Red Hat

Red Hat JBoss Enterprise Application Platform 7.4

Configuring Messaging

Instructions and information for developers and administrators who want to develop and deploy messaging applications for Red Hat JBoss Enterprise Application Platform.

Last Updated: 2024-01-17

Red Hat JBoss Enterprise Application Platform 7.4 Configuring Messaging

Instructions and information for developers and administrators who want to develop and deploy messaging applications for Red Hat JBoss Enterprise Application Platform.

Legal Notice

Abstract

This document provides information for developers and administrators who want to develop and deploy messaging applications with Red Hat JBoss Enterprise Application Platform.

Table of Contents

PROVIDING FEEDBACK ON JBOSS EAP DOCUMENTATION

To report an error or to improve our documentation, log in to your Red Hat Jira account and submit an issue. If you do not have a Red Hat Jira account, then you will be prompted to create an account.

Procedure

1. Click the following link to **create a ticket**.
2. Please include the **Document URL**, the **section number** and **describe the issue**.
3. Enter a brief description of the issue in the **Summary**.
4. Provide a detailed description of the issue or enhancement in the **Description**. Include a URL to where the issue occurs in the documentation.
5. Clicking **Submit** creates and routes the issue to the appropriate documentation team.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

PART I. ABOUT MESSAGING AND JBOSS EAP 7

The messaging broker in JBoss EAP 6 was called HornetQ, a JBoss community project. The HornetQ codebase was donated to the Apache ActiveMQ project, and the HornetQ community joined that project to enhance the donated codebase and create a next-generation messaging broker. The result is Apache ActiveMQ Artemis, the messaging broker for JBoss EAP 7, providing messaging consolidation and backwards compatibility with JBoss EAP 6. While ActiveMQ Artemis retains protocol compatibility with the HornetQ broker in JBoss EAP 6, it also contains some smart new features. This guide will explore, and provide useful examples for, the many features of the ActiveMQ Artemis broker available in JBoss EAP 7.4.

CHAPTER 1. MESSAGING CONCEPTS

1.1. MESSAGING SYSTEMS

Messaging systems allow you to loosely couple heterogeneous systems together with added reliability. Unlike systems based on a Remote Procedure Call (RPC) pattern, messaging systems primarily use an asynchronous message passing pattern with no tight relationship between requests and responses. Most messaging systems are flexible enough to also support a request-response mode if needed, but this is not a primary feature of messaging systems.

Messaging systems decouple a message's sender of messages from its consumers. In fact, the senders and consumers of messages are completely independent and know nothing of each other, which allows you to create flexible, loosely coupled systems. Large enterprises often use a messaging system to implement a message bus which loosely couples heterogeneous systems together. Message buses can form the core of an Enterprise Service Bus (ESB). Using a message bus to decouple disparate systems allows the system to grow and adapt more easily. It also allows more flexibility to add new systems or retire old ones since they do not have brittle dependencies on each other.

Messaging systems can also incorporate concepts such as delivery guarantees to ensure reliable messaging, transactions to aggregate the sending or consuming of multiple message as a single unit of work, and durability to allow messages to survive server failure or restart.

1.2. MESSAGING STYLES

There are two kinds of messaging styles that most messaging systems support: the *point-to-point* pattern and the *publish-subscribe* pattern.

- Point-to-Point Pattern
  The point-to-point pattern involves sending a message to a single consumer listening on a queue. Once in the queue, the message is usually made persistent to guarantee delivery. Once the message has moved through the queue, the messaging system delivers it to a consumer. The consumer acknowledges the delivery of the message once it is processed. There can be multiple consumers listening on the same queue for the same message, but only one consumer will receive each message.

- Publish-Subscribe Pattern
  The publish-subscribe pattern allow senders to send messages to multiple consumers using a single destination. This destination is often known as a *topic*. Each topic can have multiple consumers, or subscribers, and unlike point-to-point messaging, every subscriber receives any message published to the topic.

  Another interesting distinction is that subscribers can be durable. Durable subscriptions pass the server a unique identifier when connecting, which allows the server to identify and send any messages published to the topic since the last time the subscriber made a connection. Such messages are typically retained by the server even after a restart.

1.3. JAKARTA MESSAGING

Jakarta Messaging 2.0 is defined in Jakarta Messaging. Jakarta Messaging is a Java API that provides both point-to-point and publish-subscriber messaging styles. Jakarta Messaging also incorporates the use of transactions. Jakarta Messaging does not define a standard wire format so while vendors of Jakarta Messaging providers may all use the standard APIs, they may use different internal wire protocols to communicate between their clients and servers.

1.4. JAKARTA MESSAGING DESTINATIONS

Jakarta Messaging destinations, along with Jakarta Messaging connection factories, are administrative objects. Destinations are used by Jakarta Messaging clients for both producing and consuming messages. The destination allows clients to specify the target when it produces messages and the source of messages when consuming messages. When using a publish-subscribe pattern, destinations are referred to as topics. When using a point-to-point pattern, destinations are referred to as queues.

Applications may use many different Jakarta Messaging destinations which are configured on the server side and usually accessed via JNDI.

CHAPTER 2. THE INTEGRATED ACTIVEMQ ARTEMIS MESSAGING BROKER

2.1. ACTIVEMQ ARTEMIS

Apache ActiveMQ Artemis is an open source project for an asynchronous messaging system. It is high performance, embeddable, clustered and supports multiple protocols. JBoss EAP 7 uses Apache ActiveMQ Artemis as its Jakarta Messaging broker and is configured using the **messaging-activemq** subsystem. This fully replaces the HornetQ broker but retains protocol compatibility with JBoss EAP 6.

The core ActiveMQ Artemis is Jakarta Messaging-agnostic and provides a non-Jakarta Messaging API, which is referred to as the *core API*. ActiveMQ Artemis also provides a Jakarta Messaging client API which uses a facade layer to implement the Jakarta Messaging semantics on top of the core API. Essentially, Jakarta Messaging interactions are translated into core API operations on the client side using the Jakarta Messaging client API. From there, all operations are sent using the core client API and Apache ActiveMQ Artemis wire format. The server itself only uses the core API. For more details on the core API and its concepts, refer to the ActiveMQ Artemis documentation.

2.2. APACHE ACTIVEMQ ARTEMIS CORE API AND JAKARTA MESSAGING DESTINATIONS

Let’s quickly discuss how Jakarta Messaging destinations are mapped to Apache ActiveMQ Artemis addresses.

Apache ActiveMQ Artemis core is Jakarta Messaging-agnostic. It does not have any concept of a Jakarta Messaging topic. A Jakarta Messaging topic is implemented in core as an address (the topic name) with zero or more queues bound to it. Each queue bound to that address represents a topic subscription. Likewise, a Jakarta Messaging queue is implemented as an address (the Jakarta Messaging queue name) with one single queue bound to it which represents the Jakarta Messaging queue.

By convention, all Jakarta Messaging queues map to core queues where the core queue name has the string **jms.queue.** prepended to it. For example, the Jakarta Messaging queue with the name **orders.europe** would map to the core queue with the name **jms.queue.orders.europe**. The address at which the core queue is bound is also given by the core queue name.

For Jakarta Messaging topics the address at which the queues that represent the subscriptions are bound is given by prepending the string **jms.topic.** to the name of the Jakarta Messaging topic. For example, the Jakarta Messaging topic with name **news.europe** would map to the core address **jms.topic.news.europe**.

In other words if you send a **Jakarta Messaging** message to a Jakarta Messaging queue with name **orders.europe**, it will get routed on the server to any core queues bound to the address **jms.queue.orders.europe**. If you send a **Jakarta Messaging** message to a Jakarta Messaging topic with name **news.europe**, it will get routed on the server to any core queues bound to the address **jms.topic.news.europe**.

If you want to configure settings for a Jakarta Messaging queue with the name **orders.europe**, you need to configure the corresponding core queue **jms.queue.orders.europe**:

```
<!-- expired messages in JMS Queue "orders.europe" will be sent to the JMS Queue "expiry.europe" -
->
<address-setting match="jms.queue.orders.europe">
```

```
  <expiry-address>jms.queue.expiry.europe</expiry-address>
  ...
</address-setting>
```

PART II. CONFIGURING SINGLE-NODE MESSAGING SYSTEMS

Part II begins with a guide to getting started with JBoss EAP 7 messaging by using the **helloworld-mdb** quickstart. Configuration options available to any installation follow, including topics such as security and persistence. For configuration relating to multiple installations of JBoss EAP 7, including topics such as clustering, high availability, and connecting to another server, see Part III, Configuring Multi-Node Messaging Systems.

CHAPTER 3. GETTING STARTED

3.1. USING THE HELLOWORLD-MDB QUICKSTART

The **helloworld-mdb** quickstart uses a simple message-driven bean to demonstrate basic Jakarta EE messaging features. Having the quickstart up and running as you review the basic configuration is an excellent way to introduce yourself to the features included with the JBoss EAP messaging server.

Build and Deploy the helloworld-mdb Quickstart

See the instructions in the **README.md** file provided with the quickstart for instructions on building and deploying the **helloworld-mdb** quickstart. You will need to start the JBoss EAP server specifying the **full** configuration, which contains the **messaging-activemq** subsystem. See the **README.md** file or the JBoss EAP *Configuration Guide* for details on starting JBoss EAP with a different configuration file.

3.2. OVERVIEW OF THE MESSAGING SUBSYSTEM CONFIGURATION

Default configuration for the **messaging-activemq** subsystem is included when starting the JBoss EAP server with the **full** or **full-ha** configuration. The **full-ha** option includes advanced configuration for features like clustering and high availability.

Although not necessary, it is recommended that you use the **helloworld-mdb** quickstart as a working example to have running alongside this overview of the configuration.

For information on all settings available in the **messaging-activemq** subsystem, see the schema definitions located in the ***EAP_HOME*/docs/schema/** directory, or run the **read-resource-description** operation on the subsystem from the management CLI, as shown below.

```
/subsystem=messaging-activemq:read-resource-description(recursive=true)
```

The following extension in the server configuration file tells JBoss EAP to include the **messaging-activemq** subsystem as part of its runtime.

```
<extensions>
  ...
  <extension module="org.wildfly.extension.messaging-activemq"/>
  ...
</extensions>
```

The configuration for the **messaging-activemq** subsystem is contained within the **<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">** element.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
    <server name="default">
        <cluster password="${jboss.messaging.cluster.password:CHANGE ME!!}"/>
        <security-setting name="#">
            <role name="guest" send="true" consume="true" create-non-durable-queue="true" delete-non-durable-queue="true"/>
        </security-setting>
        <address-setting name="#" dead-letter-address="jms.queue.DLQ" expiry-address="jms.queue.ExpiryQueue" max-size-bytes="10485760" page-size-bytes="2097152" message-counter-history-day-limit="10" redistribution-delay="1000"/>
        <http-connector name="http-connector" socket-binding="http" endpoint="http-acceptor"/>
        <http-connector name="http-connector-throughput" socket-binding="http" endpoint="http-
```

```
acceptor-throughput">
            <param name="batch-delay" value="50"/>
        </http-connector>
        <in-vm-connector name="in-vm" server-id="0"/>
        <http-acceptor name="http-acceptor" http-listener="default"/>
        <http-acceptor name="http-acceptor-throughput" http-listener="default">
            <param name="batch-delay" value="50"/>
            <param name="direct-deliver" value="false"/>
        </http-acceptor>
        <in-vm-acceptor name="in-vm" server-id="0"/>
        <broadcast-group name="bg-group1" connectors="http-connector" jgroups-cluster="activemq-
cluster"/>
        <discovery-group name="dg-group1" jgroups-cluster="activemq-cluster"/>
        <cluster-connection name="my-cluster" address="jms" connector-name="http-connector"
discovery-group="dg-group1"/>
        <jms-queue name="ExpiryQueue" entries="java:/jms/queue/ExpiryQueue"/>
        <jms-queue name="DLQ" entries="java:/jms/queue/DLQ"/>
        <connection-factory name="InVmConnectionFactory" connectors="in-vm"
entries="java:/ConnectionFactory"/>
        <connection-factory name="RemoteConnectionFactory" ha="true" block-on-acknowledge="true"
reconnect-attempts="-1" connectors="http-connector"
entries="java:jboss/exported/jms/RemoteConnectionFactory"/>
        <pooled-connection-factory name="activemq-ra" transaction="xa" connectors="in-vm"
entries="java:/JmsXA java:jboss/DefaultJMSConnectionFactory"/>
    </server>
</subsystem>
```

Connection Factories
Messaging clients use a Jakarta Messaging **ConnectionFactory** object to make connections to the server. The default JBoss EAP configuration defines several connection factories. Note that there is a **<connection-factory>** for in-vm, http, and pooled connections.

```
  <connection-factory name="InVmConnectionFactory" connectors="in-vm"
entries="java:/ConnectionFactory"/>
  <connection-factory name="RemoteConnectionFactory" ha="true" block-on-acknowledge="true"
reconnect-attempts="-1" connectors="http-connector"
entries="java:jboss/exported/jms/RemoteConnectionFactory"/>
  <pooled-connection-factory name="activemq-ra" transaction="xa" connectors="in-vm"
entries="java:/JmsXA java:jboss/DefaultJMSConnectionFactory"/>
```

See the Configuring Connection Factories section for more details.

Connectors and Acceptors
Each Jakarta Messaging connection factory uses connectors to enable Jakarta Messaging-enabled communication from a client producer or consumer to a messaging server. The connector object defines the transport and parameters used to connect to the messaging server. Its counterpart is the acceptor object, which identifies the type of connections accepted by the messaging server.

The default JBoss EAP configuration defines several connectors and acceptors.

Example: Default Connectors

```
<http-connector name="http-connector" socket-binding="http" endpoint="http-acceptor"/>
<http-connector name="http-connector-throughput" socket-binding="http" endpoint="http-acceptor-
throughput">
```

```
  <param name="batch-delay" value="50"/>
</http-connector>
<in-vm-connector name="in-vm" server-id="0"/>
```

Example: Default Acceptors

```
<http-acceptor name="http-acceptor" http-listener="default"/>
<http-acceptor name="http-acceptor-throughput" http-listener="default">
  <param name="batch-delay" value="50"/>
  <param name="direct-deliver" value="false"/>
</http-acceptor>
```

See the Acceptors and Connectors section for more details.

Socket Binding Groups

The **socket-binding** attribute for the default connectors reference a socket binding named **http**. The http connector is used because JBoss EAP can multiplex inbound requests over standard web ports.

You can find this **socket-binding** as part of the **<socket-binding-group>** section elsewhere in the configuration file. Note how the configuration for the http and https socket bindings appear within the **<socket-binding-groups>** element:

```
<socket-binding-group name="standard-sockets" default-interface="public" port-
offset="${jboss.socket.binding.port-offset:0}">
   ...
  <socket-binding name="http" port="${jboss.http.port:8080}"/>
  <socket-binding name="https" port="${jboss.https.port:8443}"/>
   ...
</socket-binding-group>
```

For information on socket bindings, see Configuring Socket Bindings in the JBoss EAP *Configuration Guide*.

Messaging Security

The **messaging-activemq** subsystem includes a single **security-setting** element when JBoss EAP is first installed:

```
<security-setting name="#">
  <role name="guest" delete-non-durable-queue="true" create-non-durable-queue="true"
consume="true" send="true"/>
</security-setting>
```

This declares that any user with the role **guest** can access any address on the server, as noted by the wildcard **#**. See Configuring Address Settings for more information on the wildcard syntax .

For more information on securing destinations and remote connections see Configuring Messaging Security.

Messaging Destinations

The **full** and **full-ha** configurations include two helpful queues that JBoss EAP can use to hold messages that have expired or that cannot be routed to their proper destination.

```
<jms-queue name="ExpiryQueue" entries="java:/jms/queue/ExpiryQueue"/>
<jms-queue name="DLQ" entries="java:/jms/queue/DLQ"/>
```

You can add your own messaging destinations in JBoss EAP using one of the following methods.

- Using the management CLI
  Use the following management CLI command to add a queue.

  ```
  jms-queue add --queue-address=testQueue  --
  entries=queue/test,java:jboss/exported/jms/queue/test
  ```

  Use the following management CLI command to add a topic.

  ```
  jms-topic add --topic-address=testTopic  --
  entries=topic/test,java:jboss/exported/jms/topic/test
  ```

- Using the management console
  Messaging destinations can be configured from the management console by navigating to **Configuration** → **Subsystems** → **Messaging (ActiveMQ)** → **Server**, selecting the server, selecting **Destinations**, and clicking **View**. Select the **JMS Queue** tab to configure queues and select the **JMS Topic** to configure topics.

- Defining your destinations using a Jakarta EE deployment descriptor or annotation.
  In Jakarta EE 8, deployment descriptors can include configuration for queues and topics. Below is a snippet from a Jakarta EE descriptor file that defines a Jakarta Messaging queue.

