Red Hat JBoss A-MQ 6.1

Managing and Monitoring a Broker

Administrative tasks made simple

Last Updated: 2017-10-13

Red Hat JBoss A-MQ 6.1 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 JBoss A-MQ provides many tools to ensure that it is running at peak performance.

Table of Contents

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 JBoss A-MQ 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 JBoss A-MQ.

The following tools are included with JBoss A-MQ:

- 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 JBoss A-MQ 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.

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.

2.1. UNDERSTANDING THE RED HAT JBOSS A-MQ 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.

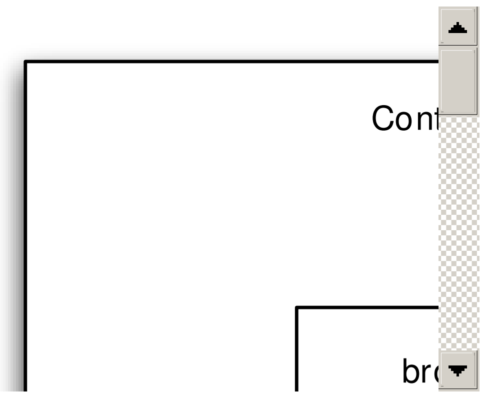
Overview

One of the weaknesses of the Apache ActiveMQ configuration model is that any changes require a broker restart. Red Hat JBoss A-MQ 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 JBoss A-MQ 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 JBoss A-MQ 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 JBoss A-MQ Configuration System



Configuration templates

The JBoss A-MQ configuration template is an XML file that is based on the Apache ActiveMQ configuration file. The main differences between an Apache ActiveMQ and a JBoss A-MQ 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 JBoss A-MQ Configuration”.

Example 2.1. Adding Property Placeholder Support to Red Hat JBoss A-MQ Configuration

```
<!-- Allows us to use system properties and fabric as variables in this
configuration file -->
<bean
class="org.springframework.beans.factory.config.PropertyPlaceholderConfi
gurer">
  <property name="properties">
    <bean class="org.fusesource.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 JBoss A-MQ instance have the prefix **`org.fusesource.mq.fabric.server`**.

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

2.2. EDITING A STANDALONE BROKER'S CONFIGURATION

Abstract

A standalone Red Hat JBoss A-MQ 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/org.fusesource.mq.fabric.server-default.cfg`** file.

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

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

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

In addition to the broker's messaging behavior, a number of the broker's runtime behavior such as logging levels, the 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

2.3. MODIFYING A RUNNING STANDALONE BROKER'S XML CONFIGURATION

Abstract

A select set of properties in a standalone Red Hat JBoss A-MQ 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/org.fusesource.mq.fabric.server-default.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 (see the section called "Editing the OSGi properties" .

- 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

Attribute	Type	Queues	Topics
`blockedProducerWarningInterval`	long	Y	N
`consumersBeforeDispatchStarts`	int	Y	N
`cursorMemoryHighWaterMark`	int	Y	N
`doOptimizeMessageStore`	boolean	Y	N
`gcIsInactiveDestinations`	boolean	Y	N
`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 (see).

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" />
          ...
```

2.4. EDITING A BROKER'S CONFIGURATION IN A FABRIC

Abstract

Red Hat JBoss A-MQ supports deploying brokers into a cluster called a fabric. When a broker is deployed into a fabric, the Fabric Agent controls its configuration and you must use special commands to update it.

Overview

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

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

Profiles

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

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

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

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

Procedure

The recommended approach to configuring brokers in a fabric is:

1. Create a configuration template.

   See Section 2.1, “Understanding the Red Hat JBoss A-MQ Configuration Model” .

2. Create a base profile for all of the brokers in the fabric using the configuration template.

3. Create profiles that inherit from the base profile that will be assigned to one or more brokers.

4. Modify the properties in each of the profiles to the desired values for the brokers to which the profile will be assigned.

5. Assign the new profiles to the desired brokers.

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

Creating a base profile

To create a base profile:

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

   This will create a new copy of the existing profiles.

2. Import the new XML template into the registry using the **`fabric:import`** command as shown in Example 2.4, “Creating a Profile Using an XML Configuration Template” .

   Example 2.3. Importing an XML Configuration Template

   ```
   JBossA-MQ:karaf@root> fabric:import -t
   /fabric/configs/versions/version/profiles/mq-base/xmlTemplate
   xmlTemplatePath
   ```

3. Create a new configuration profile instance to hold the new XML template using the **`fabric:mq-create`** command as shown in Example 2.4, “Creating a Profile Using an XML Configuration Template”.

   Example 2.4. Creating a Profile Using an XML Configuration Template

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

   This will create a new profile that is based on the default broker profile but uses the imported XML template.

Creating deployment profiles and assigning them to brokers

To create deployment profiles and assigned them to the brokers:

1. Create new profile using the **`fabric:profile-create`** command as shown in Example 2.5, “Creating a Deployment Profile”.

   Example 2.5. Creating a Deployment Profile

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

2. Add values for the property placeholders using the **`fabric:profile-edit`** command as shown in Example 2.6, “Setting Properties in a Profile” .

Example 2.6. Setting Properties in a Profile

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

The fabric properties for a broker are specified using the PID **`org.fusesource.mq.fabric.server-profileName`**, so to specify a value for the broker-name property for the profile called **`myBroker`** you would use the command shown in Example 2.7, "Setting the Broker Name Property".

Example 2.7. Setting the Broker Name Property

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

3. Assign the new profile to one or more brokers using the **`fabric:container-add-profile`** command as shown in Example 2.8, "Assigning a Profile to a Broker".

Example 2.8. Assigning a Profile to a Broker

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

Using the management console

The management console simplifies the process of configuring brokers in a fabric by providing an easy to use Web-based interface and reducing the number of steps required to make the changes. For more information on using the management console see *Using the Management Console*

CHAPTER 3. SECURITY BASICS

Abstract

By default, Red Hat JBoss A-MQ 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

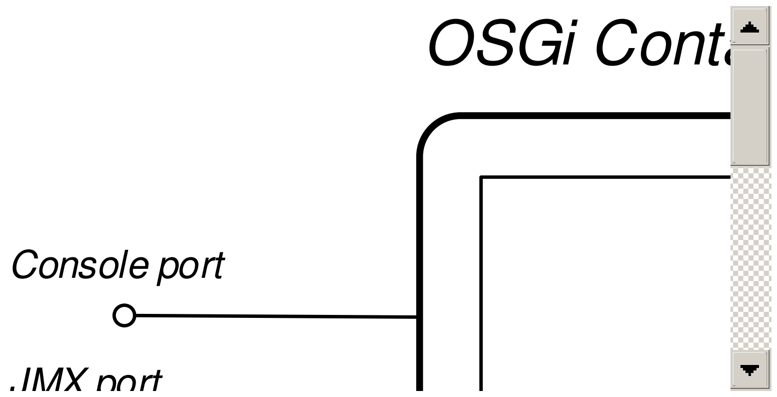
Overview

The Red Hat JBoss A-MQ 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 Red Hat JBoss A-MQ container

Figure 3.1, "Ports Exposed by the Red Hat JBoss A-MQ Container" shows the ports exposed by the JBoss A-MQ container by default.

