distribution of patched artifacts through Maven proxy

When you install the incremental patch on your local container, the patch artifacts are installed into the local **system**/ directory, whose directory structure is laid out like a Maven repository. The local container distributes these patch artifacts to remote containers by behaving as a Maven proxy, enabling remote containers to upload bundle JARs as needed (this process is managed by the Fabric agent running on each Fabric container). For more details, see chapter "Fabric Maven Proxies" in "Fabric Guide" .

Profiles

The incremental patching process defines bundle overrides, so that profiles switch to use the patched dependencies (bundle JARs). This mechanism works as follows:

1. The patch mechanism creates a new profile, **patch-*PatchProfileID***, which defines bundle overrides for all of the patched bundles.
2. The new patch profile, **patch-*PatchProfileID***, is inserted as the parent of the **default** profile (at the base of the entire profile tree).
3. All of the profiles that inherit from default now use the bundle versions defined by the overrides in **patch-*PatchProfileID***. The contents of the existing profiles themselves *are not modified* in any way.

Is it necessary to patch the underlying container?

Usually, when patching a fabric with an incremental patch, it is *not* necessary to patch the underlying container as well. Fabric has its own mechanisms for distributing patch artifacts (for example, using a git repository for the profile data, and Apache Maven for the OSGi bundles), which are independent of the underlying container installation.

In exceptional cases, however, it might be necessary to patch the underlying container (for example, if there was an issue with the **fabric:create** command). Always read the patch **README** file to find out whether there are any special steps required to install a particular patch. In these cases, however, it is more likely that the patch would be distributed in the form of a rollup patch, which has the capability to patch the underlying container automatically—see Section 16.6, "Patching a Fabric Container with a Rollup Patch".

Applying an incremental patch

To apply an incremental patch to a Fabric container:

1. Before you proceed to install the incremental patch, make sure to read the text of the **README** file that comes with the patch, as there might be additional *manual steps* required to install a particular incremental patch.
2. Create a new version, using the **fabric:version-create** command:

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

IMPORTANT

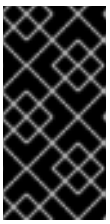
Version names are important! The tooling sorts version names based on the numeric version string, according to *major.minor* numbering, to determine the version on which to base a new one. You can safely add a text description to a version name as long as you append it to the end of the generated default name like this: **1.3 <.description >** .If you abandon the default naming convention and use a textual name instead (for example, Patch051312), the next version you create will be based, not on the last version (Patch051312), but on the highest-numbered version determined by dot syntax.

3. Apply the patch to the new version, using the **fabric:patch-apply** command. For example, to apply the **activemq.zip** patch file to version **1.1**:

```
JBossFuse:karaf@root> fabric:patch-apply --version 1.1 file:///patches/activemq.zip
```

4. Upgrade a container using the **fabric:container-upgrade** command, specifying which container you want to upgrade. For example, to upgrade the **child1** container, enter the following command:

```
JBossFuse:karaf@root> fabric:container-upgrade 1.1 child1
Upgraded container child1 from version 1.0 to 1.1
```



IMPORTANT

It is recommended that you upgrade only one or two containers to the patched profile version, to ensure that the patch does not introduce any new issues. Upgrade the **root** container (the one that you applied the patch to, using the **fabric:patch-apply** command) last.

5. You can check that the new patch profile has been created using the **fabric:profile-list** command, as follows:

```
BossFuse:karaf@root> fabric:profile-list --version 1.1 | grep patch
default                           0             patch-activemq-patch
patch-activemq-patch
```

Where we presume that the patch was applied to profile version 1.1.

NOTE

If you want to avoid specifying the profile version (with **--version**) every time you invoke a profile command, you can change the default profile version using the **fabric:version-set-default** ***Version*** command.