  ```
  ...
  <jms-destination>
    <name>java:global/jms/MyQueue</name>
    <interfaceName>javax.jms.Queue</interfaceName>
    <destinationName>myQueue</destinationName>
  </jms-destination>
  ...
  ```

  For example, the message-driven beans in the **helloworld-mdb** quickstart contain annotations that define the queue and topic needed to run the application. Destinations created in this way will appear in the list of runtime queues. Use the management CLI to display the list of runtime queues. After deploying the quickstart the runtime queues it created will appear as below:

  ```
  /subsystem=messaging-activemq/server=default/runtime-queue=*:read-resource
  {
      "outcome" => "success",
      "result" => [
          ...
          {
              "address" => [
                  ("subsystem" => "messaging-activemq"),
                  ("server" => "default"),
                  ("runtime-queue" => "jms.queue.HelloWorldMDBQueue")
              ],
              "outcome" => "success",
              "result" => {"durable" => undefined}
          },
          ...
          {
              "address" => [
                  ("subsystem" => "messaging-activemq"),
  ```

```
                ("server" => "default"),
                ("runtime-queue" => "jms.topic.HelloWorldMDBTopic")
            ],
            "outcome" => "success",
            "result" => {"durable" => undefined}
        },
        ...
    ]
}
```

See Configuring Messaging Destinations for more detailed information.

CHAPTER 4. CONFIGURING MESSAGING DESTINATIONS

NOTE

Remember, configuring messaging destinations requires JBoss EAP to have messaging enabled. This functionality is enabled by default when running with the **standalone-full.xml** or **standalone-full-ha.xml** configuration files. The **domain.xml** configuration file also has messaging enabled.

4.1. ADDING A QUEUE

To add a Jakarta Messaging queue, use the **jms-queue** command from the management CLI:

```
jms-queue add --queue-address=myQueue --entries=[queue/myQueue jms/queue/myQueue
java:jboss/exported/jms/queue/myQueue]
```

Note how the **entries** attribute is a list containing multiple JNDI names separated by a single space. Also note the use of square brackets, **[]**, to enclose the list of JNDI names. The **queue-address** provides routing configuration, and **entries** provides a list of JNDI names that clients can use to look up the queue.

Reading a Queue’s Attributes

You can read a queue’s configuration using the **jms-queue** command in the management CLI.

```
jms-queue read-resource --queue-address=myQueue
```

Alternatively, you can read a queue’s configuration by accessing the **messaging-activemq** subsystem using the management CLI:

```
/subsystem=messaging-activemq/server=default/jms-queue=myQueue:read-resource()
{
    "outcome" => "success",
    "result" => {
        "durable" => true,
        "entries" => ["queue/myQueue jms/queue/myQueue java:jboss/exported/jms/queue/myQueue"],
        "legacy-entries" => undefined,
        "selector" => undefined
    }
}
```

Attributes of a jms-queue

The management CLI displays all the attributes of the **jms-queue** configuration element when given the following command:

```
/subsystem=messaging-activemq/server=default/jms-queue=*:read-resource-description()
```

The table below provides all the attributes of a **jms-queue**:

Attribute	Description

Attribute	Description
consumer-count	The number of consumers consuming messages from this queue. Available at runtime.
dead-letter-address	The address to send dead messages to. See Configuring Dead Letter Addresses for more information.
delivering-count	The number of messages that this queue is currently delivering to its consumers. Available at runtime.
durable	Whether the queue is durable or not. See Messaging Styles for more information on durable subscriptions.
entries	The list of JNDI names the queue will be bound to. Required.
expiry-address	The address that will receive expired messages. See Configuring Message Expiry for details.
legacy-entries	The JNDI names the queue will be bound to.
message-count	The number of messages currently in this queue. Available at runtime.
messages-added	The number of messages added to this queue since it was created. Available at runtime.
paused	Whether the queue is paused. Available at runtime.
queue-address	The queue address defines what address is used for routing messages. See Configuring Address Settings for details on address settings. Required.
scheduled-count	The number of scheduled messages in this queue. Available at runtime.
selector	The queue selector. For more information on selectors see Filter Expressions and Message Selectors.
temporary	Whether the queue is temporary. See Temporary Queues and Runtime Queues for more information.

4.2. ADDING A TOPIC

Adding or reading a topic is much like adding a queue:

```
jms-topic add --topic-address=myTopic --entries=[topic/myTopic jms/topic/myTopic
java:jboss/exported/jms/topic/myTopic]
```

Reading a Topic's Attributes

Reading topic attributes also has syntax similar to that used for a queue:

```
jms-topic read-resource --topic-address=myTopic

entries
  topic/myTopic jms/topic/myTopic java:jboss/exported/jms/topic/myTopic
legacy-entries=n/a
```

```
/subsystem=messaging-activemq/server=default/jms-topic=myTopic:read-resource
{
    "outcome" => "success",
    "result" => {
        "entries" => ["topic/myTopic jms/topic/myTopic java:jboss/exported/jms/topic/myTopic"],
        "legacy-entries" => undefined
    }
}
```

Attributes of a **jms-topic**

The management CLI displays all the attributes of the **jms-topic** configuration element when given the following command:

```
/subsystem=messaging-activemq/server=default/jms-topic=*:read-resource-description()
```

The table below lists the attributes of a **jms-topic**:

Attribute	Description
delivering-count	The number of messages that this queue is currently delivering to its consumers. Available at runtime.
durable-message-count	The number of messages for all durable subscribers for this topic. Available at runtime.
durable-subscription-count	The number of durable subscribers for this topic. Available at runtime.
entries	The JNDI names the topic will be bound to. Required.
legacy-entries	The legacy JNDI names the topic will be bound to.
message-count	The number of messages currently in this queue. Available at runtime.
messages-added	The number of messages added to this queue since it was created. Available at runtime.
non-durable-message-count	The number of messages for all non-durable subscribers for this topic. Available at runtime.
non-durable-subscription-count	The number of non-durable subscribers for this topic. Available at runtime.
subscription-count	The number of (durable and non-durable) subscribers for this topic. Available at runtime.

Attribute	Description
temporary	Whether the topic is temporary.
topic-address	The address the topic points to. Required.

4.3. JNDI ENTRIES AND CLIENTS

A queue or topic must be bound to the **java:jboss/exported** namespace for a remote client to be able to look it up. The client must use the text after **java:jboss/exported/** when doing the lookup. For example, a queue named **testQueue** has for its **entries** the list **jms/queue/test java:jboss/exported/jms/queue/test**. A remote client wanting to send messages to **testQueue** would look up the queue using the string **jms/queue/test**. A local client on the other hand could look it up using **java:jboss/exported/jms/queue/test**, **java:jms/queue/test**, or more simply **jms/queue/test**.

Management CLI Help

You can find more information about the **jms-queue** and **jms-topic** commands by using the **--help --commands** flags:

```
jms-queue --help --commands
```

```
jms-topic --help --commands
```

4.4. PAUSE METHOD FOR JAKARTA MESSAGING TOPICS USING THE MANAGEMENT API

You can pause a topic by pausing all its consumers. If the topic has any new subscriptions being registered while the topic is in pause are also paused.

The subscribers of the topic do not receive new messages from the paused topic. However, the paused topic can still receive messages sent to it. When you resume the topic, the queued messages are delivered to the subscribers.

You can use the **persist** parameter to store the state of the topic so that the topic stays paused even if you restart the broker.

Additional resources

- For information about pausing a topic, see Pausing a topic.
- For information about resuming a topic, see Resuming a topic.

4.5. PAUSING A TOPIC

You can pause a topic so that all the subscribers of the topics stop receiving new messages from a paused topic.

Procedure

- Pause the topic as shown in the following example:

```
/subsystem=messaging-activemq/server=default/jms-topic=topic:pause()
{
    "outcome" => "success",
    "result" => undefined
}
```

A paused topic is as shown in the following example:

```
/subsystem=messaging-activemq/server=default/jms-topic=topic:read-
attribute(name=paused)
{
    "outcome" => "success",
    "result" => true
}
```

Additional resources

- For information about the pause method for topics, see Pause method for Jakarta Messaging topics using the management API.
- For information about resuming a topic, see Resuming a topic.

4.6. RESUMING A TOPIC

You can resume a paused topic. When you resume the topic, the messages the topic received while it was paused are delivered to the subscribers.

Procedure

- Resume the topic as shown in the following example:

```
/subsystem=messaging-activemq/server=default/jms-topic=topic:resume()
{
    "outcome" => "success",
    "result" => undefined
}
```

A resumed topic is as shown in the following example:

```
/subsystem=messaging-activemq/server=default/jms-topic=topic:read-
attribute(name=paused)
{
    "outcome" => "success",
    "result" => false
}
```

Additional resources

- For information about the pause method for topics, see Pause method for Jakarta Messaging topics using the management API.
- For information about pausing a topic, see Pausing a topic.

CHAPTER 5. CONFIGURING LOGGING

You can configure logging for the **messaging-activemq** subsystem by adding a log category in the JBoss EAP **logging** subsystem for **org.apache.activemq** and setting the desired log level. You can also configure a log handler for the category to configure how the log messages are recorded.

To see more information in the logs regarding XA transactions, change the log level of the **com.arjuna** category to a more verbose setting such as **TRACE** or **DEBUG**.

For more information on logging, including configuration for categories and other options, see the section on logging in the JBoss EAP *Configuration Guide*.

Table 5.1. Logging Categories

If you want logs for...	Use this category...
XA transactions	com.arjuna
All messaging activity	org.apache.activemq
Messaging Journal calls only	org.apache.activemq.artemis.journal
Jakarta Messaging calls only	org.apache.activemq.artemis.jms
Messaging utils calls only	org.apache.activemq.artemis.utils
Messaging core server only	org.apache.activemq.artemis.core.server

Configuring a Client for Logging

Configure messaging clients by following these steps.

1. Download dependencies to the JBoss Jakarta Messaging client and log manager.
   If you are using Maven, add the following dependencies to your **pom.xml** file:

```
<dependencies>
  ...
  <dependency>
    <groupId>org.jboss.logmanager</groupId>
    <artifactId>jboss-logmanager</artifactId>
    <version>1.5.3.Final</version>
  </dependency>
  <dependency>
     <groupId>org.jboss.eap</groupId>
     <artifactId>wildfly-jms-client-bom</artifactId>
     <type>pom</type>
  </dependency>
  ...
</dependencies>
```

   For more information, see the section on using Maven with JBoss EAP in the JBoss EAP *Development Guide*.

2. Create a properties file for the logger. Name it **logging.properties** and save it to a known location. Below is an example properties file. See the section on logging in the JBoss EAP *Development Guide* for more information on configuring logging options on the client side.

   ```
   # Root logger option
   loggers=org.jboss.logging,org.apache.activemq.artemis.core.server,org.apache.activemq.artemis.utils,org.apache.activemq.artemis.journal,org.apache.activemq.artemis.jms,org.apache.activemq.artemis.ra

   # Root logger level
   logger.level=INFO
   # Apache ActiveMQ Artemis logger levels
   logger.org.apache.activemq.artemis.jms.level=INFO
   logger.org.apache.activemq.artemis.journal.level=INFO
   logger.org.apache.activemq.artemis.utils.level=INFO
   logger.org.apache.activemq.artemis.core.server.level=INFO

   # Root logger handlers
   logger.handlers=FILE

   # File handler configuration
   handler.FILE=org.jboss.logmanager.handlers.FileHandler
   handler.FILE.level=FINE
   handler.FILE.properties=autoFlush,fileName
   handler.FILE.autoFlush=true
   handler.FILE.fileName=activemq.log
   handler.FILE.formatter=PATTERN

   # Formatter pattern configuration
   formatter.PATTERN=org.jboss.logmanager.formatters.PatternFormatter
   formatter.PATTERN.properties=pattern
   formatter.PATTERN.pattern=%d{HH:mm:ss,SSS} %-5p [%c] %s%E%n
   ```

3. Start the client with the expected parameters. When starting your client code using the **java** command, add the following parameters:

   a. Add the JBoss client and logger JARs to the class path:

      ```
      -cp /PATH/TO/jboss-client.jar:/PATH/TO/jboss-logmanager.jar
      ```

   b. Enable the JBoss logging manager:

      ```
      -Djava.util.logging.manager=org.jboss.logmanager.LogManager
      ```

   c. Set the location of the logging properties file:

      ```
      -Dlogging.configuration=/PATH/TO/logging.properties
      ```

   The full command to start the client will look something like the following example:

```
$ java -Djava.util.logging.manager=org.jboss.logmanager.LogManager -
Dlogging.configuration=/PATH/TO/logging.properties -cp /PATH/TO/jboss-client.jar:/PATH/TO/jboss-
logmanager.jar org.example.MyClient
```

CHAPTER 6. ADDRESS SETTINGS

The **messaging-activemq** subsystem has several configurable options which control aspects of how and when a message is delivered, how many attempts should be made, and when the message expires. These configuration options all exist within the **<address-setting>** configuration element. You can configure JBoss EAP to apply a single **<address-setting>** to multiple destinations by using a wildcard syntax.

6.1. WILDCARD SYNTAX

Wildcards can be used to match similar addresses with a single statement, much like how many systems use the asterisk character, *, to match multiple files or strings with a single query. The following table lists the special characters that can be used to define an **<address-setting>**.

Table 6.1. Jakarta Messaging Wildcard Syntax

Character	Description
. (a single period)	Denotes the space between words in a wildcard expression.
# (a pound or hash symbol)	Matches any sequence of zero or more words.
* (an asterisk)	Matches a single word.

The examples in the table below illustrate how wildcards are used to match a set of addresses.

Table 6.2. Jakarta Messaging Wildcard Examples

Example	Description
news.europe.#	Matches **news.europe**, **news.europe.sport**, **news.europe.politics.fr**, but not **news.usa** or **europe**.
news.*	Matches **news.europe** and **news.usa**, but not **news.europe.sport**.
news.*.sport	Matches **news.europe.sport** and **news.usa.sport**, but not **news.europe.fr.sport**.

6.2. DEFAULT ADDRESS-SETTING

Out of the box, JBoss EAP includes a single **address-setting** element as part of the configuration for the **messaging-activemq** subsystem:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <address-setting
      name="#"
      dead-letter-address="jms.queue.DLQ"
      expiry-address="jms.queue.ExpiryQueue"
```

```
        max-size-bytes="10485760"
        page-size-bytes="2097152"
        message-counter-history-day-limit="10" />
      ...
  </server>
</subsystem>
```

NOTE

The use of a single **#** for the **name** attribute makes this default **address-setting** the configuration to be used for all destinations since **#** matches any address. You can continue to apply this catch-all configuration to all of your addresses, or you can add a new **address-setting** for each address or group of addresses that requires its own configuration set.

Configuring Address Settings Using the Management CLI

Configuring address settings is done by using either the management CLI or the management console, but the management CLI exposes more of the configuration attributes for editing. See Address Setting Attributes in the appendix of this guide for the full list of attributes.

Add a new address-setting

Use the **add** operation to create a new address setting if required. You can run this command from the root of the management CLI session, which in the following examples creates a new pattern named . You can include configuration attributes for the **address-setting**. Below, a new **address-setting** matching **news.europe.#** is created with its **dead-letter-address** attribute set to the queue **DLQ.news**, which was created beforehand. Examples for both a standalone server and a managed server domain using the **full** profile are shown respectively.

```
/subsystem=messaging-activemq/server=default/address-setting=news.europe.#/:add(dead-letter-
address=DLQ.news)
```

```
/profile=full/subsystem=messaging-activemq/server=default/address-
setting=news.europe.#/:add(dead-letter-address=DLQ.news)
```

Edit an address-setting attribute

Use the **write-attribute** operation to write a new value to an attribute. You can use tab completion to help complete the command string as you type, as well as to expose the available attributes. The following example updates the **max-delivery-attempts** value to **10**.

```
/subsystem=messaging-activemq/server=default/address-setting=news.europe.#/:write-
attribute(name=max-delivery-attempts,value=10)
```

```
/profile=full/subsystem=messaging-activemq/server=default/address-setting=news.europe.#/:write-
attribute(name=max-delivery-attempts,value=10)
```

Read address-setting Attributes

Confirm the values are changed by running the **read-resource** operation with the **include-runtime=true** parameter to expose all current values active in the server model.

```
/subsystem=messaging-activemq/server=default/address-setting=news.europe.#/:read-
resource(include-runtime=true)
```

```
/profile=full/subsystem=messaging-activemq/server=default/address-setting=news.europe.#/:read-
resource(include-runtime=true)
```

Configuring Address Settings Using the Management Console

You can use the management console to create and review address settings by following these steps:

1. Log in to the management console.
2. Select the **Configuration** tab at the top of the screen. When running a managed domain, select the profile to update.
3. Select **Messaging (ActiveMQ) → Server**.
4. Select a messaging server. In the default configuration, only one server, called **default**, is shown.
5. Select **Destinations** and click **View**.
6. Select the **Address Setting** tab to configure address settings.

Remember that when adding a new pattern, for example **news.europe.#**, the **Pattern** field refers to the **name** attribute of the **address-setting** element. You enter this value when using the management CLI to read or write attributes.

You can edit only the **dead-letter-address**, **expiry-address**, **redelivery-delay**, and **max-delivery-attempts** attributes while using the management console. Other attributes must be configured using the management CLI.

Configure Global Resource Usage for Messaging Servers

Three attributes in the **address-setting** element help you control the global resources usage for messaging servers:

Attribute	Description
global-max-memory-size	Controls the maximum amount of memory that Artemis can use to store messages for its addresses before they are considered full and their **address-full-policy** starts to apply. The default value is **-1**, indicating no limit.
global-max-disk-usage	Controls the maximum space Artemis can use to store data in the file system. When the limit is reached, any new message is blocked. This attribute is expressed in percentage of the available space on the disk. The minimum is **0%** and the maximum is **100%**. The default value is **100%**.
disk-scan-period	Controls the frequency at which Artemis checks the available space on the file system. The default value is **5000 milliseconds**.

6.3. LAST-VALUE QUEUES

Last-value queues are special queues which discard any messages when a newer message with the same value for a well-defined last-value property is put in the queue. In other words, a last-value queue only retains the last value. A typical application of a last-value queue might involve stock prices, where you are interested only in the latest price of a particular stock.

IMPORTANT

Last-value queues will not work as expected if the queue has paging enabled. Be sure to disable paging before using a last-value queue.

Configuring Last-value Queues

Last-value queues are defined within the **address-setting** configuration element:

```
<address-setting name="jms.queue.lastValueQueue" last-value-queue="true" />
```

Use the management CLI to read the value of **last-value-queue** for a given **address-setting**:

```
/subsystem=messaging-activemq/server=default/address-setting=news.europe.#:read-attribute(name=last-value-queue)
{
    "outcome" => "success",
    "result" => false
}
```

The accepted values for **last-value-queue** are **true** or **false**. Use the management CLI to set either value, like so:

```
/subsystem=messaging-activemq/server=default/address-setting=news.europe.#:write-attribute(name=last-value-queue,value=true)

/subsystem=messaging-activemq/server=default/address-setting=news.asia.#:write-attribute(name=last-value-queue,value=false)
```

Using the Last-value Property

The property name used to identify the last value is **_AMQ_LVQ_NAME** (or the constant **Message.HDR_LAST_VALUE_NAME** from the Core API). Let the following Java code illustrate how to use the last-value property.