Figure 3.1. Ports Exposed by the Red Hat JBoss A-MQ 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 JBoss A-MQ 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. JBoss A-MQ'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 JBoss A-MQ 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,admin
```

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

WARNING

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

Assigning roles for remote access

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

- SSH (remote console login)

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

  ```
  sshRole=admin
  ```

- JMX management

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

  ```
  jmxRole=jmx
  ```

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

CHAPTER 4. SECURING A STANDALONE RED HAT JBOSS A-MQ CONTAINER

Abstract

The Red Hat JBoss A-MQ 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 JBoss A-MQ 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.

`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 JBoss A-MQ uses the same properties as a standard Java login configuration file, however Red Hat JBoss A-MQ requires that they are specified slightly differently. To see how the Red Hat JBoss A-MQ 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 JBoss A-MQ 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
className="org.apache.activemq.jaas.PropertiesLoginModule"
flags="required">
            org.apache.activemq.jaas.properties.user=users.properties
            org.apache.activemq.jaas.properties.group=groups.properties
        </jaas:module>
    </jaas:config>

</blueprint>
```

IMPORTANT

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

Example

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

For a detailed description and example of using the LDAP login module, see Section 4.2, "Enabling LDAP Authentication".

4.2. ENABLING LDAP AUTHENTICATION

Overview

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

Procedure

To enable JBoss A-MQ to use LDAP for user authentication you need to create a JAAS realm that includes the JBoss A-MQ LDAP login module. As shown in Example 4.6, "Red Hat JBoss A-MQ LDAP JAAS Login Module", this is done by adding a **jaas:module** element to the realm and setting its **className** attribute to **org.apache.karaf.jaas.modules.ldap.LDAPLoginModule**.

Example 4.6. Red Hat JBoss A-MQ LDAP JAAS Login Module

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

You will also need to provide values for the properties described in Table 4.2, "Properties for the Red Hat JBoss A-MQ LDAP Login Module".

LDAP properties

Table 4.2, "Properties for the Red Hat JBoss A-MQ LDAP Login Module" describes the properties used to configure the Red Hat JBoss A-MQ JAAS LDAP login module.

Table 4.2. Properties for the Red Hat JBoss A-MQ LDAP Login Module

Property	Description
connection.url	Specifies specify the location of the directory server using an ldap URL, ldap://*Host*:*Port*. You can optionally qualify this URL, by adding a forward slash, `/`, followed by the DN of a particular node in the directory tree.
connection.username	Specifies the DN of the user that opens the connection to the directory server. For example, **uid=admin,ou=system**.
connection.password	Specifies the password that matches the DN from connection.username. In the directory server, the password is normally stored as a **userPassword** attribute in the corresponding directory entry.

Property	Description
user.base.dn	Specifies the DN of the subtree of the DIT to search for user entries.
user.filter	Specifies the LDAP search filter used to locate user credentials. It is applied to the subtree selected by user.base.dn. Before being passed to the LDAP search operation, the value is subjected to string substitution such that all occurrences of `%u` are replaced by the user name extracted from the incoming credentials.
user.search.subtree	Specifies if the user entry search's scope includes the subtrees of the tree selected by user.base.dn.
role.base.dn	Specifies the DN of the subtree of the DIT to search for role entries.
role.filter	Specifies the LDAP search filter used to locate roles. It is applied to the subtree selected by role.base.dn. Before being passed to the LDAP search operation, the value is subjected to string substitution such that all occurrences of `%u` are replaced by the user name extracted from the incoming credentials.
role.name.attribute	Specifies the attribute type of the role entry that contains the name of the role/group. If you omit this option, the role search feature is effectively disabled.
role.search.subtree	Specifies if the role entry search's scope includes the subtrees of the tree selected by role.base.dn.
authentication	Specifies the authentication method used when binding to the LDAP server. Valid values are • `simple`—bind with user name and password authentication • `none`—bind anonymously
initial.context.factory	Specifies the class of the context factory used to connect to the LDAP server. This must always be set to `com.sun.jndi.ldap.LdapCtxFactory`.
ssl	Specifies if the connection to the LDAP server is secured via SSL. If connection.url starts with ldaps:// SSL is used regardless of this property.

Property	Description
ssl.provider	Specifies the SSL provider to use for the LDAP connection. If not specified, the default SSL provider is used.
ssl.protocol	Specifies the protocol to use for the SSL connection. You *must* set this property to **TLSv1**, in order to prevent the SSLv3 protocol from being used (POODLE vulnerability).
ssl.algorithm	Specifies the algorithm used by the trust store manager.
ssl.keystore	Specifies the keystore name.
ssl.keyalias	Specifies the name of the private key in the keystore.
ssl.truststore	Specifies the trust keystore name.

All of the properties are mandatory except the SSL properties.

Example

Example 4.7, "Configuring a JAAS Realm that Uses LDAP Authentication" defines a JASS realm that uses the LDAP server located at ldap://localhost:10389.

Example 4.7. Configuring a JAAS Realm that Uses LDAP Authentication

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

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

IMPORTANT

You must set `ssl.protocol` to `TLSv1`, in order to protect against the Poodle vulnerability (CVE-2014-3566)

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.

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

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

Encrypted properties

Encrypted properties are stored in plain properties files. They are identified by wrapping them in the `ENC()` function as shown in Example 4.8, "Property File with an Encrypted Property".

Example 4.8. Property File with an Encrypted Property

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

IMPORTANT

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

Namespaces

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

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

Example 4.9, "Encrypted Property Namespaces" shows a blueprint file with the required namespaces.

Example 4.9. Encrypted Property Namespaces

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

Placeholder extension

In order to use encrypted property placeholders in a blueprint file you need to include an Aries **`property-paceholder`** element to you blueprint file. As shown in Example 4.10, "Aries Placeholder Extension", it must come before the Jasypt configuration or the use of placeholders.

Example 4.10. Aries Placeholder Extension

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

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

...
</blueprint>
```

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

Jasypt configuration

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

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

For detailed information on how to configure the different Jasypt encryptors, see the Jasypt documentation.

Example 4.11, "Jasypt Blueprint Configuration" shows configuration for using the string encryptor and retrieving the password from an environment variable.

Example 4.11. Jasypt Blueprint Configuration

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

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

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

Placeholders

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

Example 4.12, "Jasypt Blueprint Configuration" shows an LDAP JAAS realm that uses the properties file in Example 4.8, "Property File with an Encrypted Property".

Example 4.12. Jasypt Blueprint Configuration

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:ext="http://aries.apache.org/blueprint/xmlns/blueprint-
ext/v1.0.0"
  xmlns:enc="http://karaf.apache.org/xmlns/jasypt/v1.0.0">
  <ext:property-placeholder>
    <location>file://ldap.properties</location>

  <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="FUSE_ENCRYPTION_PASSWORD" />
        </bean>
      </property>
    </enc:encryptor>
  </enc:property-placeholder>

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

connectionUsername=cn=mqbroker,ou=Services,ou=system,dc=fusesource,dc=co
m
        connectionPassword=${ldap.password}
        connectionProtocol=
        authentication=simple
        userRoleName=cn
        userBase = ou=User,ou=ActiveMQ,ou=system,dc=fusesource,dc=com
        userSearchMatching=(uid={0})
        userSearchSubtree=true
        roleBase = ou=Group,ou=ActiveMQ,ou=system,dc=fusesource,dc=com
        roleName=cn
        roleSearchMatching= (member:=uid={1})
        roleSearchSubtree=true
    </jaas:module>
  </jaas:config>

</blueprint>
```

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

Installing the Jasypt features

By default, Red Hat JBoss A-MQ does not have the Jasypt encryption libraries installed. In order to use encrypted property placeholders, you will need to install the **`jasypt-encryption`** feature using the **`features:install`** command as shown in Example 4.13, “Installing the Jasypt Feature” .