You can also check whether specific JARs are included in the patch, for example:

```
JBossFuse:karaf@root> list | grep -i activemq
[ 131] [Active     ] [Created     ] [       ] [   50] activemq-osgi (5.9.0.redhat-61037X)
[ 139] [Active     ] [Created     ] [       ] [   50] activemq-karaf (5.9.0.redhat-61037X)
[ 207] [Active     ] [            ] [       ] [   60] activemq-camel (5.9.0.redhat-61037X)
```

Rolling back an incremental patch

To roll back an incremental patch on a Fabric container, use the **fabric:container-rollback** command. For example, assuming that **1.0** is an unpatched profile version, you can roll the **child1** container back to the unpatched version **1.0** as follows:

```
fabric:container-rollback 1.0 child1
```

APPENDIX A. REQUIRED JARS

OVERVIEW

To simplify deploying Red Hat AMQ 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 Red Hat AMQ 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

The following JARs are installed with AMQ and must be placed on the broker's **CLASSPATH**:

- **activemq-broker.jar**
- **activemq-client.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**

PERSISTENT MESSAGING JARS

If you want to use persistent messaging you will need to add JARs to the broker's **CLASSPATH** for the desired persistence store. The JAR names follow the pattern **activemq-*store*-store**. The following message stores are included:

- **activemq-amq-store.jar**
- **activemq-jdbc-store.jar**
- **activemq-kahadb-store.jar**
- **activemq-leveldb-store.jar**

Additionally, you will need to include any other JARs required by the persistence manager used by the store:

- For KahaDB you will need **kahadb.jar**.
- For JDBC you will need the JARs for your database's JDBC driver.

INDEX

A

Active, Subscription statistics

activemq.xml, Editing the configuration template

administration client
  running, Running the administration client

amq, Starting in console mode

AverageEnqueueTime, Destination statistics

B

BlockedProducerWarningInterval, Destination statistics

broker
  addConnector, Broker actions
  addNetworkConnector, Broker actions
  addQueue, Broker actions
  addTopic, Broker actions
  createDurableSubscriber, Broker actions
  destroyDurableSubscriber, Broker actions
  disableStatistics, Broker actions
  enableStatistics, Broker actions
  gc, Broker actions
  reloadLog4jProperties, Broker actions
  removeConnector, Broker actions
  removeNetworkConnector, Broker actions
  removeQueue, Broker actions
  removeTopic, Broker actions
  resetStatistics, Broker actions
  start, Broker actions
  stop, Broker actions

stopGracefully, Broker actions

terminateJVM, Broker actions

useJmx, Enabling and disabling

BrokerId, Broker statistics

BrokerName, Broker statistics

BrokerVersion, Broker statistics

C

client, Running the administration client

command console

getting help, Using the broker console

remote access, Connecting a console to a remote broker

config shell, Editing the OSGi properties

config.properties, Overview

configuration

persistent identifier, OSGi PIDs

PID, OSGi PIDs

template, Configuration templates

connector

connectionCount, Connector actions

disableStatistics, Connector actions

enableStatistics, Connector actions

resetStatistics, Connector actions

start, Connector actions

stop, Connector actions

connectorHost, Advanced configuration

connectorPath, Advanced configuration

connectorPort, Advanced configuration

connectors, Activating a connector

console

config shell, Editing the OSGi properties

console mode

starting, Starting in console mode

stopping, Stopping the broker from console mode

ConsumerCount, Destination statistics

createConnector, Advanced configuration

createMBeanServer, Advanced configuration

CursorFull, Destination statistics

CursorMemoryUsage, Destination statistics

CursorPercentUsage, Destination statistics

D

daemon mode

starting, Starting in daemon mode

stopping, Stopping a broker running in daemon mode

DataDirectory, Broker statistics

deploying

standalone broker, Deploying a Standalone Broker

DequeueCount, Destination statistics

DequeueCounter, Subscription statistics

DispatchCount, Destination statistics

DispatchedCounter, Subscription statistics

DispatchedQueueSize, Subscription statistics

E

EnqueueCount, Destination statistics

EnqueueCounter, Subscription statistics

ExpiredCount, Destination statistics

F

fabric

starting a broker, Starting a broker in a fabric

stopping a broker, Shutting down remote brokers in a fabric

fabric:container-connect, Connecting a console to a remote broker

fabric:container-start, Starting a broker in a fabric

fabric:container-stop, Shutting down remote brokers in a fabric

fabric:container-upgrade, Applying a rollup patch, Applying an incremental patch

I

InFlightCount, Destination statistics

io.fabric8.mq.fabric.server.*, Activating a connector

J

JAAS
- configuration syntax, Configuring a JAAS realm
- converting to blueprint, Converting standard JAAS login properties to XML
- namespace, Namespace

jaas:config, Configuring a JAAS realm

jaas:module, Configuring a JAAS realm

JBoss Operations Network, Tools

jconsole, Tools

JMX
- disabling, Enabling and disabling
- roles, Securing access to JMX

jmxDomainName, Advanced configuration

L

logging
- console commands, Viewing the log with the console, Viewing the log with the administration client
- viewing as text, Viewing the log in a text editor
- viewing in an editor, Viewing the log in a text editor
- viewing in the console, Viewing the log with the console
- viewing with the admin client, Viewing the log with the administration client