- First, the publisher sends a message to the last-value queue

```
TextMessage message = session.createTextMessage("My 1st message with the last-value property set");
message.setStringProperty("_AMQ_LVQ_NAME", "MY_MESSAGE");
producer.send(message);
```

- Then it sends another message to the queue using the same last-value

```
message = session.createTextMessage("My 2nd message with the last-value property set");
message.setStringProperty("_AMQ_LVQ_NAME", "MY_MESSAGE");
producer.send(message);
```

- Next, the consumer receives the message with the last-value

```
TextMessage messageReceived = (TextMessage)messageConsumer.receive(5000);
System.out.format("Received message: %s\n", messageReceived.getText());
```

In the above example the client's output would be **"My 2nd message with the last-value property set"** since both messages set **_AMQ_LVQ_NAME** to **"MY_MESSAGE"**, and the second message was received in the queue after the first.

CHAPTER 7. CONFIGURING SECURITY

7.1. SECURING REMOTE CONNECTIONS

7.1.1. Using the Legacy Security Subsystem

You can use the legacy **security** subsystem in JBoss EAP to secure the **messaging-activemq** subsystem. The legacy **security** subsystem uses legacy security realms and domains. See the JBoss EAP *Security Architecture* guide for more information on security realms and security domains. The **messaging-activemq** subsystem is pre-configured to use the security realm named **ApplicationRealm** and the security domain named **other**.

NOTE

The legacy **security** subsystem approach is the default configuration from JBoss EAP 7.0.

The **ApplicationRealm** is defined near the top of the configuration file.

```
<management>
  <security-realms>
    ...
    <security-realm name="ApplicationRealm">
      <authentication>
        <local default-user="$local" allowed-users="*" skip-group-loading="true"/>
        <properties
         path="application-users.properties"
         relative-to="jboss.server.config.dir" />
      </authentication>
      <authorization>
        <properties
         path="application-roles.properties"
         relative-to="jboss.server.config.dir" />
      </authorization>
    </security-realm>
  </security-realms>
  ...
</management>
```

As its name implies, **ApplicationRealm** is the default security realm for all application-focused subsystems in JBoss EAP such as the **messaging-activemq**, **undertow**, and **ejb3** subsystems. **ApplicationRealm** uses the local filesystem to store usernames and hashed passwords. For convenience JBoss EAP includes a script that you can use to add users to the **ApplicationRealm**. See Default User Configuration in the JBoss EAP *How To Configure Server Security* guide for details.

The **other** security domain is the default security domain for the application-related subsystems like **messaging-activemq**. It is not explicitly declared in the configuration; however, you can confirm which security domain is used by the **messaging-activemq** subsystem with the following management CLI command:

```
/subsystem=messaging-activemq/server=default:read-attribute(name=security-domain)
{
    "outcome" => "success",
    "result" => "other"
}
```

You can also update which security domain is used:

```
/subsystem=messaging-activemq/server=default:write-attribute(name=security-domain,
value=mySecurityDomain)
```

The JBoss EAP *How To Configure Server Security* guide has more information on how to create new security realms and domains. For now, it is worth noting how the **other** domain appears in the configuration:

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
    <security-domains>
       <security-domain name="other" cache-type="default">
          <authentication>
             <login-module code="Remoting" flag="optional">
                <module-option name="password-stacking" value="useFirstPass"/>
             </login-module>
             <login-module code="RealmDirect" flag="required">
                <module-option name="password-stacking" value="useFirstPass"/>
             </login-module>
          </authentication>
       </security-domain>
       ...
    <security-domains>
 </subsystem>
```

The 'other' domain uses two login-modules as its means of authentication. The first module, **Remoting**, authenticates remote Jakarta Enterprise Beans invocations, while the **RealmDirect** module uses the information store defined in a given realm to authenticate users. In this case the default realm **ApplicationRealm** is used, since no realm is declared. Each module has its **password-stacking** option set to **useFirstPass**, which tells the login-module to store the principal name and password of the authenticated user. See the JBoss EAP *Login Module Reference* for more details on the login modules and their options.

Role-based access is configured at the address level, see Role Based Security for Addresses.

7.1.2. Using the Elytron Subsystem

You can also use the **elytron** subsystem to secure the **messaging-activemq** subsystem. You can find more information on using the **elytron** subsystem and creating and Elytron security domains in the Elytron Subsystem section of *How to Configure Identity Management* guide.

To use an Elytron security domain:

1. Undefine the legacy security domain.

   ```
   /subsystem=messaging-activemq/server=default:undefine-attribute(name=security-domain)
   ```

2. Set an Elytron security domain.

   ```
   /subsystem=messaging-activemq/server=default:write-attribute(name=elytron-domain,
   value=myElytronSecurityDomain)
   ```

```
reload
```

7.1.2.1. Setting an Elytron Security Domain Using the Management Console

To set an Elytron security domain using the management console:

1. Access the management console. For more information, see Management Console in the JBoss EAP Configuration Guide.
2. Navigate to **Configuration** → **Subsystems** → **Messaging (ActiveMQ)** → **Server** → **default** and click **View**.
3. Navigate to the **Security** tab and click **Edit**.
4. Add or edit the value of **Elytron Domain**.
5. Click **Save** to save the changes.
6. Reload the server for the changes to take effect.

NOTE

You can only define either **security-domain** or **elytron-domain**, but you cannot have both defined at the same time. If neither is defined, JBoss EAP will use the **security-domain** default value of **other**, which maps to the **other** legacy security domain.

7.1.3. Securing the Transport

The default **http-connector** that comes bundled with JBoss EAP messaging is not secured by default. You can secure the message transport and enable web traffic for SSL/TLS by following the instructions to configure one-way and two-way SSL/TLS for applications in *How to Configure Server Security* for JBoss EAP.

NOTE

The above approach to secure a message transport also works for securing the **http-acceptor**.

When you configure the transport as described above, you must perform the following additional steps.

- By default, all HTTP acceptors are configured to use the default **http-listener**, which listens on the HTTP port. You must configure HTTP acceptors to use the **https-listener**, which listens on the HTTPS port.
- The **socket-binding** element for all HTTP connectors must be updated to use **https** instead of **http**.
- Each **http-connector** that communicates through SSL/TLS must set the **ssl-enabled** parameter to **true**.
- If an HTTP connector is used to connect to another server, you must configure the related parameters such as **trust-store** and **key-store**. Securing the **http-connector** requires that you configure the same parameters as you do with a **remote-connector**, which is documented in Securing a Remote Connector.

See Configuring the Messaging Transports for information about the configuring acceptors and connectors for messaging transports.

7.1.4. Securing a Remote Connector

If you are not using the default **http-connector** and have instead created your own **remote-connector** and **remote-acceptor** for TCP communications, you can configure each for SSL/TLS by using the properties in the table below. The properties appear in the configuration as part of the child **<param>** elements of the acceptor or connector.

Typically, a server owns its private SSL/TLS key and shares its public key with clients. In this scenario, the server defines the **key-store-path** and **key-store-password** parameters in a **remote-acceptor**. Since each client can have its truststore located at a different location, and be encrypted by a different password, specifying the **trust-store-path** and **trust-store-password** properties on the **remote-connector** is not recommended. Instead, configure these parameters on the client side using the system properties **javax.net.ssl.trustStore** and **javax.net.ssl.trustStorePassword**. The parameters you need to configure for a **remote-connector** are **ssl-enabled=true** and **useDefaultSslContext=true**. However, if the server uses **remote-connector** to connect to another server, it makes sense in this case to set the **trust-store-path** and **trust-store-password** parameters of the **remote-connector**.

In the above use case, the **remote-acceptor** would be created using the following management CLI command:

```
/subsystem=messaging-activemq/server=default/remote-acceptor=mySslAcceptor:add(socket-binding=netty,params={ssl-enabled=true, key-store-path=PATH/TO/server.jks, key-store-password=${VAULT::server-key::key-store-password::sharedKey}})
```

To create the **remote-connector** from the above use case, use the following management CLI command:

```
/subsystem=messaging-activemq/server=default/remote-connector=mySslConnector:add(socket-binding=netty,params={ssl-enabled=true, useDefaultSslContext=true})
```

The management CLI also allows you to add a parameter to an already existing **remote-acceptor** or **remote-connector** as well:

```
/subsystem=messaging-activemq/server=default/remote-connector=myOtherSslConnector:map-put(name=params,key=ssl-enabled,value=true)
```

Note that the **remote-acceptor** and **remote-connector** both reference a **socket-binding** to declare the port to be used for communication. See the Overview of the Messaging Subsystem Configuration for more information on socket bindings and their relationship to acceptors and connectors.

Table 7.1. SSL/TLS-related Configuration Properties for theNettyConnectorFactory

Property	Description
enabled-cipher-suites	Can be used to configure an acceptor or connector. This is a comma separated list of cipher suites used for SSL/TLS communication. The default value is null which means the JVM’s default will be used.

Property	Description
enabled-protocols	Can be used to configure an acceptor or connector. This is a comma separated list of protocols used for SSL/TLS communication. The default value is null which means the JVM's default will be used.
key-store-password	When used on an acceptor, this is the password for the server-side keystore. When used on a connector, this is the password for the client-side keystore. This is only relevant for a connector if you are using two-way SSL/TLS. Although this value can be configured on the server, it is downloaded and used by the client. If the client needs to use a different password from that set on the server, it can override the server-side setting by either using the standard **javax.net.ssl.keyStorePassword** system property. Use the **org.apache.activemq.ssl.keyStorePassword** property if another component on the client is already making use of the standard system property.
key-store-path	When used on an acceptor, this is the path to the SSL/TLS keystore on the server which holds the server's certificates. Use for certificates either self-signed or signed by an authority. When used on a connector, this is the path to the client-side SSL/TLS keystore which holds the client certificates. This is only relevant for a connector if you are using two-way SSL/TLS. Although this value is configured on the server, it is downloaded and used by the client. If the client needs to use a different path from that set on the server, it can override the server-side setting by using the standard **javax.net.ssl.keyStore** system property. Use the **org.apache.activemq.ssl.keyStore** system property if another component on the client is already making use of the standard property.
key-store-provider	Defines the format of the file in which keys are stored, PKCS11 or PKCS12 for example. The accepted values are JDK specific.
needs-client-auth	This property is only for an acceptor. It tells a client connecting to this acceptor that two-way SSL/TLS is required. Valid values are **true** or **false**. Default is **false**.
ssl-enabled	Must be **true** to enable SSL/TLS. Default is **false**.

Property	Description
trust-store-password	When used on an acceptor, this is the password for the server-side truststore. This is only relevant for an acceptor if you are using two-way SSL/TLS. When used on a connector, this is the password for the client-side truststore. Although this value can be configured on the server, it is downloaded and used by the client. If the client needs to use a different password from that set on the server, it can override the server-side setting by using either the standard **javax.net.ssl.trustStorePassword** system property. Use the **org.apache.activemq.ssl.trustStorePassword** system property if another component on the client is already making use of the standard property.
trust-store-path	When used on an acceptor, this is the path to the server-side SSL/TLS keystore that holds the keys of all the clients that the server trusts. This is only relevant for an acceptor if you are using two-way SSL/TLS. When used on a connector, this is the path to the client-side SSL/TLS keystore which holds the public keys of all the servers that the client trusts. Although this value can be configured on the server, it is downloaded and used by the client. If the client needs to use a different path from that set on the server, it can override the server-side setting by using either the standard **javax.net.ssl.trustStore** system property. Use the **org.apache.activemq.ssl.trustStore** system property if another component on the client is already making use of the standard system property.
trust-store-provider	Defines the format of the file in which keys are stored, PKCS11 or PKCS12 for example. The accepted values are JDK specific.

7.2. SECURING DESTINATIONS

In addition to securing remote connections into the messaging server, you can also configure security around specific destinations. This is done by adding a security constraint using the **security-setting** configuration element. JBoss EAP messaging comes with a **security-setting** configured by default, as shown in the output from the following management CLI command:

```
/subsystem=messaging-activemq/server=default:read-resource(recursive=true)
{
    "outcome" => "success",
    "result" => {
        ....
        "security-setting" => {"#" => {"role" => {"guest" => {
            "consume" => true,
            "create-durable-queue" => false,
            "create-non-durable-queue" => true,
            "delete-durable-queue" => false,
                "delete-non-durable-queue" => true,
                "manage" => false,
                "send" => true
            }}}}
        }
    }
```

The **security-setting** option makes use of wildcards in the **name** field to handle which destinations to apply the security constraint. The value of a single **#** will match any address. For more information on using wildcards in security constraints, see Role Based Security for Addresses .

7.2.1. Role-Based Security for Addresses

JBoss EAP messaging contains a flexible role-based security model for applying security to queues, based on their addresses.

The core JBoss EAP messaging server consists mainly of sets of queues bound to addresses. When a message is sent to an address, the server first looks up the set of queues that are bound to that address and then routes the message to the bound queues.

JBoss EAP messaging has a set of permissions that can be applied against queues based on their address. An exact string match on the address can be used or a wildcard match can be used using the wildcard characters **#** and *****. See Address Settings for more information on how to use the wildcard syntax.

You can create multiple roles for each **security-setting**, and there are 7 permission settings that can be applied to a role. Below is the complete list of the permissions available:

- **create-durable-queue** allows the role to create a durable queue under matching addresses.
- **delete-durable-queue** allows the role to delete a durable queue under matching addresses.
- **create-non-durable-queue** allows the role to create a non-durable queue under matching addresses.
- **delete-non-durable-queue** allows the role to delete a non-durable queue under matching addresses.
- **send** allows the role to send a message to matching addresses.
- **consume** allows the role to consume a message from a queue bound to matching addresses.
- **manage** allows the role to invoke management operations by sending management messages to the management address.

Configuring Role-Based Security

To start using role-based security for a **security-setting**, you first must create one. As an example, a **security-setting** of **news.europe.#** is created below. It would apply to any destination starting with **news.europe.**, such as **news.europe.fr** or **news.europe.tech.uk**.

```
/subsystem=messaging-activemq/server=default/security-setting=news.europe.#:add()
{"outcome" => "success"}
```

Next, you add a role to the **security-setting** you created and declare permissions for it. In the example below, the **dev** role is created and given permissions to consume from, and send to, queues, as well as to create and delete non-durable queues. Because the default is **false**, you have to tell JBoss EAP only about the permissions you want to switch on.

```
/subsystem=messaging-activemq/server=default/security-
setting=news.europe.#/role=dev:add(consume=true,delete-non-durable-queue=true,create-non-
durable-queue=true,send=true)
{"outcome" => "success"}
```

To further illustrate the use of permissions, the example below creates an **admin** role and allows it to send management messages by switching on the **manage** permission. The permissions for creating and deleting durable queues are switched on as well:

```
/subsystem=messaging-activemq/server=default/security-
setting=news.europe.#/role=admin:add(manage=true,create-durable-queue=true,delete-durable-
queue=true)
{"outcome" => "success"}
```

To confirm the configuration of a **security-setting**, use the management CLI. Remember to use the **recursive=true** option to get the full display of permissions:

```
/subsystem=messaging-activemq/server=default:read-children-resources(child-type=security-
setting,recursive=true)
{
    "outcome" => "success",
    "result" => {
        "#" => {"role" => {"guest" => {
            "consume" => true,
            "create-durable-queue" => false,
            "create-non-durable-queue" => true,
            "delete-durable-queue" => false,
            "delete-non-durable-queue" => true,
            "manage" => false,
            "send" => true
        }}},
        "news.europe.#" => {"role" => {
            "dev" => {
                "consume" => true,
                "create-durable-queue" => false,
                "create-non-durable-queue" => true,
                "delete-durable-queue" => false,
                "delete-non-durable-queue" => true,
                "manage" => false,
                "send" => true
            },
            "admin" => {
                "consume" => false,
                "create-durable-queue" => true,
                "create-non-durable-queue" => false,
                "delete-durable-queue" => true,
                "delete-non-durable-queue" => false,
                "manage" => true,
                "send" => false
            }
        }}
    }
```

Above, the permissions for addresses that start with string **news.europe.** are displayed in full by the management CLI. To summarize, only users who have the **admin** role can create or delete durable queues, while only users with the **dev** role can create or delete non-durable queues. Furthermore, users with the **dev** role can send or consume messages, but **admin** users cannot. They can, however, send management messages since their **manage** permission is set to **true**.

In cases where more than one match applies to a set of addresses the more specific match takes precedence. For example, the address **news.europe.tech.uk.#** is more specific than **news.europe.tech.#**. Because permissions are not inherited, you can effectively deny permissions in more specific **security-setting** blocks by simply not specifying them. Otherwise it would not be possible to deny permissions in sub-groups of addresses.

The mapping between a user and what roles they have is handled by the security manager. JBoss EAP ships with a user manager that reads user credentials from a file on disk, and can also plug into JAAS or JBoss EAP security.

For more information on configuring the security manager, see the JBoss EAP *Security Architecture* guide.

7.2.1.1. Granting Unauthenticated Clients the guest Role Using the Legacy Security Subsystem

If you want JBoss EAP to automatically grant unauthenticated clients the **guest** role make the following two changes:

1. Add a new **module-option** to the **other** security domain. The new option, **unauthenticatedIdentity**, will tell JBoss EAP to grant **guest** access to unauthenticated clients. The recommended way to do this is by using the management CLI:

   ```
   /subsystem=security/security-domain=other/authentication=classic/login-
   module=RealmDirect:map-put(name=module-
   options,key=unauthenticatedIdentity,value=guest)
   {
     "outcome" => "success",
     "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
     }
   }
   ```

   Note that the server requires a reload after issuing the command. You can confirm the new option by using the following management CLI command:

   ```
   /subsystem=security/security-domain=other/authentication=classic/login-
   module=RealmDirect:read-resource()
   {
      "outcome" => "success",
      "result" => {
         "code" => "RealmDirect",
         "flag" => "required",
         "module" => undefined,
         "module-options" => {
            "password-stacking" => "useFirstPass",
            "unauthenticatedIdentity" => "guest"
   ```

```
        }
    }
}
```

Also, your server configuration file should look something like this after the command executes:

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
  <security-domains>
    <security-domain name="other" cache-type="default">
      <authentication>
        ...
        <login-module code="RealmDirect" flag="required">
          ...
          <module-option name="unauthenticatedIdentity" value="guest"/>
          ...
        </login-module>
        ...
      </authentication>
    </security-domain>
  ...
  </security-domains>
</subsystem>
```

2. Uncomment the following line in the file **application-roles.properties** by deleting the **#** character. The file is located in ***EAP_HOME*/standalone/configuration/** or ***EAP_HOME*/domain/configuration/**, depending on whether you are using standalone servers or a domain controller respectively.

```
#guest=guest
```

Remote clients should now be able to access the server without needing to authenticate. They will be given the permissions associated with the **guest** role.