Example 4.13. Installing the Jasypt Feature

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

4.4. CONFIGURING ROLES FOR THE ADMINISTRATIVE PROTOCOLS

Overview

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

Administration protocols

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

- SSH (remote console login)
- JMX management

Default role

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

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

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

Changing the remote console's role

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

```
sshRole=admin
```

Changing the JMX role

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

CHAPTER 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.

The default fabric authentication mechanism uses a text-based authentication cache that is stored on the fabric's registry. This authentication mechanism is used to control who can access fabric containers, who can issue commands to remote containers, who can update fabric profiles, and who can retrieve the details on the container's in the fabric. The management console also uses the fabric's authentication scheme.

The authentication system uses the JAAS framework. The fabric uses the `karaf` JAAS realm. This realm is defined in the `fabric-jaas` feature. It is deployed to all containers in the fabric as part of the `default` profile.

IMPORTANT

The default authentication system is not recommended for production use.

You can configure the fabric to use a different authentication mechanism by overriding the `karaf` JAAS realm to use a proper login module. The fabric containers include an LDAP module that is preloaded and simply needs to be activated. Doing so requires that the new JAAS realm be defined in OSGi blueprint files and deployed to the container's in the fabric.

5.1. USING THE DEFAULT AUTHENTICATION SYSTEM

Abstract

The default authentication system for the fabric can be managed using management console's `Users`. It can also be enhanced to store passwords using encryption.

By default fabric uses a simple text-based authentication system. 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.

You can mange the users in the default realm using the `Users` tab. You can also strengthen the default system by configuring it to use encryption when storing user credentials.

5.1.1. Managing Users

Viewing user data

To view the users configured to access fabric select the **Users** item from the main menu. This will open the `Users` page.

The `Users` page lists all of the users along the left hand side of the page. If you select the user, the user's roles will be displayed along the right hand of the page and you the buttons to edit the user will become active.

Adding a user

To add a user:

1. Click **Users** from the main menu.

   The **`Users`** page opens.

2. Click **`Create User`**.

   The **`Create New User`** dialog opens.

3. In the **`Username`** field, enter a unique name for the user.

   The user name must be at least five characters long.

4. In the **`Password`** field, enter a password for the user.

   The password must be atleast six characters long.

5. In the **`Repeat Password`** field, reenter the password for the user.

6. Click **`Create`**.

   The dialog closes and the new user is added to the list.

Removing a user

To remove a user:

1. Click **Users** from the main menu.

   The **`Users`** page opens.

2. Select the user to be deleted from the list.

3. Click **`Delete User`**.

   A confirmation dialog opens.

4. Click **Yes**.

   The dialog closes and the user is removed from the list.

Changing a user's password

To change a user's password:

1. Click **Users** from the main menu.

   The **`Users`** page opens.

2. Select the user to update from the list.

3. Click **`Change Password`**.

   The **`Change Password`** dialog opens.

4. In the **`Password`** field, enter a password for the user.

   The password must be at least six characters long.

5. In the **`Repeat Password`** field, reenter the password for the user.

6. Click **Change**.

   The dialog closes and a message shows that the password was changed.

Adding a role to a user

NOTE

Roles are not enforced.

To add a role to a user:

1. Click **Users** from the main menu.

   The **`Users`** page opens.

2. Select the user to update from the list.

3. Click **`Add Role`**.

   The **`Add New Role`** dialog opens.

4. In the **`Role Name`** field, enter a role for the user.

5. Click **Add**.

   The dialog closes and a message shows that the role was added.

Deleting a role from a user

NOTE

Roles are not enforced.

To delete a role from a user:

1. Click **Users** from the main menu.

   The **`Users`** page opens.

2. Select the user to update from the list.

   The user's roles are listed on the right side of the page.

3. Click the **X** opposite the role to delete.

   A confirmation dialog opens.

4. Click **Yes**.

   The dialog closes and the role is removed from the list.

5.1.2. Encrypting Stored Passwords

Overview

By default, the JAAS login modules store passwords as plain text. You can provide additional protection to passwords by storing them in an encrypted format. This can be done by adding the appropriate JAAS configuration to the profile defining the fabric's **karaf** realm. 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.

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.

Editing the default profile

The easiest way to update all of the containers in a fabric is to modify the **default** profile. It is applied to all containers in a fabric through inheritance.

IMPORTANT

Before making changes to the **default** profile, you should create a new profile version. Editing the active **default** profile could make your fabric unstable.

To enable password encryption by editing the **default** profile:

1. Select the management console's **Profiles** tab.

2. Create a new version by clicking **Create Version**.

   The **Create New Version** dialog appears.

3. Optionally enter a name for the new version in the **Name**

   If you leave the **Name** field blank, the management console will use the name of the parent version and append the next available version number, starting from 1, to the name. For example, if version **brokers1** existed, the management console would create version **brokers1.1**.

4. From the **Parent Version** list, select the version from which the new version will originate.

   By default **Use most recently created version** is selected.

5. Click **Create** to create the version.

   The dialog will close and the new version will be listed in the **Versions** table.

6. Select the new version from the **Versions** list.

7. Select the **default** profile from the **Profiles** list.

   The **Profiles** page for the selected profile opens.

8. Select the **Config Files** tab.

9. Select **io.fabric8.jaas** from the list.

   A text editing window opens.

10. Enter values for the properties described in Table 5.1, "JAAS Encryption Properties" following the example shown in Example 5.1, "Password Encryption Configuration".

    Example 5.1. Password Encryption Configuration

    ```
    encryption.enabled = true
    encryption.name = jasypt
    encryption.algorithm = SHA-256
    encryption.encoding = base64
    encryption.iterations = 100000
    encryption.saltSizeBytes = 16
    ```

11. Click **Save**.

12. If you configure the encryption to use Jasypt, add the **jasypt-encryption** feature to the profile.

    a. Select the **Repositories** tab.

    b. In the **Add repository** field enter **mvn:org.apache.karaf.assemblies.features/standard/2.3.0.fuse-71-044/xml/features**.

    c. Click **Add**.

    d. Select the **Features** tab.

    e. From the **Repository** list select **mvn:org.apache.karaf.assemblies.features/standard/2.3.0.fuse-71-044/xml/features**.

    f. From the **Features** list select **jasypt-encryption**.

13. Select the management console's **Containers** tab.

14. Click **Migrate Containers**.

15. Click **Select All**.

16. Select the version to which you added encryption from the **Target Version** table.

17. Click **Apply**.

Configuration properties

Table 5.1, “JAAS Encryption Properties” describes the properties used to enable password encryption.