M

management console, Tools

managementContext, Advanced configuration
- connectorHost, Advanced configuration
- connectorPath, Advanced configuration

connectorPort, Advanced configuration

createConnector, Advanced configuration

createMBeanServer, Advanced configuration

jmxDomainName, Advanced configuration

rmiServerPort, Advanced configuration

useMBeanServer, Advanced configuration

MaxEnqueueTime, Destination statistics

MaximumPendingMessageLimit, Subscription statistics

MaxPageSize, Destination statistics

MemoryLimit, Broker statistics, Destination statistics

MemoryPercentageUsed, Broker statistics, Destination statistics

MemoryUsagePortion, Destination statistics

MessageCountAwaitingAcknowledge, Subscription statistics

MinEnqueueTime, Destination statistics

N

network connector

start, Network connector actions

stop, Network connector actions

O

org.apache.karaf.log, Overview

org.ops4j.pax.logging, Overview

org.ops4j.pax.logging.DefaultServiceLog.level, Overview

osgi:shutdown, Using a remote console

P

patch:add, Applying a patch

patch:install, Applying a patch

patch:list, Applying a patch, Rolling back a patch

patch:rollback, Rolling back a patch

patch:simulate, Applying a patch

patching

fabric

command console, Applying a rollup patch, Applying an incremental patch

standalone, Applying a patch
rollback, Rolling back a patch

PendingQueueSize, Subscription statistics

persistent identifier, OSGi PIDs

PID, OSGi PIDs

PrefetchSize, Subscription statistics

ProducerCount, Destination statistics

Q

queue
browse, Queue actions
browseAsTable, Queue actions
browseMessages, Queue actions
copyMatchingMessagesTo, Queue actions
copyMessageTo, Queue actions
cursorSize, Queue actions
doesCursorHaveMessagesBuffered, Queue actions
doesCursorHaveSpace, Queue actions
getMessage, Queue actions
moveMatchingMessagesTo, Queue actions
moveMessageTo, Queue actions
purge, Queue actions
removeMatchingMessages, Queue actions
removeMessage, Queue actions
resetStatistics, Queue actions
retryMessage, Queue actions
sendTextMessage, Queue actions

QueueSize, Destination statistics

R

rmiServerPort, Advanced configuration

roles

JMX, Securing access to JMX

routine tasks, Routine tasks

S

shell, Starting a basic console

shutdown, Stopping the broker from console mode

ssh:ssh, Connecting a console to a remote broker, Using a remote console

standalone broker
- configuration template, Editing the configuration template
- deploying, Deploying a Standalone Broker
- runtime configuration, Editing the OSGi properties

start, Starting in daemon mode

stop, Stopping a broker running in daemon mode

StoreLimit, Broker statistics

StorePercentageUsed, Broker statistics

subscription
- browse, Subscription actions
- browseAsTable, Subscription actions
- cursorSize, Subscription actions
- destory, Subscription actions
- doesCursorHaveMessagesBuffered, Subscription actions
- doesCursorHaveSpace, Subscription actions
- isMatchingQueue, Subscription actions
- isMatchingTopic, Subscription actions

T

TempLimit, Broker statistics

TempPercentageUsed, Broker statistics

tooling, Tools

topic
- browse, Topic actions
- browseAsTable, Topic actions
- browseMessages, Topic actions

resetStatistics, Topic actions

sendTextMessage, Topic actions

TotalConsumerCount, Broker statistics

TotalDequeueCount, Broker statistics

TotalEnqueueCount, Broker statistics

TotalMessageCount, Broker statistics

TotalProducerCount, Broker statistics

transportConnector, Adding a transport connector definition

transportConnectors, Adding a transport connector definition

U

useJmx, Enabling and disabling

useMBeanServer, Advanced configuration

V

VisualVM, Tools