7.3. CONTROLLING JAKARTA MESSAGING OBJECTMESSAGE DESERIALIZATION

Because an **ObjectMessage** can contain potentially dangerous objects, ActiveMQ Artemis provides a simple class filtering mechanism to control which packages and classes are to be trusted and which are not. You can add objects whose classes are from trusted packages to a white list to indicate they can be deserialized without a problem. You can add objects whose classes are from untrusted packages to a black list to prevent them from being deserialized.

ActiveMQ Artemis filters objects for deserialization as follows.

- If both the white list and the black list are empty, which is the default, any serializable object is allowed to be deserialized.
- If an object’s class or package matches one of the entries in the black list, it is not allowed to be deserialized.
- If an object’s class or package matches an entry in the white list, it is allowed to be deserialized.
- If an object’s class or package matches an entry in both the black list and the white list, the one in black list takes precedence, meaning it is not allowed to be deserialized.

- If an object's class or package matches neither the black list nor the white list, the object deserialization is denied, unless the white list is empty, meaning there is no white list specified.

An object is considered a match if its full name exactly matches one of the entries in the list, if its package matches one of the entries in the list, or if it is a subpackage of one of the entries in the list.

You can specify which objects can be deserialized on a **connection-factory** and on a **pooled-connection-factory** using the **deserialization-white-list** and **deserialization-black-list** attributes. The **deserialization-white-list** attribute is used to define the list of classes or packages that are allowed to be deserialized. The **deserialization-black-list** attribute is used to define the list of classes or packages that are not allowed to be deserialized.

The following commands create a black list for the **RemoteConnectionFactory** connection factory and a white list for the **activemq-ra** pooled connection factory for the default server.

```
/subsystem=messaging-activemq/server=default/connection-
factory=RemoteConnectionFactory:write-attribute(name=deserialization-black-list,value=
[my.untrusted.package,another.untrusted.package])
/subsystem=messaging-activemq/server=default/pooled-connection-factory=activemq-ra:write-
attribute(name=deserialization-white-list,value=[my.trusted.package])
```

These commands generate the following configuration in the **messaging-activemq** subsystem.

```
<connection-factory name="RemoteConnectionFactory"
entries="java:jboss/exported/jms/RemoteConnectionFactory" connectors="http-connector" ha="true"
block-on-acknowledge="true" reconnect-attempts="-1" deserialization-black-
list="my.untrusted.package another.untrusted.package"/>
<pooled-connection-factory name="activemq-ra" entries="java:/JmsXA
java:jboss/DefaultJMSConnectionFactory" connectors="in-vm" deserialization-white-
list="my.trusted.package" transaction="xa"/>
```

For information about connection factories and pooled connection factories, see Configuring Connection Factories in this guide.

You can also specify which objects can be deserialized in an MDB by configuring the activation properties. The **deserializationWhiteList** property is used to define the list of classes or packages that are allowed to be deserialized. The **deserializationBlackList** property is used to define the list of classes or packages that are not allowed to be deserialized. For more information about activation properties, see Configuring MDBs Using a Deployment Descriptor in *Developing Jakarta Enterprise Beans Applications* for JBoss EAP.

7.4. AUTHORIZATION INVALIDATION MANAGEMENT

The **security-invalidation-interval** attribute on the server in the **messaging-activemq** subsystem determines how long an authorization is cached before an action must be re-authorized.

When the system authorizes a user to perform an action at an address, the authorization is cached. The next time the same user performs the same action at the same address, the system uses the cached authorization for the action.

For example, the user **admin** attempts to send a message to the address **news**. The system authorizes the action, and caches the authorization. The next time **admin** attempts to send a message to **news**, the system uses the cached authorization.

If the cached authorization is not used again within the time specified by the invalidation interval, the authorization is cleared from the cache. The system must re-authorize the user to perform the requested action at the requested address.

After installation, JBoss EAP assumes a default value of 10000 milliseconds (10 seconds).

```
/subsystem=messaging-activemq/server=default:read-attribute(name=security-invalidation-interval)
{
    "outcome" => "success",
    "result" => 10000L
}
```

The **security-invalidation-interval** attribute is configurable. For example, the following command updates the interval to 60000 milliseconds (60 seconds or one minute).

```
/subsystem=messaging-activemq/server=default:write-attribute(name=security-invalidation-
interval,value=60000)
{
    "outcome" => "success",
    "response-headers" => {
        "operation-requires-reload" => true,
        "process-state" => "reload-required"
    }
}
```

You must reload the server for the modification of the configuration to take effect.

Reading the attribute shows the new result.

```
/subsystem=messaging-activemq/server=default:read-attribute(name=security-invalidation-interval)
{
    "outcome" => "success",
    "result" => 60000L
}
```

CHAPTER 8. CONFIGURING THE MESSAGING TRANSPORTS

This section describes the concepts critical to understanding JBoss EAP messaging transports, specifically connectors and acceptors. Acceptors are used on the server to define how it can accept connections, while connectors are used by the client to define how it connects to a server. Each concept is discussed in turn and then a practical example shows how clients can make connections to a JBoss EAP messaging server, using JNDI or the Core API.

8.1. ACCEPTOR AND CONNECTOR TYPES

There are three main types of acceptor and connector defined in the configuration of JBoss EAP.

in-vm: In-vm is short for Intra Virtual Machine. Use this connector type when both the client and the server are running in the same JVM, for example, Message Driven Beans (MDBs) running in the same instance of JBoss EAP.

http: Used when client and server are running in different JVMs. Uses the **undertow** subsystem's default port of **8080** and is thus able to multiplex messaging communications over HTTP. Red Hat recommends using the **http** connector when the client and server are running in different JVMs due to considerations such as port management, especially in a cloud environment.

remote: Remote transports are Netty-based components used for native TCP communication when the client and server are running in different JVMs. An alternative to **http** when it cannot be used.

A client must use a connector that is compatible with one of the server's acceptors. For example, only an **in-vm-connector** can connect to an **in-vm-acceptor**, and only a **http-connector** can connect to an **http-acceptor**, and so on.

You can have the management CLI list the attributes for a given acceptor or connector type using the **read-children-attributes** operation. For example, to see the attributes of all the **http-connectors** for the default messaging server you would enter:

```
/subsystem=messaging-activemq/server=default:read-children-resources(child-type=http-connector,include-runtime=true)
```

The attributes of all the **http-acceptors** are read using a similar command:

```
/subsystem=messaging-activemq/server=default:read-children-resources(child-type=http-acceptor,include-runtime=true)
```

The other acceptor and connector types follow the same syntax. Just provide **child-type** with the acceptor or connector type, for example, **remote-connector** or **in-vm-acceptor**.

8.2. ACCEPTORS

An acceptor defines which types of connection are accepted by the JBoss EAP integrated messaging server. You can define any number of acceptors per server. The sample configuration below is modified from the default **full-ha** configuration profile and provides an example of each acceptor type.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <http-acceptor name="http-acceptor" http-listener="default"/>
    <remote-acceptor name="legacy-messaging-acceptor" socket-binding="legacy-messaging"/>
    <in-vm-acceptor name="in-vm" server-id="0"/>
    ...
  </server>
</subsystem>
```

In the above configuration, the **http-acceptor** is using Undertow's default **http-listener** which listens on JBoss EAP's default http port, 8080. The **http-listener** is defined in the **undertow** subsystem:

```
<subsystem xmlns="urn:jboss:domain:undertow:10.0">
  ...
  <server name="default-server">
    <http-listener name="default" redirect-socket="https" socket-binding="http"/>
    ...
  </server>
  ...
</subsystem>
```

Also note how the **remote-acceptor** above uses the **socket-binding** named **legacy-messaging**, which is defined later in the configuration as part of the server's default **socket-binding-group**.

```
<server xmlns="urn:jboss:domain:8.0">
  ...
  <socket-binding-group name="standard-sockets" default-interface="public" port-
offset="${jboss.socket.binding.port-offset:0}">
      ...
      <socket-binding name="legacy-messaging" port="5445"/>
      ...
  </socket-binding-group>
</server>
```

In this example, the **legacy-messaging socket-binding** binds JBoss EAP to port **5445**, and the **remote-acceptor** above claims the port on behalf of the **messaging-activemq** subsystem for use by legacy clients.

Lastly, the **in-vm-acceptor** uses a unique value for the **server-id** attribute so that this server instance can be distinguished from other servers that might be running in the same JVM.

8.3. CONNECTORS

A connector defines how to connect to an integrated JBoss EAP messaging server, and is used by a client to make connections.

You might wonder why connectors are defined on the server when they are actually used by the client. The reasons for this include:

- In some instances, the server might act as a client when it connects to another server. For example, one server might act as a bridge to another, or it might want to participate in a cluster. In such cases, the server needs to know how to connect to other servers, and that is defined by connectors.
- A server can provide connectors using a **ConnectionFactory** which is looked up by clients using JNDI, so creating connection to the server is simpler.

You can define any number of connectors per server. The sample configuration below is based on the **full-ha** configuration profile and includes connectors of each type.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <http-connector name="http-connector" endpoint="http-acceptor" socket-binding="http" server-name="messaging-server-1"/>
    <remote-connector name="legacy-remoting-connector" socket-binding="legacy-remoting"/>
    <in-vm-connector name="in-vm" server-id="0"/>
    ...
  </server>
</subsystem>
```

Like the **http-acceptor** from the **full-ha** profile, the **http-connector** uses the default **http-listener** defined by the **undertow** subsystem. The **endpoint** attribute declares which **http-acceptor** to connect to. In this case, the connector will connect to the default **http-acceptor**.

JBoss EAP 7.1 introduced a new **server-name** attribute for the **http-connector**. This new attribute is optional, but it is required to be able to connect to the correct **http-acceptor** on a remote server that is running more than one ActiveMQ Artemis instance. If this attribute is not defined, the value is resolved at runtime to be the name of the parent ActiveMQ Artemis server in which the connector is defined.

Also, note that the **remote-connector** references the same **socket-binding** as its **remote-acceptor** counterpart. Lastly, the **in-vm-connector** uses the same value for **server-id** as the **in-vm-acceptor** since they both run inside the same server instance.

NOTE

If the bind address for the public interface is set to **0.0.0.0**, you will see the following warning in the log when you start the JBoss EAP server:

```
AMQ121005: Invalid "host" value "0.0.0.0" detected for "connector" connector.
Switching to <HOST_NAME>. If this new address is incorrect please manually
configure the connector to use the proper one.
```

This is because a remote connector cannot connect to a server using the **0.0.0.0** address and the **messaging-activemq** subsystem tries to replace it with the server's host name. The administrator should configure the remote connector to use a different interface address for the socket binding.

8.4. CONFIGURING ACCEPTORS AND CONNECTORS

There are a number of configuration options for connectors and acceptors. They appear in the configuration as child **<param>** elements. Each **<param>** element includes a **name** and **value** attribute pair that is understood and used by the default Netty-based factory class responsible for instantiating a connector or acceptor.

In the management CLI, each remote connector or acceptor element includes an internal map of the parameter name and value pairs. For example, to add a new **param** to a **remote-connector** named **myRemote** use the following command:

```
/subsystem=messaging-activemq/server=default/remote-connector=myRemote:map-put(name=params,key=foo,value=bar)
```

Retrieve parameter values using a similar syntax.

```
/subsystem=messaging-activemq/server=default/remote-connector=myRemote:map-get(name=params,key=foo)
{
    "outcome" => "success",
    "result" => "bar"
}
```

You can also include parameters when you create an acceptor or connector, as in the example below.

```
/subsystem=messaging-activemq/server=default/remote-connector=myRemote:add(socket-binding=mysocket,params={foo=bar,foo2=bar2})
```

Table 8.1. Transport Configuration Properties

Property	Description
batch-delay	Before writing packets to the transport, the messaging server can be configured to batch up writes for a maximum of **batch-delay** in milliseconds. This increases the overall throughput for very small messages by increasing average latency for message transfer. The default is 0.
direct-deliver	When a message arrives on the server and is delivered to waiting consumers, by default, the delivery is done on the same thread on which the message arrived. This gives good latency in environments with relatively small messages and a small number of consumers but reduces the throughput and latency. For highest throughput you can set this property as **false**. The default is**true**.
http-upgrade-enabled	Used by an **http-connector** to specify that it is using HTTP upgrade and therefore is multiplexing messaging traffic over HTTP. This property is set automatically by JBoss EAP to **true** when the **http-connector** is created and does not require an administrator.
http-upgrade-endpoint	Specifies the **http-acceptor** on the server-side to which the**http-connector** will connect. The connector will be multiplexed over HTTP and needs this info to find the relevant **http-acceptor** after the HTTP upgrade. This property is set automatically by JBoss EAP when the **http-connector** is created and does not require an administrator.
local-address	For a http or a remote connector, this is used to specify the local address which the client will use when connecting to the remote address. If a local address is not specified then the connector will use any available local address.
local-port	For a http or a remote connector, this is used to specify which local port the client will use when connecting to the remote address. If the local-port default is used (0) then the connector will let the system pick up an ephemeral port. Valid port values are **0** to **65535**.

Property	Description
nio-remoting-threads	If configured to use NIO, the messaging will by default use a number of threads equal to three times the number of cores (or hyper-threads) as reported by **Runtime.getRuntime().availableProcessors()** for processing incoming packets. To override this value, you can set a custom value for the number of threads. The default is **-1**.
tcp-no-delay	If this is **true** then Nagle's algorithm will be enabled. This algorithm helps improve the efficiency of TCP/IP networks by reducing the number of packets sent over a network. The default is **true**.
tcp-send-buffer-size	This parameter determines the size of the TCP send buffer in bytes. The default is **32768**.
tcp-receive-buffer-size	This parameter determines the size of the TCP receive buffer in bytes. The default is **32768**.
use-nio-global-worker-pool	This parameter will ensure all Jakarta Messaging connections share a single pool of Java threads, rather than each connection having its own pool. This serves to avoid exhausting the maximum number of processes on the operating system. The default is **true**.

8.5. CONNECTING TO A SERVER

If you want to connect a client to a server, you have to have a proper connector. There are two ways to do that. You could use a ConnectionFactory which is configured on the server and can be obtained via JNDI lookup. Alternatively, you could use the ActiveMQ Artemis core API and configure the whole **ConnectionFactory** on the client side.

8.5.1. Jakarta Messaging Connection Factories

Clients can use JNDI to look up ConnectionFactory objects which provide connections to the server. Connection Factories can expose each of the three types of connector:

A **connection-factory** referencing a **remote-connector** can be used by a remote client to send messages to or receive messages from the server (assuming the connection-factory has an appropriately exported entry). A **remote-connector** is associated with a **socket-binding** that tells the client using the **connection-factory** where to connect.

A **connection-factory** referencing an **in-vm-connector** is suitable to be used by a local client to either send messages to or receive messages from a local server. An **in-vm-connector** is associated with a **server-id** which tells the client using the **connection-factory** where to connect, since multiple messaging servers can run in a single JVM.

A **connection-factory** referencing a **http-connector** is suitable to be used by a remote client to send messages to or receive messages from the server by connecting to its HTTP port before upgrading to the messaging protocol. A **http-connector** is associated with the **socket-binding** that represents the HTTP socket, which by default is named **http**.

Since Jakarta Messaging 2.0, a default Jakarta Messaging connection factory is accessible to Jakarta

EE applications under the JNDI name **java:comp/DefaultJMSConnectionFactory**. The **messaging-activemq** subsystem defines a **pooled-connection-factory** that is used to provide this default connection factory.

Below are the default connectors and connection factories that are included in the **full** configuration profile for JBoss EAP:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    [...]
    <http-connector name="http-connector" socket-binding="http" endpoint="http-acceptor" />
    <http-connector name="http-connector-throughput" socket-binding="http" endpoint="http-acceptor-
throughput">
      <param name="batch-delay" value="50"/>
    </http-connector>
    <in-vm-connector name="in-vm" server-id="0"/>
    [...]
    <connection-factory name="InVmConnectionFactory" connectors="in-vm"
entries="java:/ConnectionFactory" />
    <pooled-connection-factory name="activemq-ra" transaction="xa" connectors="in-vm"
entries="java:/JmsXA java:jboss/DefaultJMSConnectionFactory"/>
    [...]
    </server>
</subsystem>
```

The **entries** attribute of a factory specifies the JNDI names under which the factory will be exposed. Only JNDI names bound in the **java:jboss/exported** namespace are available to remote clients. If a **connection-factory** has an entry bound in the **java:jboss/exported** namespace a remote client would look-up the **connection-factory** using the text after **java:jboss/exported**. For example, the **RemoteConnectionFactory** is bound by default to **java:jboss/exported/jms/RemoteConnectionFactory** which means a remote client would look-up this connection-factory using **jms/RemoteConnectionFactory**. A **pooled-connection-factory** should not have any entry bound in the **java:jboss/exported** namespace because a **pooled-connection-factory** is not suitable for remote clients.

8.5.2. Connecting to the Server Using JNDI

If the client resides within the same JVM as the server, it can use the **in-vm** connector provided by the **InVmConnectionFactory**. Here is how the **InvmConnectionFactory** is typically configured, as found for example in **standalone-full.xml**.