Table 5.1. JAAS Encryption Properties

Property	Description
`encryption.enabled`	Specifies if password encryption is enabled.
`encryption.name`	Specifies the name of the encryption service, which has been registered as an OSGi service. See the section called “Encryption services”.
`encryption.prefix`	Specifies the prefix for encrypted passwords.
`encryption.suffix`	Specifies the suffix for encrypted passwords.
`encryption.algorithm`	Specifies the name of the encryption algorithm—for example, `MD5` or `SHA-1`. You can specify one of the following encryption algorithms: • `MD2` • `MD5` • `SHA-1` • `SHA-256` • `SHA-384` • `SHA-512`
`encryption.encoding`	Specifies the encrypted passwords encoding: `hexadecimal` or `base64`.
`encryption.providerName` *(Jasypt only)*	Name of the `java.security.Provider` instance that is to provide the digest algorithm.
`encryption.providerClassName` *(Jasypt only)*	Specifies the class name of the security provider that is to provide the digest algorithm.
`encryption.iterations` *(Jasypt only)*	Specifies the number of times to apply the hash function recursively.
`encryption.saltSizeBytes` *(Jasypt only)*	Specifies the size of the salt used to compute the digest.
`encryption.saltGeneratorClassName` *(Jasypt only)*	Specifies the class name of the salt generator.

Property	Description
`role.policy`	Specifies the policy for identifying role principals. Can have the values, `prefix` or `group`.
`role.discriminator`	Specifies the discriminator value to be used by the role policy.

Encryption services

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

- Basic encryption service—installed in the standalone container by default and you can reference it by setting the `encryption.name` property to the value, `basic`. In the basic encryption service, the message digest algorithms are provided by the SUN security provider (the default security provider in the Oracle JDK).
- Jasypt encryption—can be installed in the standalone container by installing the `jasypt-encryption` feature. To access the Jasypt encryption service, set the `encryption.name` property to the value, `jasypt`.

  For more information about Jasypt encryption, see the Jasypt documentation.

5.2. DEFINING JAAS REALMS IN A FABRIC

Overview

Fabric containers, like standalone containers, use a special `jaas:config` element for defining JAAS realms. The difference is that when containers are deployed in a fabric, the JAAS realms need to be pushed out the fabric registry and *all* of the containers in the fabric need to share the same JAAS realms.

Procedure

To change the JAAS realm used by the fabric containers for authentication:

1. Create a Maven project to package and deploy the JAAS realm to the fabric's Maven proxy as shown in the section called “Creating a Maven project for deploying a JAAS realm” .
2. Create a JAAS realm that uses the LDAP login module:
   a. Open the blueprint XML file in `src/main/resources/OSGI-INF/my-service.xml` in a text editor.
   b. Delete the `bean` and `service` elements.
   c. Add a `jaas:config` element to the blueprint.

      See Section 4.1, “Defining JAAS Realms” for details on configuring the JAAS realm.

3. Deploy the JAAS realm to the fabric's maven proxy using the `mvn deploy` command.

4. In the management console create a new profile for deploying the new realm.

5. Select the **`Bundles`** tab.

6. Add the bundle you uploaded for the JAAS realm to the profile.

   Bundles are specified using Maven URLs. For example if your project's group ID is `my.jaas.realm` and the artifact ID is `jaas`, the Maven URL for the bundle will be mvn:my.jaas.realm/ldap/*version*.

7. Add the new profile to **all** of the containers in the fabric.

Creating a Maven project for deploying a JAAS realm

To create a Maven project for deploying a JAAS realm to a fabric registry:

1. Use the `karaf-blueprint-archetype` archetype to generate a template project.

   Example 5.2, “Create a Maven Project” shows how to invoke the archetype from the command line.

   Example 5.2. Create a Maven Project

   ```
   mvn archetype:generate -
   DarchetypeGroupId=org.apache.karaf.archetypes -
   DarchetypeArtifactId=karaf-blueprint-archetype -DgroupId=groupID -
   DartifactId=artifactID
   ```

2. Remove the project's `src/main/java` folder.

   This folder holds template Java classes for implementing an OSGi service, but the JAAS realm does not require any Java classes.

3. Open the project's POM in a text editor.

4. Add a `distributionManagement` element, similar to the one shown in Example 5.3, “Fabric Maven Proxy Distribution Settings”, to the POM.

   Example 5.3. Fabric Maven Proxy Distribution Settings

   ```
   <distributionManagement>
     <repository>
       <id>fabric-maven-proxy</id>
       <name>FMC Maven Proxy</name>

   <url>http://username:password@localhost:8107/maven/upload/</url>
     </repository>
   </distributionManagement>
   ```

You will need to modify the **url** element to include the connection details for your environment:

- The *username* and *password* are the credentials used access the Fabric Server to which you are trying to connect.
- The hostname, **localhost** in Example 5.3, "Fabric Maven Proxy Distribution Settings", is the address of the machine hosting the Fabric Server.
- The port number, **8107** in Example 5.3, "Fabric Maven Proxy Distribution Settings", is the port number exposed by the Fabric Server. **8107** is the default setting.
- The path, **/maven/upload/** in Example 5.3, "Fabric Maven Proxy Distribution Settings", is the same for all Fabric Servers.

5.3. ENABLING LDAP AUTHENTICATION

Abstract

Fabric containers come with a preinstalled LDAP login module. To activate it you need to reconfigure the default JAAS realm to use the LDAP login module and associate the new realm with all of the containers in the fabric.

Overview

Fabric containers supply a JAAS login module that enables it to use LDAP to authenticate users. The JAAS LDAP login module is implemented by the **org.apache.karaf.jaas.modules.ldap.LDAPLoginModule** class. It is preloaded by the containers, so you do not need to install its bundle.

To enable LDAP authentication, you need to create a new profile that redefines the default **karaf** realm to use the LDAP login module and deploy it to every container in the management console's fabric. Once this is done, all access to the management console Fuse Management Console, **and all of the fabric containers' command consoles**, will be authenticated against your LDAP server.

Procedure

To enable the fabric containers to use LDAP for user authentication:

1. Create a Maven project to package and deploy the LDAP JAAS realm to the fabric's Maven proxy as shown in the section called "Creating a Maven project for deploying a JAAS realm" .
2. Create a JAAS realm that uses the LDAP login module:
   a. Open the blueprint XML file in **src/main/resources/OSGI-INF/my-service.xml** in a text editor.
   b. Delete the **bean** and **service** elements.
   c. Add a **jaas:config** element to the blueprint.
   d. Add a **name** attribute to the **jaas:config** element and set its value to **karaf**.

> **NOTE**
>
> This will override the default realm used by the container.

e. Add a **`rank`** attribute to the **`jaas:config`** element and set its value to **`5`**.

> **NOTE**
>
> This will insure that this realm is used by the container.

f. Add a **`jaas:module`** element to the **`jaas:config`** element.

g. Add a **`className`** attribute to the **`jaas:module`** element and set its value to **`org.apache.karaf.jaas.modules.ldap.LDAPLoginModule`**.

   Example 5.4, "LDAP JAAS Login Module" shows the blueprint file.

   Example 5.4. LDAP JAAS Login Module

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

h. Provide values for the properties described in Table 4.2, "Properties for the Red Hat JBoss A-MQ LDAP Login Module".

3. Deploy the JAAS realm to the fabric's maven proxy using the **`mvn deploy`** command.

4. In the management console create a new profile for deploying the LDAP realm.

   See *Using the Management Console*

5. Select the **`Bundles`** tab.

6. Add the bundle you uploaded for the JAAS realm to the profile.

   Bundles are specified using Maven URLs. For example if your project's group ID is **`my.jaas.realm`** and the artifact ID is **`ldap`**, the Maven URL for the bundle will be mvn:my.jaas.realm/ldap/*version*.