```
<connection-factory
  name="InVmConnectionFactory"
  entries="java:/ConnectionFactory"
  connectors="in-vm" />
```

Note the value of the **entries** attribute. Clients using the **InVmConnectionFactory** should drop the leading **java:/** during lookup, as in the following example:

```
InitialContext ctx = new InitialContext();
ConnectionFactory cf = (ConnectionFactory)ctx.lookup("ConnectionFactory");
Connection connection = cf.createConnection();
```

Remote clients use the **RemoteConnectionFactory**, which is usually configured as below:

```
<connection-factory
  name="RemoteConnectionFactory"
  scheduled-thread-pool-max-size="10"
  entries="java:jboss/exported/jms/RemoteConnectionFactory"
  connectors="http-connector"/>
```

Remote clients should ignore the leading **java:jboss/exported/** of the value for **entries**, following the example of the code snippet below:

```
final Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"org.wildfly.naming.client.WildFlyInitialContextFactory");
env.put(Context.PROVIDER_URL, "http-remoting://remotehost:8080");
InitialContext remotingCtx = new InitialContext(env);
ConnectionFactory cf = (ConnectionFactory) remotingCtx.lookup("jms/RemoteConnectionFactory");
```

Note the value for the **PROVIDER_URL** property and how the client is using the JBoss EAP http-remoting protocol. Note also how the client is using the **org.wildfly.naming.client.WildFlyInitialContextFactory**, which implies the client has this class and its encompassing client JAR somewhere in the classpath. For maven projects, this can be achieved by including the following dependency:

```
<dependencies>
  <dependency>
    <groupId>org.wildfly</groupId>
    <artifactId>wildfly-jms-client-bom</artifactId>
    <type>pom</type>
  </dependency>
</dependencies>
```

8.5.3. Connecting to the Server Using the Core API

You can use the Core API to make client connections without needing a JNDI lookup. Clients using the Core API require a client JAR in their classpath, just as JNDI-based clients.

ServerLocator

Clients use **ServerLocator** instances to create **ClientSessionFactory** instances. As their name implies, **ServerLocator** instances are used to locate servers and create connections to them.

In Jakarta Messaging terms think of a **ServerLocator** in the same way you would a Jakarta Messaging Connection Factory.

ServerLocator instances are created using the **ActiveMQClient** factory class.

```
ServerLocator locator = ActiveMQClient.createServerLocatorWithoutHA(new
TransportConfiguration(InVMConnectorFactory.class.getName()));
```

ClientSessionFactory

Clients use a **ClientSessionFactory** to create **ClientSession** instances, which are basically connections to a server. In Jakarta Messaging terms think of them as Jakarta Messaging connections.

ClientSessionFactory instances are created using the **ServerLocator** class.

```
ClientSessionFactory factory =  locator.createClientSessionFactory();
```

ClientSession

A client uses a **ClientSession** for consuming and producing messages and for grouping them in transactions. **ClientSession** instances can support both transactional and non transactional semantics and also provide an XAResource interface so messaging operations can be performed as part of the Jakarta Transactions operation.

ClientSession instances group **ClientConsumers** and **ClientProducers**.

```
ClientSession session = factory.createSession();
```

The simple example below highlights some of what was just discussed:

```
ServerLocator locator = ActiveMQClient.createServerLocatorWithoutHA(
  new TransportConfiguration( InVMConnectorFactory.class.getName()));

// In this simple example, we just use one session for both
// producing and consuming
ClientSessionFactory factory =  locator.createClientSessionFactory();
ClientSession session = factory.createSession();

// A producer is associated with an address ...
ClientProducer producer = session.createProducer("example");
ClientMessage message = session.createMessage(true);
message.getBodyBuffer().writeString("Hello");

// We need a queue attached to the address ...
session.createQueue("example", "example", true);

// And a consumer attached to the queue ...
ClientConsumer consumer = session.createConsumer("example");

// Once we have a queue, we can send the message ...
producer.send(message);

// We need to start the session before we can -receive- messages ...
session.start();
ClientMessage msgReceived = consumer.receive();

System.out.println("message = " + msgReceived.getBodyBuffer().readString());

session.close();
```

8.6. MESSAGING THROUGH A LOAD BALANCER

When using JBoss EAP as a load balancer, clients can call messaging servers behind either a static Undertow HTTP load balancer, or behind a mod_cluster load balancer.

Configurations to support messaging clients calling messaging servers through a static load balancer must meet the following requirements:

- When using JBoss EAP as a load balancer, you must configure the load balancer using HTTP or HTTPS. AJP is not supported for messaging load balancers.
  - For details about configuring Undertow as a static load balancer, see Configure Undertow as a Static Load Balancer in the JBoss EAP *Configuration Guide*.

- If JNDI lookups occur on the messaging servers behind the load balancer, you must configure the back-end messaging workers.
- Clients connecting to the load balancer must reuse the initial connections to the load balancer to ensure they communicate with the same server. Clients connecting to a load balancer must not use the cluster topology to connect to the load balancer. Using the cluster topology might result in messages being sent to a different server, which might result in disruptions to transaction processing.

For details about configuring Undertow as a load balancer using mod_cluster, Configure Undertow as a Load Balancer Using mod_cluster in the JBoss EAP *Configuration Guide*.

Configuration of messaging clients to communicate through a load balancer

Clients that connect to a load balancer must be configured to re-use the initial connection rather than using the cluster topology to connect to the load balancer.

Re-using the initial connection ensures that the client connects to the same server. Using the cluster topology might result in messages being directed to a different server, which might result in disruptions to transaction processing.

A connection factory or pooled connection factory that is used to connect to a load balancer must be configured with the attribute **use-topology-for-load-balancing** set to false. The following example illustrates how to define this configuration in the CLI.

```
/subsystem=messaging-activemq/pooled-connection-factory=remote-artemis:write-
attribute(name=use-topology-for-load-balancing, value=false)
```

Configuring back-end workers

You must configure back-end messaging workers only if you plan to do JNDI lookups behind the load balancer.

1. Create a new outbound socket binding that points to the load-balancing server.

   ```
   /socket-binding-group=standard-sockets/remote-destination-outbound-socket-
   binding=balancer-binding:add(host=load_balance.example.com,port=8080)
   ```

2. Create an HTTP connector that references the load-balancing server socket binding.

   ```
   /subsystem=messaging-activemq/server=default/http-connector=balancer-
   connector:add(socket-binding=balancer-binding, endpoint=http-acceptor)
   ```

3. Add the HTTP connector to the connection factory used by the client.

   ```
   /subsystem=messaging-activemq/server=default/connection-
   factory=RemoteConnectionFactory:write-attribute(name=connectors,value=[balancer-
   connector])
   ```

   Make sure you configure the clients to re-use the initial connection:

   ```
   /subsystem=messaging-activemq/server=default/connection-
   factory=RemoteConnectionFactory:write-attribute(name=use-topology-for-load-
   balancing,value=false)
   ```

CHAPTER 9. CONFIGURING CONNECTION FACTORIES

By default, the JBoss EAP **messaging-activemq** subsystem provides the **InVmConnectionFactory** and **RemoteConnectionFactory** connection factories, as well as the **activemq-ra** pooled connection factory.

Basic Connection Factories

InVmConnectionFactory references an **in-vm-connector** and can be used to send and receive messages when both the client and server are running in the same JVM. **RemoteConnectionFactory** references an **http-connector** and can be used to send and receive messages over HTTP when the client and server are running in different JVMs.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <connection-factory name="InVmConnectionFactory" connectors="in-vm"
entries="java:/ConnectionFactory"/>
    <connection-factory name="RemoteConnectionFactory" connectors="http-connector"
entries="java:jboss/exported/jms/RemoteConnectionFactory"/>
    ...
  </server>
</subsystem>
```

For more information on the different types of connectors, see the Acceptors and Connectors section.

Add a Connection Factory

You can add a new connection factory using the following management CLI command. When adding a connection factory, you must provide the **connectors** and the JNDI **entries**.

```
/subsystem=messaging-activemq/server=default/connection-
factory=MyConnectionFactory:add(entries=[java:/MyConnectionFactory],connectors=[in-vm])
```

Configure a Connection Factory

You can update a connection factory’s settings using the management CLI.

```
/subsystem=messaging-activemq/server=default/connection-factory=MyConnectionFactory:write-
attribute(name=thread-pool-max-size,value=40)
```

For information on the available attributes for a connection factory, see Connection Factory Attributes.

Remove a Connection Factory

You can remove a connection factory using the management CLI.

```
/subsystem=messaging-activemq/server=default/connection-factory=MyConnectionFactory:remove
```

Pooled Connection Factories

The JBoss EAP **messaging-activemq** subsystem provides a pooled connection factory that allows you to configure the inbound and outbound connectors of the integrated ActiveMQ Artemis resource adapter. For more information on configuring a **pooled-connection-factory** to connect to a remote ActiveMQ Artemis server, see Using the Integrated Resource Adapter for Remote Connections .

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
```

```
    ...
    <pooled-connection-factory name="activemq-ra" transaction="xa" entries="java:/JmsXA
java:jboss/DefaultJMSConnectionFactory" connectors="in-vm"/>
  </server>
</subsystem>
```

There are several unique characteristics of a pooled connection factory:

- It is only available to local clients, though it can be configured to point to a remote server. For more information on connecting to a remote ActiveMQ Artemis server, see Using the Integrated Artemis Resource Adapter for Remote Connections.
- It should only be used to send messages when looked up in JNDI or injected.
- It can be configured to use security credentials, which is useful if it is pointing to a secured remote server.
- Resources acquired from it will be automatically enlisted in any ongoing Jakarta Transactions.

Add a Pooled Connection Factory

You can add a new pooled connection factory using the following management CLI command. When adding a connection factory, you must provide the **connectors** and the JNDI **entries**.

```
/subsystem=messaging-activemq/server=default/pooled-connection-
factory=MyPooledConnectionFactory:add(entries=[java:/MyPooledConnectionFactory],connectors=
[in-vm])
```

Configure a Pooled Connection Factory

You can update a pooled connection factory's settings using the management CLI.

```
/subsystem=messaging-activemq/server=default/pooled-connection-
factory=MyPooledConnectionFactory:write-attribute(name=max-retry-interval,value=3000)
```

For information on the available attributes for a pooled connection factory, see Pooled Connection Factory Attributes.

You can disable the recording of enlistment traces for this pooled connection factory using the management CLI by setting the **enlistment-trace** attribute to **false**.

```
/subsystem=messaging-activemq/server=default/pooled-connection-
factory=MyPooledConnectionFactory:write-attribute(name=enlistment-trace,value=false)
```

WARNING

Disabling the enlistment trace will make tracking down errors during transaction enlistment more difficult.

You can also configure the managed connection pool implementation used by the pooled connection factory. For more information, see the Configure Managed Connection Pools section of the JBoss EAP *Configuration Guide*.

Remove a Pooled Connection Factory

You can remove a pooled connection factory using the management CLI.

```
/subsystem=messaging-activemq/server=default/pooled-connection-
factory=MyPooledConnectionFactory:remove
```

CHAPTER 10. CONFIGURING PERSISTENCE

10.1. ABOUT PERSISTENCE IN JBOSS EAP 7 MESSAGING

JBoss EAP ships with two persistence options for storing binding data and messages:

- You can use the default file-based journal, which is highly optimized for messaging use cases and provides great performance. This option is provided by default and is used if you do not do any additional configuration.
- You can store the data in a JDBC data store, which uses JDBC to connect to a database of your choice. This option requires configuration of the **datasources** and **messaging-activemq** subsystems in the server configuration file.

10.2. MESSAGING JOURNAL PERSISTENCE USING THE DEFAULT FILE JOURNAL

JBoss EAP messaging ships with a high-performance, file-based journal that is optimized for messaging.

The JBoss EAP messaging journal has a configurable file size and is append only, which improves performance by enabling single write operations. It consists of a set of files on disk, which are initially pre-created to a fixed size and filled with padding. As server operations, such as add message, delete message, update message, are performed, records of the operations are appended to the journal until the journal file is full, at which point the next journal file is used.

A sophisticated garbage collection algorithm determines whether journal files can be reclaimed and re-used when all of their data has been deleted. A compaction algorithm removes dead space from journal files and compresses the data.

The journal also fully supports both local and XA transactions.

10.2.1. Messaging Journal File System Implementations

The majority of the journal is written in Java, but interaction with the file system has been abstracted to allow different pluggable implementations. The two implementations shipped with JBoss EAP messaging are:

Java New I/O (NIO)

This implementation uses standard Java NIO to interface with the file system. It provides extremely good performance and runs on any platform with a Java 6 or later runtime. Note that JBoss EAP 7 requires Java 8. Using NIO is supported on any operating system that JBoss EAP supports.

Linux Asynchronous IO (ASYNCIO)

This implementation uses a native code wrapper to talk to the Linux asynchronous IO library (ASYNCIO). This implementation removes the need for explicit synchronization. ASYNCIO typically provides better performance than Java NIO.
To check which journal type is in use, issue the following CLI request:

```
/subsystem=messaging-activemq/server=default:read-attribute(name=runtime-journal-type)
```

The system returns one of the following values:

Table 10.1. Journal Type Return Values

Return Value	Description
NONE	Persistence is disabled
NIO	Java NIO is in use
ASYNCIO	AsyncIO with **libaio** is in use
DATABASE	JDBC persistence is in use

The following file systems have been tested and are supported only on Red Hat Enterprise Linux 6, Red Hat Enterprise Linux 7, and Red Hat Enterprise Linux 8 when using the **libaio** natives. They are not tested and are not supported on other operating systems.

- EXT4
- XFS
- NFSv4
- GFS2

The following table lists the HA shared store file systems that have been tested, both with and without the **libaio** natives, and whether they are supported.

Operating System	File System	Supported Using **libaio** Natives? (journal-type="ASYNCIO")	Supported Without Using **libaio** Natives? (journal-type="NIO")
Red Hat Enterprise Linux 6	NFSv4	Yes	Yes
Red Hat Enterprise Linux 7 and later	NFSv4	Yes	Yes
Red Hat Enterprise Linux 6	GFS2	Yes	No
Red Hat Enterprise Linux 7 and later	GFS2	Yes	No

10.2.2. Standard Messaging Journal File System Instances

The standard JBoss EAP messaging core server uses the following journal instances:

Bindings Journal

This journal is used to store bindings related data, including the set of queues that are deployed on the server and their attributes. It also stores data such as id sequence counters.

The bindings journal is always a NIO journal as it is typically low throughput compared to the message journal.

The files on this journal are prefixed as activemq-bindings. Each file has a bindings extension. File size is 1048576, and it is located at the bindings folder.

Jakarta Messaging Journal

This journal instance stores all Jakarta Messaging related data, such as any Jakarta Messaging queues,topics, connection factories and any JNDI bindings for these resources.
Any Jakarta Messaging Resource created via the management API will be persisted to this journal. Any resource configured via configuration files will not. The Jakarta Messaging Journal will only be created if Jakarta Messaging is being used.

The files on this journal are prefixed as activemq-jms. Each file has a **jms** extension. File size is 1048576, and it is located at the bindings folder.

Message Journal

This journal instance stores all message related data, including the message themselves and also duplicate-id caches.
By default JBoss EAP messaging will try to use an ASYNCIO journal. If ASYNCIO is not available, for example the platform is not Linux with the correct kernel version or ASYNCIO has not been installed then it will automatically fall back to using Java NIO which is available on any Java platform.

The files on this journal are prefixed as activemq-data. Each file has an amq extension. File size is by default 10485760 (configurable), and it is located at the journal folder.

For large messages, JBoss EAP messaging persists them outside the message journal. This is discussed in the section on Large Messages.

JBoss EAP messaging can also be configured to page messages to disk in low memory situations. This is discussed in the Paging section.

If no persistence is required at all, JBoss EAP messaging can also be configured not to persist any data at all to storage as discussed in the Configuring JBoss EAP Messaging for Zero Persistence section.

10.2.3. Configuring the Bindings and Jakarta Messaging Journals

Because the bindings journal shares its configuration with the Jakarta Messaging journal, you can read the current configuration for both by using the single management CLI command below. The output is also included to highlight default configuration.

```
/subsystem=messaging-activemq/server=default/path=bindings-directory:read-resource

{
    "outcome" => "success",
    "result" => {
        "path" => "activemq/bindings",
        "relative-to" => "jboss.server.data.dir"
    }
}
```

Note that by default the **path** to the journal is **activemq/bindings**. You can change the location for **path** by using the following management CLI command.

```
/subsystem=messaging-activemq/server=default/path=bindings-directory:write-attribute(name=path,value=PATH_LOCATION)
```

Also note the **relative-to** attribute in the output above. When **relative-to** is used, the value of the **path** attribute is treated as relative to the file path specified by **relative-to**. By default this value is the JBoss EAP property **jboss.server.data.dir**. For standalone servers, **jboss.server.data.dir** is located at ***EAP_HOME*/standalone/data**. For domains, each server will have its own **serverX/data/activemq** directory located under ***EAP_HOME*/domain/servers**. You can change the value of **relative-to** using the following management CLI command.

```
/subsystem=messaging-activemq/server=default/path=bindings-directory:write-attribute(name=relative-to,value=RELATIVE_LOCATION)
```

By default, JBoss EAP is configured to automatically create the bindings directory if it does not exist. Use the following management CLI command to toggle this behavior.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=create-bindings-dir,value=TRUE/FALSE)
```

Setting **value** to **true** will enable automatic directory creation. Setting **value** to **false** will disable it.

10.2.4. Configuring the Message Journal Location

You can read the location information for the message journal by using the management CLI command below. The output is also included to highlight default configuration.

```
/subsystem=messaging-activemq/server=default/path=journal-directory:read-resource
{
    "outcome" => "success",
    "result" => {
        "path" => "activemq/journal",
        "relative-to" => "jboss.server.data.dir"
    }
}
```

Note that by default the **path** to the journal is **activemq/journal**. You can change the location for **path** by using the following management CLI command.

```
/subsystem=messaging-activemq/server=default/path=journal-directory:write-attribute(name=path,value=PATH_LOCATION)
```

NOTE

For the best performance, Red Hat recommends that the journal be located on its own physical volume in order to minimize disk head movement. If the journal is on a volume which is shared with other processes which might be writing other files, such as a bindings journal, database, or transaction coordinator, then the disk head may well be moving rapidly between these files as it writes them, thus drastically reducing performance.

Also note the **relative-to** attribute in the output above. When **relative-to** is used, the value of the **path** attribute is treated as relative to the file path specified by **relative-to**. By default this value is the JBoss EAP property **jboss.server.data.dir**. For standalone servers, **jboss.server.data.dir** is located at

***EAP_HOME*/standalone/data**. For domains, each server will have its own **serverX/data/activemq** directory located under ***EAP_HOME*/domain/servers**. You can change the value of **relative-to** using the following management CLI command.

```
/subsystem=messaging-activemq/server=default/path=journal-directory:write-attribute(name=relative-to,value=RELATIVE_LOCATION)
```

By default, JBoss EAP is configured to automatically create the journal directory if it does not exist. Use the following management CLI command to toggle this behavior.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=create-journal-dir,value=TRUE/FALSE)
```

Setting **value** to **true** will enable automatic directory creation. Setting **value** to **false** will disable it.

10.2.5. Configuring Message Journal Attributes

The attributes listed below are all child properties of the messaging server. Therefore, the command syntax for getting and setting their values using the management CLI is the same for each.

To read the current value of a given attribute, the syntax is as follows:

```
/subsystem=messaging-activemq/server=default:read-attribute(name=ATTRIBUTE_NAME)
```

The syntax for writing an attribute's value follows a corresponding pattern.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=ATTRIBUTE_NAME,value=NEW_VALUE)
```

- **create-journal-dir**
  If this is set to **true**, the journal directory will be automatically created at the location specified in **journal-directory** if it does not already exist. The default value is **true**.

- **journal-file-open-timeout**
  This attribute modifies the timeout value for opening a journal file. The default value is **5** seconds.

- **journal-buffer-timeout**
  Instead of flushing on every write that requires a flush, we maintain an internal buffer, and flush the entire buffer either when it is full, or when a timeout expires, whichever is sooner. This is used for both NIO and ASYNCIO and allows the system to scale better with many concurrent writes that require flushing.

  This parameter controls the timeout at which the buffer will be flushed if it has not filled already. ASYNCIO can typically cope with a higher flush rate than NIO, so the system maintains different defaults for both NIO and ASYNCIO. The default for NIO is **3333333** nanoseconds, or 300 times per second. The default for ASYNCIO is **500000** nanoseconds, or 2000 times per second.

  NOTE

  By increasing the timeout, you may be able to increase system throughput at the expense of latency, the default parameters are chosen to give a reasonable balance between throughput and latency.

- **journal-buffer-size**
  The size, in bytes, of the timed buffer on ASYNCIO. Both **journal-buffer-size** and **journal-file-size** must be set larger than **min-large-message-size**. Otherwise, messages will not be written to the journal. See Configuring Large Messages for more information.

- **journal-compact-min-files**
  The minimal number of files before we can consider compacting the journal. The compacting algorithm won't start until you have at least **journal-compact-min-files**.

  Setting this to **0** will disable the feature to compact completely. This could be dangerous though as the journal could grow indefinitely. Use it wisely!

  The default for this parameter is **10**

- **journal-compact-percentage**
  The threshold to start compacting. When less than this percentage is considered live data, we start compacting. Note also that compacting will not kick in until you have at least **journal-compact-min-files** data files on the journal

  The default for this parameter is **30**.

- **journal-file-size**
  The size of each journal file, in bytes. The default value for this is **10485760** bytes, or 10MB. Both **journal-file-size** and **journal-buffer-size** must be set larger than **min-large-message-size**. Otherwise, messages will not be written to the journal. See Configuring Large Messages for more information.

- **journal-max-io**
  Write requests are queued up before being submitted to the system for execution. This parameter controls the maximum number of write requests that can be in the IO queue at any one time. If the queue becomes full then writes will block until space is freed up.

  The system maintains different defaults for this parameter depending on whether it's NIO or ASYNCIO. The default for NIO is **1**, and the default for ASYNCIO is **500**.

  There is a limit and the total max ASYNCIO cannot be higher than what is configured at the OS level, found at /proc/sys/fs/aio-max-nr, usually **65536**.

- **journal-min-files**
  The minimum number of files the journal will maintain. When JBoss EAP starts and there is no initial message data, JBoss EAP will pre-create **journal-min-files** number of files. The default is **2**.

  Creating journal files and filling them with padding is a fairly expensive operation and we want to minimize doing this at run-time as files get filled. By pre-creating files, as one is filled the journal can immediately resume with the next one without pausing to create it.

  Depending on how much data you expect your queues to contain at steady state you should tune this number of files to match that total amount of data.

- **journal-pool-files**
  The number of journal files that can be reused. ActiveMQ will create as many files as needed however when reclaiming files it will shrink back to the value. The default is **-1**, which means no limit.

- **journal-sync-transactional**

If this is set to true then JBoss EAP will make sure all transaction data is flushed to disk on transaction boundaries, such as a commit, prepare, or rollback. The default value is **true**.

- **journal-sync-non-transactional**
  If this is set to true then JBoss EAP will make sure non transactional message data, such as sends and acknowledgements, are flushed to disk each time. The default value is **true**.

- **journal-type**
  Valid values are **NIO** or **ASYNCIO**.

  Choosing **NIO** tells JBoss EAP to use a Java NIO journal. **ASYNCIO** tells it to use a Linux asynchronous IO journal. If you choose **ASYNCIO** but are not running Linux, or you do not have libaio installed, JBoss EAP will use a Java NIO journal.

10.2.6. Note on Disabling Disk Write Cache

This happens irrespective of whether you have executed a **fsync()** from the operating system or correctly synced data from inside a Java program!

By default many systems ship with disk write cache enabled. This means that even after syncing from the operating system there is no guarantee the data has actually made it to disk, so if a failure occurs, critical data can be lost.

Some more expensive disks have non volatile or battery backed write caches which will not necessarily lose data on event of failure, but you need to test them!

If your disk does not have an expensive non volatile or battery backed cache and it's not part of some kind of redundant array, for example RAID, and you value your data integrity you need to make sure disk write cache is disabled.

Be aware that disabling disk write cache can give you a nasty shock performance wise. If you've been used to using disks with write cache enabled in their default setting, unaware that your data integrity could be compromised, then disabling it will give you an idea of how fast your disk can perform when acting really reliably.

On Linux you can inspect or change your disk's write cache settings using the tools **hdparm** for IDE disks, or **sdparm** or **sginfo** for SDSI/SATA disks.

On Windows, you can check and change the setting by right clicking on the disk and then clicking **properties**.

10.2.7. Installing libaio

The Java NIO journal is highly performant, but if you are running JBoss EAP messaging using Linux Kernel 2.6 or later, Red Hat highly recommends that you use the ASYNCIO journal for the very best persistence performance.

NOTE

JBoss EAP supports ASYNCIO only when installed on versions 6, 7 or 8 of Red Hat Enterprise Linux and only when using the ext4, xfs, gfs2 or nfs4 file systems. It is not possible to use the ASYNCIO journal under other operating systems or earlier versions of the Linux kernel.

You will need **libaio** installed to use the ASYNCIO journal. To install, use the following command:

- For Red Hat Enterprise Linux 6 and 7:

  ```
  yum install libaio
  ```

- For Red Hat Enterprise Linux 8:

  ```
  dnf install libaio
  ```

> **WARNING**
>
> Do not place your messaging journals on a tmpfs file system, which is used for the **/tmp** directory for example. JBoss EAP will fail to start if the ASYNCIO journal is using tmpfs.

10.2.8. Configuring the NFS Shared Store for Messaging

When using dedicated, shared store, high availability for data replication, you must configure both the live server and the backup server to use a shared directory on the NFS client. If you configure one server to use a shared directory on the NFS server and the other server to use a shared directory on the NFS client, the backup server cannot recognize when the live server starts or is running. So to work properly, both servers must specify a shared directory on the NFS client.

You must also configure the following options for the NFS client mount:

- **sync**: This option specifies that all changes are immediately flushed to disk.
- **intr**: This option allows NFS requests to be interrupted if the server goes down or cannot be reached.
- **noac**: This option disables attribute caching and is needed to achieve attribute cache coherence among multiple clients.
- **soft**: This option specifies that if the host serving the exported file system is unavailable, the error should be reported rather than waiting for the server to come back online.
- **lookupcache=none**: This option disables lookup caching.
- **timeo=*n***: The time in deciseconds (tenths of a second) the NFS client waits for a response before it retries an NFS request. For NFS over TCP, the default **timeo** value is **600** (60 seconds). For NFS over UDP, the client uses an adaptive algorithm to estimate an appropriate timeout value for frequently used request types, such as read and write requests.
- **retrans=*n***: The number of times the NFS client retries a request before it attempts further recovery action. If the **retrans** option is not specified, the NFS client tries each request three times.

IMPORTANT

It is important to use reasonable values when you configure the **timeo** and **retrans** options. A default **timeo** wait time of **600** deciseconds (60 seconds) combined with a **retrans** value of **5** retries can result in a five minute wait for ActiveMQ Artemis to detect an NFS disconnection.

See the Shared Store section in this guide for more information about how to use a shared file system for high availability.

10.3. MESSAGING JOURNAL PERSISTENCE USING A JDBC DATABASE

To use JDBC to persist messages and binding data to a database instead of using the default file-based journal, you must configure JBoss EAP 7 messaging.

To do this, you must first configure the **datasource** element in the **datasources** subsystem, and then define a **journal-datasource** attribute on the **server** element in the **messaging-activemq** subsystem to use that datasource. The presence of the **journal-datasource** attribute notifies the messaging subsystem to persist the journal entries to the database instead of the file-based journal. The **journal-database** attribute on the **server** resource in the **messaging-activemq** subsystem defines the SQL dialect that is used to communicate with the database. This attribute is configured automatically using the datasource metadata.

When persisting messages to a file-based journal, the large message size is limited only by the size of the disk. However, when persisting messages to a database, the large message size is limited to the maximum size of the **BLOB** data type for that database.

IMPORTANT

JBoss EAP 7.4 currently supports only the Oracle 12c and IBM DB2 Enterprise databases.

10.3.1. Considerations to configure a database persistent store

For improved reliability, JBoss EAP makes messaging calls through a connection pool, which provides a set of open connections to a specified database that can be shared among multiple applications. This means if JBoss EAP drops a connection, another connection in the pool replaces that failed connection to avoid failure.

NOTE

Previous versions of JBoss EAP support only one connection from a pool.

When you configure a database persistent store or pool in the datasources subsystem, consider the following points:

- Set the value of the **min-pool-size** attribute to at least 4 to have a connection dedicated to each of the following usage:
  - One for the binding
  - One for the messages journal
  - One for the lease lock, if using High Availability (HA)

   - One for the node manager shared state, if using HA

- Set the value of the **max-pool-size** attribute based on the number of concurrent threads that perform paging or large message streaming operations. No rules are defined for configuring the **max-pool-size** attribute because the relation between the number of threads and the number of connections is not one-to-one.

The number of connections depends on the number of threads that process paging and large messages operations and the attribute **blocking-timeout-wait-millis** that defines the time involved in waiting to get a connection.

New large messages or paging operations occur in a dedicated thread and need a connection. Those dedicated threads are enqueued until a connection is ready or the time to obtain the connection runs out, which results in a failure.

You can customize the pool configuration according to your needs and test the configured pool in your environment.

10.3.2. Configuring a messaging journal JDBC persistence store

Follow these steps to configure JBoss EAP 7 messaging to use JDBC to persist messages and binding data to a database:

1. Configure a datasource in the datasources subsystem for use by the **messaging-activemq** subsystem. For information about how to create and configure a datasource, see Datasource Management in the JBoss EAP *Configuration Guide*.

2. Configure the **messaging-activemq** subsystem to use the new datasource.

   ```
   /subsystem=messaging-activemq/server=default:write-attribute(name=journal-datasource,value="MessagingOracle12cDS")
   ```

   This creates the following configuration in the **messaging-activemq** subsystem of the server configuration file:

   ```
   <server name="default">
     <journal datasource="MessagingOracle12cDS"/>
     ...
   </server>
   ```

JBoss EAP messaging is now configured to use the database to store messaging data.

10.3.3. Configuring messaging journal table names

JBoss EAP 7 messaging uses a separate JDBC table to store binding information, messages, large messages, and paging information. The names of these tables can be configured using the **journal-bindings-table**, **journal-jms-bindings-table**, **journal-messages-table**, **journal-large-messages-table**, and **journal-page-store-table** attributes on the **server** resource in the **messaging-activemq** subsystem of the server configuration file.

The following is a list of table name restrictions:

- JBoss EAP 7 messaging generates identifiers for paging tables using pattern *TABLE_NAME* + *GENERATED_ID*, where the *GENERATED_ID* can be up to 20 characters long. Because the maximum table name length in Oracle Database 12c is 30 characters, you must limit the table name to 10 characters. Otherwise, you might see the error **ORA-00972: identifier is too long** and paging will no longer work.

- Table names that do not follow Schema Object Naming Rules for Oracle Database 12c must be enclosed within double quotes. Quoted identifiers can begin with any character and can contain any characters and punctuation marks as well as spaces. However, neither quoted nor nonquoted identifiers can contain double quotation marks or the null character (\0). It is important to note that quoted identifiers are case sensitive.

- If multiple JBoss EAP server instances use the same database to persist messages and binding data, the table names must be unique for each server instance. Multiple JBoss EAP servers cannot access the same tables.

The following is an example of the management CLI command that configures the **journal-page-store-table** name using a quoted identifier:

```
/subsystem=messaging-activemq/server=default:write-attribute(name=journal-page-store-
table,value="\"PAGE_DATA\"")
```

This creates the following configuration in the **messaging-activemq** subsystem of the server configuration file:

```
<server name="default">
  <journal datasource="MessagingOracle12cDS" journal-page-store-
table="&quot;PAGED_DATA&quot;"/>
  ...
</server>
```

10.3.4. Configuring messaging journals in a managed domain

As mentioned in Configuring messaging journal table names, multiple JBoss EAP servers cannot access the same database tables when using JDBC to persist messages and binding data to a database. In a managed domain, all JBoss EAP server instances in a server group share the same profile configuration, so you must use expressions to configure the messaging journal names or datasources.

If all servers are configured to use the same database to store messaging data, the table names must be unique for each server instance. The following is an example of a management CLI command that creates a unique **journal-page-store-table** table name for each server in a server group by using an expression that includes the unique node identifier in the name.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=journal-page-store-
table,value="${env.NODE_ID}_page_store")
```

If each server instance accesses a different database, you can use expressions to allow the messaging configuration for each server to connect to a different datasource. The following management CLI command uses the **DB_CONNECTION_URL** environment variable in the **connection-url** to connect to a different datasource.

```
data-source add --name=messaging-journal --jndi-name=java:jboss/datasources/messaging-journal -
-driver-name=oracle12c  --connection-url=${env.DB_CONNECTION_URL}
```

10.3.5. Configuring the messaging journal network timeout

You can configure the maximum amount of time, in milliseconds, that the JDBC connection will wait for the database to reply a request. This is useful in the event that the network goes down or a connection between JBoss EAP messaging and the database is closed for any reason. When this occurs, clients are blocked until the timeout occurs.

You configure the timeout by updating the **journal-jdbc-network-timeout** attribute. The default value is **20000** milliseconds, or **20** seconds.

The following is an example of the management CLI command that sets the **journal-jdbc-network-timeout** attribute value to **10000** milliseconds, or **10** seconds:

```
/subsystem=messaging-activemq/server=default:write-attribute(name=journal-jdbc-network-timeout,value=10000)
```

10.3.6. Configuring HA for Messaging JDBC Persistence Store

The JBoss EAP **messaging-activemq** subsystem activates the JDBC HA shared store functionality when the broker is configured with a database store type. The broker then uses a shared database table to ensure that the live and backup servers coordinate actions over a shared JDBC journal store.

You can configure HA for JDBC persistence store using the following attributes:

- **journal-node-manager-store-table**: Name of the JDBC database table to store the node manager.
- **journal-jdbc-lock-expiration**: The time a JDBC lock is considered valid without keeping it alive. You specify this attribute value in seconds. The default value is **20** seconds.
- **journal-jdbc-lock-renew-period**: The period of the keep alive service of a JDBC lock. You specify this attribute value in seconds. The default value is **2** seconds.

The default values are taken into account based on the value of the server’s **ha-policy** and **journal-datasource** attributes.

For backward compatibility, you can also specify their values using the respective Artemis-specific system properties:

- **brokerconfig.storeConfiguration.nodeManagerStoreTableName**
- **brokerconfig.storeConfiguration.jdbcLockExpirationMillis**
- **brokerconfig.storeConfiguration.jdbcLockRenewPeriodMillis**

When configured, these Artemis-specific system properties have precedence over the corresponding attribute’s default value.

10.4. MANAGING MESSAGING JOURNAL PREPARED TRANSACTIONS

You can manage messaging journal prepared transactions using the following management CLI commands.

- Commit a prepared transaction:

  ```
  /subsystem=messaging-activemq/server=default:commit-prepared-transaction(transaction-as-base-64=XID)
  ```

- Roll back a prepared transaction:

  ```
  /subsystem=messaging-activemq/server=default:rollback-prepared-transaction(transaction-as-base-64=XID)
  ```

- Show the details of all prepared transactions:

  ```
  /subsystem=messaging-activemq/server=default:list-prepared-transactions
  ```

NOTE

You can also show the prepared transaction details in HTML format using the **list-prepared-transaction-details-as-html** operation, or in JSON format using the **list-prepared-transaction-details-as-json** operation.

10.5. CONFIGURING JBOSS EAP MESSAGING FOR ZERO PERSISTENCE

In some situations, zero persistence is required for a messaging system. Zero persistence means that no bindings data, message data, large message data, duplicate id caches, or paging data should be persisted.

To configure the **messaging-activemq** subsystem to perform zero persistence, set the **persistence-enabled** parameter to **false**.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=persistence-enabled,value=false)
```

IMPORTANT

Be aware that if persistence is disabled, but paging is enabled, page files continue to be stored in the location specified by the **paging-directory** element. Paging is enabled when the **address-full-policy** attribute is set to **PAGE**. If full zero persistence is required, be sure to configure the **address-full-policy** attribute of the **address-setting** element to use **BLOCK**, **DROP** or **FAIL**.