   See *Using the Management Console*

7. Add the new profile to **all** of the containers in the fabric.

   See *Using the Management Console*

CHAPTER 6. INSTALLING RED HAT JBOSS A-MQ AS A SERVICE

Abstract

Red Hat JBoss A-MQ can generate a service wrapper that can be easily configured to install Red Hat JBoss A-MQ as a system service.

To install Red Hat JBoss A-MQ as a system service, perform the following steps:

1. Generate the service wrapper for your system.

   See Section 6.1, "Generating the Wrapper".

2. Configure the launch script for your system.

   See Section 6.2, "Configure the Script" .

3. Configure the service wrapper for your system.

   See Section 6.3, "Configuring the Wrapper".

4. Install the service wrapper as system service.

   See Section 6.4, "Installing and Starting the Service" .

6.1. GENERATING THE WRAPPER

Abstract

The service wrapper is generated by the Apache Karaf container using the `wrapper:install` command.

Overview

The Red Hat JBoss A-MQ console's `wrapper` feature generates a wrapper around the JBoss A-MQ runtime that allows you to install a message broker as a system service. The `wrapper` feature does not come preinstalled in the console, so before you can generate the service wrapper you must install the `wrapper` feature.

Once the feature is installed the console gains a `wrapper:install` command. Running this command generates a generic service wrapper in the JBoss A-MQ installation.

Procedure

To generate the service wrapper:

1. Start JBoss A-MQ in console mode using the **amq** command.

2. Once the console is started and the command prompt appears, enter `features:install wrapper`.

The `features:install` command will locate the required libraries to provision the wrapper feature and deploy it into the run time.

3. Generate the wrapper by entering `wrapper:install -n` ***`serviceName`*** `-d` ***`displayName`*** `-D` ***`description`***.

   The `wrapper:install` command has the options described in Table 6.1, "Wrapper Install Options".

   Table 6.1. Wrapper Install Options

Option	Default	Description
`-s`	`AUTO_START`	*(Windows only)* Specifies the mode in which the service is installed. Valid values are `AUTO_START` or `DEMAND_START`.
`-n`	`karaf`	Specifies the service name that will be used when installing the service.
`-d`		Specifies the display name of the service.
`-D`		Specifies the description of the service.

Generated files

The following files are generated and make up the service wrapper:

- `bin\`*`ServiceName`*`-wrapper[.exe]`—the executable file for the wrapper.
- `bin\`*`ServiceName`*`-service[.bat]`—the script used to install and remove the service.
- `etc\`*`ServiceName`*`-wrapper.conf`—the wrapper's configuration file.
- Three library files required by the service wrapper:
  - `lib\libwrapper.so`
  - `lib\karaf-wrapper.jar`
  - `lib\karaf-wrapper-main.jar`

6.2. CONFIGURE THE SCRIPT

Abstract

The service launch script, ***ServiceName*-service[.bat]** file, is located under the ***InstallDir*/bin/** directory.

Overview

There are a few environment variables you can optionally customize in the ***ServiceName*-service[.bat]** file, as described here.

RUN_AS_USER

When the **RUN_AS_USER** variable is set, the ***ServiceName*-service[.bat]** script runs as the specified user. For example, to run the script as the user, **mquser**, search for the line, **#RUN_AS_USER**, uncomment the line, and set the variable as follows:

```
RUN_AS_USER=mquser
```

IMPORTANT

Make sure that the specified user has the required privileges to write the PID file and **wrapper.log** files. Failure to be able to write the log file will cause the Wrapper to exit without any way to write out an error message.

PRIORITY

(LINUX and UNIX only) You can optionally assign a priority to the launched service using the system **nice** command by setting the **PRIORITY** variable (it is not set by default).

6.3. CONFIGURING THE WRAPPER

Abstract

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

Overview

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

There are several settings you may want to change including:

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

Specifying the Red Hat JBoss A-MQ's environment

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

- **`KARAF_HOME`**—the location of the Red Hat JBoss A-MQ install directory.
- **`KARAF_BASE`**—the root directory containing the configuration and OSGi data specific to the broker instance.

  The configuration for the broker instance is stored in the ***`KARAF_BASE`*****`/conf`** directory. Other data relating to the OSGi runtime is also stored beneath the base directory.
- **`KARAF_DATA`**—the directory containing the logging and persistence data for the broker.

Example 6.1, "Default Environment Settings" shows the default values.

Example 6.1. Default Environment Settings

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

Passing parameters to the JVM

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

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

Example 6.2, "Default Java System Properties" shows the default Java properties.

Example 6.2. Default Java System Properties

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

Adding classpath entries

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

Example 6.3, "Default Wrapper Classpath" shows the default classpath entries.

Example 6.3. Default Wrapper Classpath

```
wrapper.java.classpath.1=%KARAF_BASE%/lib/karaf-wrapper.jar
wrapper.java.classpath.2=%KARAF_HOME%/lib/karaf.jar
wrapper.java.classpath.3=%KARAF_HOME%/lib/karaf-jaas-boot.jar
wrapper.java.classpath.4=%KARAF_BASE%/lib/karaf-wrapper-main.jar
```

JMX configuration

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

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

   There are three properties, shown in Example 6.4, "Wrapper JMX Properties", that are used to configure JMX.

   Example 6.4. Wrapper JMX Properties

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

2. Remove the # from in front of each of the properties.

3. Replace the **n** in each property to a number that fits into the sequence of addition properties established in the configuration.

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

For more information about using JMX see Chapter 15, *Using JMX*.

Configuring logging

The wrapper's logging in configured using the properties described in Table 6.2, "Wrapper Logging Properties".

Table 6.2. Wrapper Logging Properties

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

For more information about Red Hat JBoss A-MQ logging see Chapter 14, *Using Logging*.

6.4. INSTALLING AND STARTING THE SERVICE

Overview

The operating system determines the exact steps using to complete the installation of Red Hat JBoss A-MQ as a service. The `wrapper:install` command provides basic instructions for your operating system.

Windows

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

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

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

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

Redhat Linux

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

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

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

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

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

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

Ubuntu Linux

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

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

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

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

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

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

CHAPTER 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/org.fusesource.mq.fabric.server-default.cfg** file.

STARTING IN CONSOLE MODE

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

IMPORTANT

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

To launch a broker in console mode, change to *InstallDir* and run one of the commands in Table 7.1, “Start up Commands for Console Mode”.

Table 7.1. Start up Commands for Console Mode

`Windows`	`bin\amq.bat`
`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

```
______ ___ ___ _____ | ___| | \/ || _ | | |_ _ _ ___ ___ | . . || | | |
| _|| | | |/ __| / _ \ | |\/| || | | | | | | |_| |\__ \| __/ | | | |\
\/' / \_| \__,_||___/ \___| \_| |_/ \_/\_\ Fuse MQ (7.0.0.fuse-036)
http://fusesource.org/mq/ Hit '<tab>' for a list of available commands
```

```
and '[cmd] --help' for help on a specific command. Hit '<ctrl-d>' or
'osgi:shutdown' to shutdown Fuse MQ.
JBossA-MQ:karaf@root>
```

STARTING IN DAEMON MODE

Launching a broker in daemon mode runs Red Hat JBoss A-MQ 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`**
`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 JBoss A-MQ'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 JBoss A-MQ 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 JBoss A-MQ 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 JBoss A-MQ 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 JBoss A-MQ 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 JBoss A-MQ 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.