10.6. IMPORTING AND EXPORTING JOURNAL DATA

See the JBoss EAP 7 Migration Guide for information on importing and exporting journal data.

CHAPTER 11. CONFIGURING PAGING

11.1. ABOUT PAGING

JBoss EAP messaging supports many message queues with each queue containing millions of messages. The JBoss EAP messaging server runs with limited memory thereby making it difficult to store all message queues in memory at one time.

Paging is a mechanism used by the JBoss EAP messaging server to transparently page messages in and out of memory on an as-needed basis in order to accommodate large message queues in a limited memory.

JBoss EAP messaging starts paging messages to disk, when the size of messages in memory for a particular address exceeds the maximum configured message size.

NOTE

JBoss EAP messaging paging is enabled by default.

11.2. PAGE FILES

There is an individual folder for each address on the file system which stores messages in multiple files. These files which store the messages are called page files. Each file contains messages up to the maximum configured message size set by the **page-size-bytes** attribute.

The system navigates the page files as needed and removes the page files as soon as all messages in the page were received by client.

WARNING

For performance reasons, JBoss EAP messaging does not scan paged messages. Therefore, you should disable paging on a queue that is configured to group messages or to provide a last value. Also, message prioritization and message selectors will not behave as expected for queues that have paging enabled. You must disable paging for these features to work as expected

For example, if a consumer has a message selector to read messages from a queue, only the messages in memory that match the selector are delivered to the consumer. When the consumer acknowledges delivery of these messages, new messages are de-paged and loaded into memory. There may be messages that match a consumer’s selector on disk in page files but JBoss EAP messaging does not load them into memory until another consumer reads the messages in memory and provides free space. If the free space is not available, the consumer employing a selector may not receive any new messages.

11.3. CONFIGURING THE PAGING DIRECTORY

You can read the configuration for the paging directory by using the management CLI command below. In this example, the output displays the default configuration.

```
/subsystem=messaging-activemq/server=default/path=paging-directory:read-resource
{
    "outcome" => "success",
    "result" => {
        "path" => "activemq/paging",
        "relative-to" => "jboss.server.data.dir"
    }
}
```

The **paging-directory** configuration element specifies the location on the file system to store the page files. JBoss EAP creates one folder for each paging address in this paging directory and the page files are stored within these folders. By default, this path is **activemq/paging/**. You can change the path location by using the following management CLI command.

```
/subsystem=messaging-activemq/server=default/path=paging-directory:write-attribute(name=path,value=PATH_LOCATION)
```

Also note the **relative-to** attribute in the example output above. When **relative-to** is specified, the value of the **path** attribute is treated as relative to the file path specified by the **relative-to** attribute. By default, this value is the JBoss EAP **jboss.server.data.dir** property. For standalone servers, **jboss.server.data.dir** is located at ***EAP_HOME*/standalone/data/**. For managed domains, each server will have its own **serverX/data/activemq/** directory located under ***EAP_HOME*/domain/servers/**. You can change the value of **relative-to** using the following management CLI command.

```
/subsystem=messaging-activemq/server=default/path=paging-directory:write-attribute(name=relative-to,value=RELATIVE_LOCATION)
```

11.4. CONFIGURING PAGING MODE

When messages delivered to an address exceed the configured size, that address goes into *paging mode*.

NOTE

Paging is done individually per address. If you configure a **max-size-bytes** for an address, it means each matching address will have a maximum size that you specified. However it does not mean that the total overall size of all matching addresses is limited to **max-size-bytes**.

Even with **page** mode, the server may crash due to an out-of-memory error. JBoss EAP messaging keeps a reference to each page file on the disk. In a situation with millions of page files, JBoss EAP messaging can face memory exhaustion. To minimize this risk, it is important to set the attribute **page-size-bytes** to a suitable value. You must configure the memory for your JBoss EAP messaging server to be greater than two times the number of destinations times the **max-size-bytes**, otherwise an out-of-memory error can occur.

You can read the current maximum size in bytes (**max-size-bytes**) for an address by using the following management CLI command.

```
/subsystem=messaging-activemq/server=default/address-setting=ADDRESS_SETTING:read-attribute(name=max-size-bytes)
```

You can configure the maximum size in bytes (**max-size-bytes**) for an address by using the following management CLI command.

```
/subsystem=messaging-activemq/server=default/address-setting=ADDRESS_SETTING:write-attribute(name=max-size-bytes,value=MAX_SIZE)
```

Use a similar syntax when reading or writing the values for the other paging-related attributes of an address setting. The table below lists each attribute, along with a description and a default value.

The following table describes the parameters on the address settings:

Table 11.1. Paging Configuration for Address Settings

Element	Description
address-full-policy	This value of this attribute is used for paging decisions. The valid valid values are listed below. **PAGE** Enables paging and page messages beyond the set limit to disk. **DROP** Silently drops messages that exceed the set limit. **FAIL** Drops messages and sends an exception to client message producers. **BLOCK** Blocks client message producers when they send messages beyond the set limit. The default is **PAGE**.
max-size-bytes	This is used to specify the maximum memory size the address can have before entering into paging mode. The default is **10485760**.
page-max-cache-size	The system will keep page files up to **page-max-cache-size** in memory to optimize Input/Output during paging navigation. The default is **5**.
page-size-bytes	This is used to specify the size of each page file used on the paging system. The default is **2097152**.

IMPORTANT

By default, all addresses are configured to page messages after an address reaches **max-size-bytes**. If you do not want to page messages when the maximum size is reached, you can configure an address to drop messages, drop messages with an exception on client side, or block producers from sending further messages by setting the **address-full-policy** to **DROP**, **FAIL** and **BLOCK** respectively.

Be aware that if you change the **address-full-policy** from **PAGE** to **BLOCK** after any destination has started to page messages, consumers will no longer be able to consume paged messages.

Addresses with Multiple Queues

When a message is routed to an address that has multiple queues bound to it, there is only a single copy of the message in memory. Each queue only handles a reference to this original copy of the message, so the memory is freed up only when all the queues referencing the original message, have delivered the message.

NOTE

A single lazy queue/subscription can reduce the Input/Output performance of the entire address as all the queues will have messages being sent through an extra storage on the paging system.

CHAPTER 12. WORKING WITH LARGE MESSAGES

JBoss EAP messaging supports large messages, even when the client or server has limited amounts of memory. Large messages can be streamed as they are, or they can be compressed further for more efficient transferral. A user can send a large message by setting an **InputStream** in the body of the message. When the message is sent, JBoss EAP messaging reads this **InputStream** and transmits data to the server in fragments.

Neither the client nor the server stores the complete body of a large message in memory. The consumer initially receives a large message with an empty body and thereafter sets an **OutputStream** on the message to stream it in fragments to a disk file.

WARNING

When processing large messages, the server does not handle message properties in the same way as the message body. For example a message with a property set to a string that is bigger than **journal-buffer-size** cannot be processed by the server because it overfills the journal buffer.

12.1. STREAMING LARGE MESSAGES

If you send large messages the standard way, the heap size required to send them can be four or more times the size of the message, meaning a 1 GB message can require 4 GB in heap memory. For this reason, JBoss EAP messaging supports setting the body of messages using the **java.io.InputStream** and **java.io.OutputStream** classes, which require much less memory. Input streams are used directly for sending messages and output streams are used for receiving messages.

When receiving messages, there are two ways to deal with the output stream:

- You can block while the output stream is recovered using the **ClientMessage.saveToOutputStream(OutputStream out)** method.
- You can use the **ClientMessage.setOutputstream(OutputStream out)** method to asynchronously write the message to the stream. This method requires that the consumer be kept alive until the message has been fully received.

You can use any kind of stream you like, for example files, JDBC Blobs, or SocketInputStream, as long as it implements **java.io.InputStream** for sending messages and **java.io.OutputStream** for receiving messages.

Streaming Large Messages Using the Core API

The following table shows the methods available on the **ClientMessage** class that are available through Jakarta Messaging by using object properties.

ClientMessage Method	Description	Jakarta Messaging Equivalent Property
setBodyInputStream(InputStream)	Set the **InputStream** used to read a message body when it is sent.	**JMS_AMQ_InputStream**

ClientMessage Method	Description	Jakarta Messaging Equivalent Property
setOutputStream(OutputStream)	Set the **OutputStream** that will receive the body of a message. This method does not block.	**JMS_AMQ_OutputStream**
saveOutputStream(OutputStream)	Save the body of the message to the **OutputStream**. It will block until the entire content is transferred to the **OutputStream**.	**JMS_AMQ_SaveStream**

The following code example sets the output stream when receiving a core message.

```
ClientMessage firstMessage = consumer.receive(...);

// Block until the stream is transferred
firstMessage.saveOutputStream(firstOutputStream);

ClientMessage secondMessage = consumer.receive(...);

// Do not wait for the transfer to finish
secondMessage.setOutputStream(secondOutputStream);
```

The following code example sets the input stream when sending a core message:

```
ClientMessage clientMessage = session.createMessage();
clientMessage.setInputStream(dataInputStream);
```

NOTE

For messages larger than 2GiB, you must use the **_AMQ_LARGE_SIZE** message property. This is because the **getBodySize()** method will return an invalid value because it is limited to the maximum integer value.

Streaming Large Messages Over Jakarta Messaging

When using Jakarta Messaging, JBoss EAP messaging maps the core API streaming methods by setting object properties. You use the **Message.setObjectProperty(String name, Object value)** method to set the input and output streams.

The **InputStream** is set using the **JMS_AMQ_InputStream** property on messages being sent.

```
BytesMessage bytesMessage = session.createBytesMessage();
FileInputStream fileInputStream = new FileInputStream(fileInput);
BufferedInputStream bufferedInput = new BufferedInputStream(fileInputStream);
bytesMessage.setObjectProperty("JMS_AMQ_InputStream", bufferedInput);
someProducer.send(bytesMessage);
```

The **OutputStream** is set using the **JMS_AMQ_SaveStream** property on messages being received in a blocking manner.

```
BytesMessage messageReceived = (BytesMessage) messageConsumer.receive(120000);
File outputFile = new File("huge_message_received.dat");
FileOutputStream fileOutputStream = new FileOutputStream(outputFile);
BufferedOutputStream bufferedOutput = new BufferedOutputStream(fileOutputStream);

// This will block until the entire content is saved on disk
messageReceived.setObjectProperty("JMS_AMQ_SaveStream", bufferedOutput);
```

The **OutputStream** can also be set in a non-blocking manner by using the **JMS_AMQ_OutputStream** property.

```
// This does not wait for the stream to finish. You must keep the consumer active.
messageReceived.setObjectProperty("JMS_AMQ_OutputStream", bufferedOutput);
```

NOTE

When streaming large messages using Jakarta Messaging, only **StreamMessage** and **BytesMessage** objects are supported.

12.2. CONFIGURING LARGE MESSAGES

12.2.1. Configure Large Message Location

You can read the configuration for the large messages directory by using the management CLI command below. The output is also included to highlight default configuration.

```
/subsystem=messaging-activemq/server=default/path=large-messages-directory:read-resource
{
    "outcome" => "success",
    "result" => {
        "path" => "activemq/largemessages",
        "relative-to" => "jboss.server.data.dir"
    }
}
```

IMPORTANT

To achieve the best performance, it is recommended to store the large messages directory on a different physical volume from the message journal or the paging directory.

The **large-messages-directory** configuration element is used to specify a location on the filesystem to store the large messages. Note that by default the path is **activemq/largemessages**. You can change the location for path by using the following management CLI command.

```
/subsystem=messaging-activemq/server=default/path=large-messages-directory:write-
attribute(name=path,value=PATH_LOCATION)
```

Also note the **relative-to** attribute in the output above. When **relative-to** is used, the value of the path attribute is treated as relative to the file path specified by **relative-to**. By default this value is the JBoss EAP property **jboss.server.data.dir**. For standalone servers, **jboss.server.data.dir** is located at

***EAP_HOME*/standalone/data**. For domains, each server will have its own **serverX/data/activemq** directory located under ***EAP_HOME*/domain/servers**. You can change the value of **relative-to** using the following management CLI command.

```
/subsystem=messaging-activemq/server=default/path=large-messages-directory:write-attribute(name=relative-to,value=RELATIVE_LOCATION)
```

Configuring Large Message Size

Use the management CLI to view the current configuration for large messages. Note that the this configuration is part of a **connection-factory** element. For example, to read the current configuration for the default **RemoteConnectionFactory** that is included, use the following command:

```
/subsystem=messaging-activemq/server=default/connection-factory=RemoteConnectionFactory:read-attribute(name=min-large-message-size)
```

Set the attribute using a similar syntax.

```
/subsystem=messaging-activemq/server=default/connection-factory=RemoteConnectionFactory:write-attribute(name=min-large-message-size,value=NEW_MIN_SIZE)
```

NOTE

The value of the attribute **min-large-message-size** should be in bytes.

Configuring Large Message Compression

You can choose to compress large messages for fast and efficient transfer. All compression/decompression operations are handled on the client side. If the compressed message is smaller than **min-large-message size**, it is sent to the server as a regular message. Compress large messages by setting the boolean property **compress-large-messages** to **true** using the management CLI.

```
/subsystem=messaging-activemq/server=default/connection-factory=RemoteConnectionFactory:write-attribute(name=compress-large-messages,value=true)
```

12.2.2. Configuring Large Message Size Using the Core API

If you are using the core API on the client side, you need to use the **setMinLargeMessageSize** method to specify the minimum size of large messages. The minimum size of large messages (**min-large-message-size**) is set to 100KB by default.

```
ServerLocator locator = ActiveMQClient.createServerLocatorWithoutHA(new
TransportConfiguration(InVMConnectorFactory.class.getName()))

locator.setMinLargeMessageSize(25 * 1024);

ClientSessionFactory factory = ActiveMQClient.createClientSessionFactory();
```

CHAPTER 13. SCHEDULING MESSAGES

You can specify a time in the future, at the earliest, for a message to be delivered. This can be done by setting the **_AMQ_SCHED_DELIVERY** scheduled delivery property before the message is sent.

The specified value must be a positive **long** that corresponds to the time in milliseconds for the message to be delivered. Below is an example of sending a scheduled message using the Jakarta Messaging API.

```
// Create a message to be delivered in 5 seconds
TextMessage message = session.createTextMessage("This is a scheduled message message that
will be delivered in 5 sec.");
message.setLongProperty("_AMQ_SCHED_DELIVERY", System.currentTimeMillis() + 5000);
producer.send(message);

...

// The message will not be received immediately, but 5 seconds later
TextMessage messageReceived = (TextMessage) consumer.receive();
```

Scheduled messages can also be sent using the core API by setting the **_AMQ_SCHED_DELIVERY** property before sending the message.

CHAPTER 14. TEMPORARY QUEUES AND RUNTIME QUEUES

When designing a request-reply pattern where a client sends a request and waits for a reply, you must consider whether each runtime instance of the client requires a dedicated queue for its replies, or whether the runtime instances can access a shared queue, selecting their specific reply messages based on an appropriate attribute.

If multiple queues are required, then clients need the ability to create a queue dynamically. Jakarta Messaging provides this facility using the concept of temporary queues. A **TemporaryQueue** is created on request by the **Session**. It exists for the life of the **Connection**, for example until the connection is closed, or until the temporary queue is deleted. This means that although the temporary queue is created by a specific session, it can be reused by any other sessions created from the same connection.

The trade-off between using a shared queue and individual temporary queues for replies is influenced by the potential number of active client instances. With a shared-queue approach, at some provider-specific threshold, contention for access to the queue can become a concern. This has to be contrasted against the additional overhead associated with the provider creating queue storage at runtime and the impact on machine memory of hosting a potentially large number of temporary queues.

The following example creates a temporary queue and consumer for each client on startup. It sets the **JMSReplyTo** property on each message to the temporary queue, and then sets a correlation ID on each message to correlate request messages to response messages. This avoids the overhead of creating and closing a consumer for each request, which is expensive. The same producer and consumer can be shared or pooled across many threads. Any messages that have been received, but not yet acknowledged when the session terminates, are retained and redelivered when a consumer next accesses the queue.

Example: Temporary Queue Code

```
...
// Create a temporary queue, one per client
Destination temporaryQueue = session.createTemporaryQueue();
MessageConsumer responseConsumer = session.createConsumer(temporaryQueue);

// This class handles messages to the temporary queue
responseConsumer.setMessageListener(this);

// Create the message to send
TextMessage textMessage = session.createTextMessage();
textMessage.setText("My new message!");

// Set the reply to field and correlation ID
textMessage.setJMSReplyTo(temporaryQueue);
textMessage.setJMSCorrelationID(myCorrelationID);

producer.send(textMessage);
...
```

In a similar manner, temporary topics are created using the **Session.createTemporaryTopic()** method.

CHAPTER 15. FILTER EXPRESSIONS AND MESSAGE SELECTORS

The **messaging-activemq** subsystem in JBoss EAP provides a powerful filter language based on a subset of the SQL 92 expression syntax.