Argument	Description
`-p`	Specifies the password used to log into the broker.
`-v`	Use verbose output.
`-r`	Specifies the maximum number of attempts to establish a connection.
`-d`	Specifies, in seconds, the delay between retries. The default is 2 seconds.
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 JBoss A-MQ 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 JBoss A-MQ 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 JBoss A-MQ. Using a fabric, however, you can deploy multiple brokers from a single location and easily reuse large portions of the configuration.

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 JBoss A-MQ 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.

9.1. DEPLOYING A STANDALONE BROKER

Abstract

Deploying standalone brokers requires manually installing and configuring multiple instances of Red Hat JBoss A-MQ.

Overview

Deploying a new standalone broker involves installing Red Hat JBoss A-MQ 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 JBoss A-MQ 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*

9.2. DEPLOYING A NEW BROKER INTO A FABRIC

Abstract

Deploying a broker into a fabric allows you to deploy multiple brokers from a single location and easily reuse large portions of the configuration.

Overview

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

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

From the Red Hat JBoss A-MQ console you can use the **`fabric:mq-create`** to create new broker profiles and new containers for the brokers. You can also use the management console to perform to create the profiles and assign them to containers.

Procedure

To deploy a new broker into a fabric:

1. Create a template JBoss A-MQ XML configuration file in a location that is accessible to the container.

   See Section 2.1, "Understanding the Red Hat JBoss A-MQ Configuration Model" .

2. In the command console, use the **`fabric:import`** command to upload the your XML configuration template to the Fabric Ensemble as shown in Example 9.1, "Uploading a Template to a Fabric Ensemble".

   Example 9.1. Uploading a Template to a Fabric Ensemble

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

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

3. Use the **`fabric:mq-create`** command to create a profile for the new broker and assign it to a container.

   - To deploy the new broker into an existing container use the command shown in Example 9.2, “Creating a New Broker in an Existing Container”

     Example 9.2. Creating a New Broker in an Existing Container

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

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

   - To deploy the new broker into an new container use the command shown in Example 9.3, “Creating a New Broker in a New Container”

     Example 9.3. Creating a New Broker in a New Container

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

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

     NOTE

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

   You can add network configuration settings to the profile as well.

4. Use the **`fabric:profile-edit`** command shown in Example 9.4, “Editing a Broker Profile” to set the required properties.

   Example 9.4. Editing a Broker Profile

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

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

NOTE

The management console makes this process easier by providing a Web-based UI.

More information

For more information on configuring brokers to work together see:

- *Using Networks of Brokers*
- *Fault Tolerant Messaging*

For more information on using the management console, see *Using the Management Console*.

CHAPTER 10. SHUTTING DOWN A BROKER

Abstract

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

IMPORTANT

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

10.1. 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 JBoss A-MQ.

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 10.1, “Using the Console's Shutdown Command” .

Example 10.1. 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 10.2, “Shutting Down a Broker Remotely”.

10.2. 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 JBoss A-MQ 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 JBoss A-MQ on your local host by running the `stop` command in the ***`InstallDir`*****`/bin`** directory. The commands syntax is shown in Example 10.2, “Stop Command Syntax”.

Example 10.2. Stop Command Syntax

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

`-a` ***`port`***

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

`-h` ***`hostname`***

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

`-u` ***`username`***

Specifies the username used to connect to the remote broker.

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 10.3, “Stopping a Remote Broker” shows how to stop a remote broker on a machine named **`NEBrokerHost2`**.

Example 10.3. Stopping a Remote Broker

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

Using a remote console

Red Hat JBoss A-MQ's console can be connected to a remote broker using the **`ssh:ssh`** command. Once the console is connected to the remote broker, you can shut it down by running the **`osgi:shutdown`** command. Example 10.4, “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 10.4. Shutting Down a Broker using a Remote Console Connection

```
JBossA-MQ:karaf@root> ssh -l karaf -P karaf NWBrokerHost
       _ ____ __ __ ____ | | _ \ /\ | \/ |/ __ \ | | |_) | ___ ___ ___ /
\ ______| \ / | | | | _ | | _ < / _ \/ __/ __| / /\ \______| |\/| | | |
| | |__| | |_) | (_) \__ \__ \ / ____ \ | | | | |__| | \____/|____/
\___/|___/___/ /_/ \_\ |_| |_|\___\_\ JBoss A-MQ (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 JBoss
A-MQ.

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 10.5, "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 10.5. Shutting Down a Broker in a Fabric

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

CHAPTER 11. CONNECTING A BROKER TO A FABRIC

Abstract

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

When Red Hat JBoss A-MQ is installed it is set up to run a standalone broker. If you want the broker to become part of a fabric you have two options:

- join an existing fabric

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

  NOTE

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

- create a new fabric

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

11.1. JOINING A BROKER TO A FABRIC

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

- A *managed container* is a full member of the fabric and is managed by a Fabric Agent. The agent configures the container based on information provided by the fabric's ensemble. The ensemble knows which profiles are associated with the container and the agent determines what to install based on the contents of the profiles.

- A *non-managed container* is not managed by a Fabric Agent. It's configuration remains intact after it joins the fabric and is controlled as if the broker were a standalone broker. Joining the fabric in this manner registers the broker with the fabric's ensemble and allows clients to locate the broker using the fabric's discovery mechanism.

11.1.1. Joining a Fabric as a Managed Container

Overview

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

Becoming a managed container

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

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

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

   WARNING

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

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

   This process clears the container and resets its configuration to a default state.
5. The ensemble installs the **`fabric`** profile into the container.

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

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

Remaining a message broker

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

The default set of profiles installed with Fuse Fabric include an **`mq`** profile that loads a default message broker. You can also create a new profile for your brokers as described in Section 2.4, “Editing a Broker's Configuration in a Fabric”.

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

Procedure

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

1. Create a profile for your broker.

   For details on creating a broker profile see Section 2.4, “Editing a Broker's Configuration in a Fabric”.

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

   The URL of a Server has the following format:

   ```
   HostName:IPPort
   ```

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

   1. Connect to the command console of one of the containers in the fabric.

   2. Enter the following sequence of console commands:

      ```
      JBossA-MQ:karaf@root> config:edit io.fabric8.zookeeper
      JBossA-MQ:karaf@root> config:proplist
       service.pid = io.fabric8.zookeeper zookeeper.url =
      myhostA:2181,myhostB:2181,myhostC:2181,myhostC:2182,myhostC:2183
      fabric.zookeeper.pid = io.fabric8.zookeeper
      JBossA-MQ:karaf@root> config:cancel
      ```

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

3. Connect to the standalone broker's command console.

4. Enter the following command:

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

   You need to provide values for:

   - *brokerProfile*—the name of the profile you created in Step 1
   - *fabricURL*—the Fabric Server URL you obtained in Step 2
   - *brokerName*—the name the broker will use to register with the fabric

IMPORTANT

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

11.1.2. Joining a Fabric as a Non-Managed Container

Overview

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

Becoming a non-managed container

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

1. The broker installs the required Fuse Fabric bundles to interact with the fabric's ensemble.

2. The broker contacts the specified Fabric Server and initiates the joining process.

3. The Fabric Server registers the container with the fabric's ensemble.

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

   WARNING

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

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

Procedure

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

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

   The URL of a Server has the following format:

   ```
   HostName:IPPort
   ```

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

URLs of the Fabric Servers as follows:

1. Connect to the command console of one of the containers in the fabric.

2. Enter the following sequence of console commands:

   ```
   JBossA-MQ:karaf@root> config:edit io.fabric8.zookeeper
   JBossA-MQ:karaf@root> config:proplist
    service.pid = io.fabric8.zookeeper zookeeper.url =
   myhostA:2181,myhostB:2181,myhostC:2181,myhostC:2182,myhostC:2183
   fabric.zookeeper.pid = io.fabric8.zookeeper
   JBossA-MQ:karaf@root> config:cancel
   ```

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

2. Connect to the standalone broker's command console.

3. Enter the following command:

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

   You need to provide values for:

   - **-n**—specifies that the broker will **not** be managed by the fabric
   - *fabricURL*—the Fabric Server URL you obtained in Step 1
   - *brokerName*—the name the broker will use to register with the fabric

   IMPORTANT

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

11.2. CREATING A NEW FABRIC

Overview

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

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

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

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

Becoming a new fabric

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

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

   NOTE

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

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

   This starts a default broker instance in the container.

   NOTE

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

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

Procedure

To create a new fabric from a standalone broker:

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

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

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

  b. Use **`config:delete`** to delete all of the PIDs using the form **`org.fusesource.mq.fabric.server.`*`ID`***.

4. Enter the following command:

```
JBossA-MQ:karaf@root> fabric:create
```

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

5. Add a new broker to the fabric as described in Section 9.2, “Deploying a New Broker into a Fabric”.

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 JBoss A-MQ supports a number of transport flavors to facilitate interoperability with the widest possible array of clients.

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 `org.fusesource.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*.

12.1. 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 **`org.fusesource.mq.fabric.server.`*`id`*** PID for editing using the **`config:edit`** command.

    ```
    JBossAMQ:karaf> config:edit org.fusesource.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
    ```

12.2. ADDING A TRANSPORT CONNECTOR TO A FABRIC BROKER

NOTE

The management console makes configuring fabric brokers easier. For more information see *Using the Management Console*

Adding a transport connector definition

To add a transport connector definition:

1. Create a configuration template.

    See Section 2.1, “Understanding the Red Hat JBoss A-MQ Configuration Model” .

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.

7. In the command console, use the **`fabric:import`** command to upload the your XML configuration template to the Fabric Ensemble.

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

   version must match the version of the new profile.

8. Use the **`fabric:mq-create`** command to create a new profile.

   ```
   JBossAMQ:karaf> fabric:mq-create --config configFile profileName
   ```

   This will create a new broker profile that inherits from the **`mq-base`** profile, but uses your XML configuration template.

NOTE

The newly added transport connector is not available until the profile containing it is modified to activate the connector.

Activating a connector

To activate a transport connector in a fabric broker:

1. Connect to the broker using a command console.

2. Verify the value of the connectors property for the desired profile using the **`fabric:profile-display`** command.

   ```
   JBossAMQ:karaf> fabric:profile-display profileName
   ```

3. Change the value of the connectors property using the **`fabric:profile-edit`** command's **`-p`** option.

   ```
   JBossAMQ:karaf> fabric:profile-edit -p
   org.fusesource.mq.fabric.server-profileName/connectors="connector1
   connector2..." profileName
   ```

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

4. Deploy the new profile to one or more brokers in the fabric to test the changes.

```
JBossAMQ:karaf> fabric:container-add-profile broker profileName
```

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 *JBoss A-MQ 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.

Red Hat JBoss A-MQ 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

14.1. 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 JBoss A-MQ 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 JBoss A-MQ 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
```

14.2. VIEWING THE LOG

Abstract

You can view the log using your systems text display mechanisms, the Red Hat JBoss A-MQ 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 JBoss A-MQ 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.

CHAPTER 15. USING JMX

Abstract

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

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

15.1. 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 JBoss A-MQ 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.

[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>
```

15.2. 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.
TotalEnqueueCount	Specifies the total number of messages that have been sent to the broker.

Name	Description
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.
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.

Name	Description
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.
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.

Name	Description
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.

15.3. 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 JBoss A-MQ 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.
`void addQueue(String name);`	Adds a queue destination to the broker.
`void removeTopic(String name);`	Removes the specified topic destination from the broker.

Operation	Description
`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.

Operation	Description
`boolean copyMessageTo(String messageId, String destination);`	Copies the specified message to a new destination.
`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.

Operation	Description
`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.
`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.

Operation	Description
`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 JBoss A-MQ supports incremental patching. Red Hat will supply you with easy to install patches that only make targeted changes to a deployed broker.

Incremental patching allows you apply targets fixes to a broker without needing to reinstall an updated version of Red Hat JBoss A-MQ. It also allows you to easily back the patch out if it causes problems with your deployed applications.

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

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

- Customer Support sends you a patch.
- Customer Support sends you a link to download a patch.
- 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. The management console is the recommended way to patch brokers in a fabric. See *Using the Management Console*for more information.

16.1. FINDING THE RIGHT PATCHES TO APPLY

Abstract

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

Locate the patches on the customer portal

If you have a subscription for JBoss A-MQ, 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 JBoss A-MQ.
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, 6.1.0, from the **`Version`** dropdown menu. A table of downloads now appears, which has three tabs: **`Releases`**, **`Patches`**, and **`Security Advisories`**.

NOTE

Make sure you select the right GA version for your product. A micro version release (for example, 6.1.1) is not the same thing as a patched release.

4. Click the `Patches` tab to view the regular patches (with no security-related fixes).

5. Click the `Security Advisories` tab to view the patches with security-related fixes.

TIP

To see the *complete* set of patches, you must look under both the `Patches` tab *and* the `Security Avisories` tab.

Types of patch

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

- Patches with GA baseline (for example, Patch 1, Patch 2, Patch 3, and so on)
- Rollup patches (for example, Rollup 1, Rollup 2, and so on)
- Patches with rollup baseline (for example, Rollup 1 Patch1, Rollup1 Patch2, and so on)

Patches with GA baseline

Patches with GA baseline (Patch1, Patch2, and so on) are released shortly after the GA date to provide quick fixes for issues identified after GA. These patches can be applied directly to the GA product. These patches are *cumulative*: that is, Patch 2 would contain all of the fixes from Patch 1; and Patch 3 would contain all of the fixes from Patch 1 and Patch 2; and so on.

Rollup patches

A rollup patch (Rollup 1, Rollup 2, and so on) 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.

Patches with rollup baseline

Patches with rollup baseline (Rollup 1 Patch 1, Rollup 1 Patch2, and so on) are patches released *after* a rollup patch, and they are intended to be applied on top of the corresponding rollup patch. For example, Rollup 1 Patch 2 would be applied on top of the Rollup 1 patch; and Rollup 2 Patch 1 would be applied on top of the Rollup 2 patch.

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.

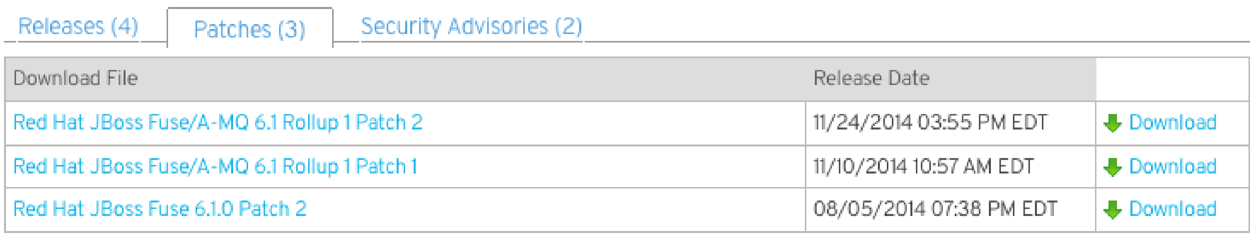
Example of identifying patches to apply

To give a concrete example of how to identify which patches to apply, we take a snapshot of the patches that were available in December 2014 and we discuss which patches you need to apply to get to the latest patch level.

Patches available under the Patches tab

The patches available under the **`Patches`** tab are shown in Figure 16.1, “Patches Tab”.

Figure 16.1. Patches Tab

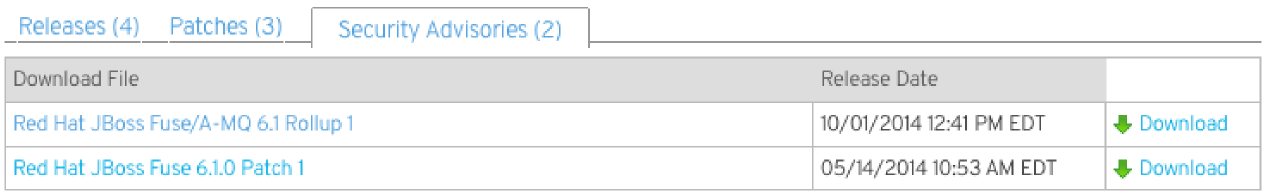


Download File	Release Date	
Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1 Patch 2	11/24/2014 03:55 PM EDT	Download
Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1 Patch 1	11/10/2014 10:57 AM EDT	Download
Red Hat JBoss Fuse 6.1.0 Patch 2	08/05/2014 07:38 PM EDT	Download

Patches available under the Security Advisories tab

The patches available under the **`Security Advisories`** tab are shown in Figure 16.2, “Security Advisories Tab”.

Figure 16.2. Security Advisories Tab



Download File	Release Date	
Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1	10/01/2014 12:41 PM EDT	Download
Red Hat JBoss Fuse 6.1.0 Patch 1	05/14/2014 10:53 AM EDT	Download

Complete list of available patches

Taking all of the patches from the **`Patches`** tab and the **`Security Advisories`** tab together, we come up with the following list of downloadable patches, in the order they were released:

- Red Hat JBoss Fuse 6.1.0 Patch 1
- Red Hat JBoss Fuse 6.1.0 Patch 2
- Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1

- Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1 Patch 1
- Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1 Patch 2

Patches you would apply to update to the latest patch level

In this case, to update the GA product to the very latest patch level, you would apply the following sequence of patches:

1. Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1
2. Red Hat JBoss Fuse/A-MQ 6.1 Rollup 1 Patch 2

16.2. 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

Patching a standalone container directs the container to load the patch versions of artifacts instead of the non-patch versions. The **patch** shell provides commands to patch the container's environment, see which bundles are effected by applying the patch, apply the patch to the container, and back the patch out if needed (see chapter "Patch Console Commands" in "Console Reference").

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

IMPORTANT

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

Applying a patch

To apply a patch to a standalone container:

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

   Example 16.1, "Adding a Patch to a Broker's Environment" shows the command for adding the patch contained in the patch file **patch.zip** from the local file system.

   Example 16.1. Adding a Patch to a Broker's Environment

   ```
   JBoss A-MQ> patch:add file://patch.zip
   ```

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

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

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

   NOTE

   The **patch:list** command displays a list of all patches added to the container's **system** folder.

3. Review the simulation log to understand the changes that will be made to the container.

4. Apply the patch to the container using the **patch:install** command.

   WARNING

   Running **patch:install** before the container is fully started and all of the bundles are active will cause the container to hang.

   NOTE

   The **patch:list** command displays a list of all patches added to the container's **system** folder.

5. Shut down the container that you just applied the patch to.

6. The extracted patch archive contains the **manual_steps** directory. Copy the content of the **manual_steps/xyz** directory to the appropriate directory (**bin**, **etc**, **lib**) in the JBoss A-MQ 6.1 installation directory. Copy the content in the **manual_steps/fabric-system-updates/system** directory to the **system** directory in the JBoss A-MQ installation directory. This is the system repository that contains some patched artifacts.

7. Start the container. If you are using a remote console, you will lose the connection to the container. If you are using the container's local console, it will automatically reconnect when the container restarts.

Rolling back a patch

Occasionally a patch will not work or introduce new issues to a container. In these cases you can easily back the patch out of the system and restore it pre-patch behavior using the **patch:rollback** command. As shown in Example 16.2, “Rolling Back a Patch” , the command takes the name of patch to be backed out.

Example 16.2. Rolling Back a Patch

```
JBoss A-MQ> patch:rollback patch1
```

NOTE

The `patch:list` command displays a list of all patches added to the container's `system` folder.

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

Adding features to a patched container

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

16.3. PATCHING STANDALONE APACHE ACTIVEMQ

Abstract

JBoss A-MQ 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 16.1, “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-5.9.0.redhat-610379.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 16.1, “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-5.9.0.redhat-610379/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.

16.4. PATCHING A CONTAINER IN A FABRIC

Abstract

In a fabric patches are applied to profiles and the patched version of the profile is applied to the container. The management console is the recommended tool for patching containers in a fabric. The **`fabric`** shell also has the commands needed to apply a patch and roll it out to running containers.

Overview

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

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

The management console is the recommended tool for patching containers in a fabric. However, the command console's **`fabric`** shell also provides the commands needed to patch containers running in a fabric.

Procedure

Patching a container in a fabric involves:

1. Getting a patch file.
   - Customer Support sends you a patch.
   - Customer Support sends you a link to download a patch.
   - You, or your organization, generate a patch file for an internally created application.
2. Uploading one or more patch files to the fabric's Maven repository.
3. Applying the patch(es) to a profile version.

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

Using the management console

The management console is the easiest and most verbose method of patching containers in a fabric. Its **`Patching`** tab uploads patches to a fabric's Maven repository and applies the patch to a specified profile version. You can then use the management console to roll the patch out to all of the containers in the fabric.

See chapter "Patching a Fabric" in "Management Console User Guide" for more information.

Using the command console

The Red Hat JBoss Fuse command console can also be used to patch containers running in a fabric. To patch a fabric container:

1. 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

   The version name must be a pure *numeric* string, such as **`1.1`**, **`1.2`**, **`2.1`**, or **`2.2`**. You cannot incorporate alphabetic characters in the version name (such as **`1.0.patch`**).

2. 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
   ```

3. Upgrade the container using the **`fabric:container-upgrade`** command, specifying which container you want to upgrade. For example, to upgrade the **`root`** container, enter the following command:

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

4. 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.

TIP

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)
```

APPENDIX A. REQUIRED JARS

OVERVIEW

To simplify deploying Red Hat JBoss A-MQ 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 JBoss A-MQ installation.

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

REQUIRED JARS FROM RED HAT JBOSS A-MQ

The following JARs are installed with JBoss A-MQ 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