It is the same as the syntax used for Jakarta Messaging selectors, but the predefined identifiers are different. For documentation on Jakarta Messaging selector syntax, refer to the javax.jms.Message interface Javadoc.

The **filter** attribute can be found in several places within the configuration.

- Predefined Queues. When pre-defining a queue, a filter expression can be defined for it. Only messages that match the filter expression will enter the queue. The configuration snippet below shows a queue definition that includes a filter:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  ...
  <queue
    name="myQueue"
    filter="FILTER_EXPRESSION"
    ...
  />
  ...
</subsystem>
```

  To create queue with a selector in the management CLI you would use something like following command:

```
jms-queue add --queue-address=QUEUE_ADDRESS --selector=FILTER_EXPRESSION
```

- Core bridges can be defined with an optional filter expression, only matching messages will be bridged. Below is a snippet from a sample configuration file where the **messaging-activemq** subsystem includes a bridge with a filter.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  ...
  <bridge
    name="myBridge"
    filter="FILTER_EXPRESSION"
    ...
  />
  ...
</subsystem>
```

- Diverts can be defined with an optional filter expression, only matching messages will be diverted. See Diverts for more information. The example snippet below shows a Divert using a filter:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  ...
  <divert
    name="myDivert"
    filter="FILTER_EXPRESSION"
    ...
  />
  ...
</subsystem>
```

There are some differences between Jakarta Messaging selector expressions and JBoss EAP messaging core filter expressions. Whereas Jakarta Messaging selector expressions operate on a Jakarta Messaging message, JBoss EAP messaging core filter expressions operate on a core message.

The following identifiers can be used in core filter expressions to refer to the attributes of a core message:

- **AMQPriority**. To refer to the priority of a message. Message priorities are integers with valid values from **0 - 9**. **0** is the lowest priority and **9** is the highest. For example, **AMQPriority** = 3 AND animal = 'aardvark'.
- **AMQExpiration**. To refer to the expiration time of a message. The value is a long integer.
- **AMQDurable**. To refer to whether a message is durable or not. The value is a string with valid values: **DURABLE** or **NON_DURABLE**.
- **AMQTimestamp**. The timestamp of when the message was created. The value is a long integer.
- **AMQSize**. The size of a message in bytes. The value is an integer.

Any other identifiers used in core filter expressions will be assumed to be properties of the message.

CHAPTER 16. CONFIGURING MESSAGE EXPIRY

Sent messages can be set to expire on the server if they are not delivered to a consumer after a specified amount of time. These expired messages can later be consumed for further inspection.

Set Message Expiry Using the Core API

Using the core API, you can set an expiration time on a message using the **setExpiration** method.

```
// The message will expire 5 seconds from now
message.setExpiration(System.currentTimeMillis() + 5000);
```

Set Message Expiry Using Jakarta Messaging

You can set the time to live for the Jakarta Messaging **MessageProducer** to use when sending messages. You specify this value, in milliseconds, using the **setTimeToLive** method.

```
// Messages sent by this producer will be retained for 5 seconds before expiring
producer.setTimeToLive(5000);
```

You can also specify the message expiry on a per-message basis by setting the time to live on the producer's **send** method.

```
// The last parameter of the send method is the time to live, in milliseconds
producer.send(message, DeliveryMode.PERSISTENT, 0, 5000)
```

Expired messages that are consumed from an expiry address have the following properties.

- **_AMQ_ORIG_ADDRESS**
  A **String** property containing the original address of the expired message.
- **_AMQ_ACTUAL_EXPIRY**
  A **Long** property containing the actual expiration time of the expired message.

16.1. EXPIRY ADDRESS

You can specify where to send expired messages by setting an expiry address. If a message expires and no expiry address is specified, the message is removed from the queue and dropped.

You can set an **expiry-address** for an **address-setting** using the management CLI. In the below example, expired messages in the **jms.queue.exampleQueue** queue will be sent to the **jms.queue.expiryQueue** expiry address.

```
/subsystem=messaging-activemq/server=default/address-setting=jms.queue.exampleQueue:write-attribute(name=expiry-address,value=jms.queue.expiryQueue)
```

16.2. EXPIRY REAPER THREAD

A reaper thread periodically inspects the queues to check whether messages have expired. You can set the scan period and thread priority for the reaper thread using the management CLI.

Set the scan period for the expiry reaper thread, which is how often, in milliseconds, the queues will be scanned to detect expired messages. The default is **30000**. You can set this to **-1** to disable the reaper thread.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=message-expiry-scan-period,value=30000)
```

Set the thread priority for the expiry reaper thread. Possible values are from **0** to **9**, with **9** being the highest priority. The default is **3**.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=message-expiry-thread-priority,value=3)
```

CHAPTER 17. CONFIGURING DELAYED REDELIVERY

Delayed redelivery to an address is defined by the **redelivery-delay** attribute of an **address-setting** configuration element. If a redelivery delay is specified, JBoss EAP waits for the duration of this delay before redelivering messages. If **redelivery-delay** is set to **0**, there is no redelivery delay. To get the current value of **redelivery-delay** for a given **address-setting**, use the following management CLI command as an example.

```
/subsystem=messaging-activemq/server=default/address-
setting=YOUR_ADDRESS_SETTING:read-attribute(name=redelivery-delay)
```

The table below lists the configuration attributes of an **address-setting** that can be used to configure the redelivery of messages. Set the value for a given attribute using the following management CLI command as an example.

```
/subsystem=messaging-activemq/server=default/address-
setting=YOUR_ADDRESS_SETTING:write-attribute(name=ATTRIBUTE,value=NEW_VALUE)
```

Table 17.1. Delivery Related Attributes of Address Settings

Attribute	Description
max-delivery-attempts	Defines how many time a canceled message can be redelivered before sending to the **dead-letter-address**. The default is **10**.
max-redelivery-delay	Maximum value for the **redelivery-delay** in milliseconds. You can set the **max-redelivery-delay** parameter to prevent the delay from becoming too large. The default is **redelivery-delay * 10**.
redelivery-delay	Defines how long to wait in milliseconds before attempting redelivery of a canceled message. The default is **0**.
redelivery-multiplier	Multiplier to apply to the **redelivery-delay** parameter. Each time a message is redelivered, the delay period becomes equal to the previous **redelivery-delay** * **redelivery-multiplier**. The default is **1.0**.

See Address Settings for details on configuring an **address-setting**.

CHAPTER 18. CONFIGURING DEAD LETTER ADDRESSES

A dead letter address is defined in the **address-setting** element of the **messaging-activemq** subsystem configuration. To read the current configuration for a given **address-setting**, use the following management CLI command as an example.

```
/subsystem=messaging-activemq/server=default/address-setting=ADDRESS_SETTING:read-attribute(name=dead-letter-address)
```

If a **dead-letter-address** is not specified, messages are removed after trying to deliver **max-delivery-attempts** times. By default, messages delivery is attempted 10 times. Setting **max-delivery-attempts** to **-1** allows infinite redelivery attempts. The example management CLI commands below illustrate how to set the **dead-letter-address** and the **max-delivery-attempts** attributes for a given **address-setting**.

```
/subsystem=messaging-activemq/server=default/address-setting=ADDRESS_SETTING:write-attribute(name=dead-letter-address,value=NEW_VALUE)

/subsystem=messaging-activemq/server=default/address-setting=ADDRESS_SETTING:write-attribute(name=max-delivery-attempts,value=NEW_VALUE)
```

For example, a dead letter can be set globally for a set of matching addresses and you can set **max-delivery-attempts** to **-1** for a specific address setting to allow infinite redelivery attempts only for this address. Address wildcards can also be used to configure dead letter settings for a set of addresses.

See Address Settings for details on creating and configuring an **address-setting**.

CHAPTER 19. FLOW CONTROL

Flow control can be used to limit the flow of messaging data between a client and server so that messaging participants are not overwhelmed. You can manage the flow of data from both the consumer side and the producer side.

19.1. CONSUMER FLOW CONTROL

JBoss EAP messaging includes configuration that defines how much data to pre-fetch on behalf of consumers and that controls the rate at which consumers can consume messages.

Window-based Flow Control

JBoss EAP messaging pre-fetches messages into a buffer on each consumer. The size of the buffer is determined by the **consumer-window-size** attribute of a **connection-factory**. The example configuration below shows a **connection-factory** with the **consumer-window-size** attribute explicitly set.

```
<connection-factory name="MyConnFactory" ... consumer-window-size="1048576" />
```

Use the management CLI to read and write the value of **consumer-window-size** attribute for a given **connection-factory**. The examples below show how this done using the **InVmConnectionFactory** connection factory, which is the default for consumers residing in the same virtual machine as the server, for example, a local **MessageDrivenBean**.

- Read the **consumer-window-size** attribute of the **InVmConnectionFactory** from the management CLI

```
/subsystem=messaging-activemq/server=default/connection-factory=InVmConnectionFactory:read-attribute(name=consumer-window-size)
{
    "outcome" => "success",
    "result" => 1048576
}
```

- Write the **consumer-window-size** attribute from the management CLI

```
/subsystem=messaging-activemq/server=default/connection-factory=InVmConnectionFactory:write-attribute(name=consumer-window-size,value=1048576)
{"outcome" => "success"}
```

The value for **consumer-window-size** must be an integer. Some values have special meaning as noted in the table below.

Table 19.1. Values for consumer-window-size

Value	Description
n	An integer value used to set the buffer's size to **n** bytes. The default is **1048576**, which should be fine in most cases. Benchmarking will help you find an optimal value for the window size if the default value is not adequate.
0	Turns off buffering. This can help with slow consumers and can give deterministic distribution across multiple consumers.

Value	Description
-1	Creates an unbounded buffer. This can help facilitate very fast consumers that pull and process messages as quickly as they are received.

WARNING

Setting **consumer-window-size** to **-1** can overflow the client memory if the consumer is not able to process messages as fast as it receives them.

If you are using the core API, the consumer window size can be set from the **ServerLocator** using its **setConsumerWindowSize()** method.

If you are using Jakarta Messaging, the client can specify the consumer window size by using the **setConsumerWindowSize()** method of the instantiated **ConnectionFactory**.

Rate-limited Flow Control

JBoss EAP messaging can regulate the rate of messages consumed per second, a flow control method known as throttling. Use the **consumer-max-rate** attribute of the appropriate **connection-factory** to ensure that a consumer never consumes messages at a rate faster than specified.

```
<connection-factory name="MyConnFactory" ... consumer-max-rate="10" />
```

The default value is **-1**, which disables rate limited flow control.

The management CLI is the recommended way to read and write the **consumer-max-rate** attribute. The examples below show how this done using the **InVmConnectionFactory** connection factory, which is the default for consumers residing in the same virtual machine as the server, e.g. a local **MessageDrivenBean**.

- Read the **consumer-max-rate** attribute using the management CLI

```
/subsystem=messaging-activemq/server=default/connection-factory=InVmConnectionFactory:read-attribute(name=consumer-max-rate)
{
    "outcome" => "success",
    "result" => -1
}
```

- Write the **consumer-max-rate** attribute using the management CLI:

```
/subsystem=messaging-activemq/server=default/connection-factory=InVmConnectionFactory:write-attribute(name=consumer-max-rate,value=100)
{"outcome" => "success"}
```

If you are using Jakarta Messaging the max rate size can be set using setConsumerMaxRate(int consumerMaxRate) method of the instantiated **ConnectionFactory**.

If you are using the Core API the rate can be set with the **ServerLocator.setConsumerMaxRate(int consumerMaxRate)** method.

19.2. PRODUCER FLOW CONTROL

JBoss EAP messaging can also limit the amount of data sent from a client in order to prevent the server from receiving too many messages.

Window-based Flow Control

JBoss EAP messaging regulates message producers by using an exchange of credits. Producers can send messages to an address as long as they have sufficient credits to do so. The amount of credits required to send a message is determined by its size. As producers run low on credits, they must request more from the server. Within the server configuration, the amount of credits a producer can request at one time is known as the **producer-window-size**, an attribute of the **connection-factory** element:

```
<connection-factory name="MyConnFactory" ... producer-window-size="1048576" />
```

The window size determines the amount of bytes that can be in-flight at any one time, thus preventing the remote connection from overloading the server.

Use the management CLI to read and write the **producer-window-size** attribute of a given connection factory. The examples below use the **RemoteConnectionFactory**, which is included in the default configuration and intended for use by remote clients.

- Read the **producer-window-size** attribute using the management CLI:

```
subsystem=messaging-activemq/server=default/connection-factory=RemoteConnectionFactory:read-attribute(name=producer-window-size)
{
    "outcome" => "success",
    "result" => 65536
}
```

- Write the **producer-window-size** attribute using the management CLI:

```
/subsystem=messaging-activemq/server=default/connection-factory=RemoteConnectionFactory:write-attribute(name=producer-window-size,value=65536)
{"outcome" => "success"}
```

If you are using Jakarta Messaging, the client can call the **setProducerWindowSize(int producerWindowSize)** method of the **ConnectionFactory** to set the window size directly.

If you are using the core API, the window size can be set using the **setProducerWindowSize(int producerWindowSize)** method of the **ServerLocator**.

Blocking Producer Window-based Flow Control

Typically, the messaging server always provides the same number of credits that was requested. However, it is possible to limit the number of credits sent by the server, which can prevent it from running out of memory due to producers sending more messages than can be handled at one time.

For example, if you have a Jakarta Messaging queue called **myqueue** and you set the maximum memory size to 10MB, the server will regulate the number of messages in the queue so that its size never exceeds

10MB. When the address gets full, producers will block on the client side until sufficient space is freed up on the address.

NOTE

Blocking producer flow control is an alternative approach to paging, which does not block producers but instead pages messages to storage. See About Paging for more information.

The **address-setting** configuration element contains the configuration for managing blocking producer flow control. An **address-setting** is used to apply a set of configuration to all queues registered to that address. See Configuring Address Settings for more information on how this is done.

For each **address-setting** requiring blocking producer flow control, you must include a value for the **max-size-bytes** attribute. The total memory for all queues bound to that address cannot exceed **max-size-bytes**. In the case of Jakarta Messaging topics, this means the total memory of all subscriptions in the topic cannot exceed **max-size-bytes**.

You must also set the **address-full-policy** attribute to **BLOCK** so the server knows that producers should be blocked if **max-size-bytes** is reached. Below is an example **address-setting** with both attributes set:

```
<address-setting ...
    name="myqueue"
    address-full-policy="BLOCK"
    max-size-bytes="100000" />
```

The above example would set the maximum size of the Jakarta Messaging queue "myqueue" to **100000** bytes. Producers will be blocked from sending to that address once it has reached its maximum size.

Use the management CLI to set these attributes, as in the examples below:

- Set **max-size-bytes** for a specified **address-setting**

```
/subsystem=messaging-activemq/server=default/address-setting=myqueue:write-
attribute(name=max-size-bytes,value=100000)
{"outcome" => "success"}
```

- Set **address-full-policy** for a specified **address-setting**

```
/subsystem=messaging-activemq/server=default/address-setting=myqueue:write-
attribute(name=address-full-policy,value=BLOCK)
{"outcome" => "success"}
```

Rate-limited Flow Control

JBoss EAP messaging limits the number of messages a producer can send per second if you specify a **producer-max-rate** for the **connection-factory** it uses, as in the example below:

```
<connection-factory name="MyConnFactory" producer-max-rate="1000" />
```

The default value is **-1**, which disables rate limited flow control.

Use the management CLI to read and write the value for **producer-max-rate**. The examples below use the **RemoteConnectionFactory**, which is included in the default configuration and intended for use by remote clients.

- Read the value of the **producer-max-rate** attribute:

```
/subsystem=messaging-activemq/server=default/connection-
factory=RemoteConnectionFactory:read-attribute(name=producer-max-rate)
{
    "outcome" => "success",
    "result" => -1
}
```

- Write the value of a **producer-max-rate** attribute:

```
/subsystem=messaging-activemq/server=default/connection-
factory=RemoteConnectionFactory:write-attribute(name=producer-max-rate,value=100)
{"outcome" => "success"}
```

If you use the core API, set the rate by using the method **ServerLocator.setProducerMaxRate(int producerMaxRate)**.

If you are using JNDI to instantiate and look up the connection factory, the max rate can be set on the client using the **setProducerMaxRate(int producerMaxRate)** method of the instantiated connection factory.

CHAPTER 20. CONFIGURING PRE-ACKNOWLEDGMENTS

Jakarta Messaging specifies three acknowledgement modes:

- AUTO_ACKNOWLEDGE
- CLIENT_ACKNOWLEDGE
- DUPS_OK_ACKNOWLEDGE

In some cases you can afford to lose messages in the event of a failure, so it would make sense to acknowledge the message on the server before delivering it to the client. This extra mode is supported by JBoss EAP messaging and is called pre-acknowledge mode.

The disadvantage of pre-acknowledging on the server before delivery is that the message will be lost if the server's system crashes after acknowledging the message but before it is delivered to the client. In that case, the message is lost and will not be recovered when the system restarts.

Depending on your messaging case, pre-acknowledge mode can avoid extra network traffic and CPU usage at the cost of coping with message loss.

An example use case for pre-acknowledgement is for stock price update messages. With these messages, it might be reasonable to lose a message in event of a crash since the next price update message will arrive soon, overriding the previous price.

NOTE

If you use pre-acknowledge mode, you will lose transactional semantics for messages being consumed since they are being acknowledged first on the server, not when you commit the transaction.

20.1. CONFIGURING THE SERVER

A connection factory can be configured to use pre-acknowledge mode by setting its **pre-acknowledge** attribute to **true** using the management CLI as below:

```
/subsystem=messaging-activemq/server=default/connection-
factory=RemoteConnectionFactory:write-attribute(name=pre-acknowledge,value=true)
```

20.2. CONFIGURING THE CLIENT

Pre-acknowledge mode can be configured in the client's JNDI context environment, for example, in the **jndi.properties** file:

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connection.ConnectionFactory=tcp://localhost:8080?preAcknowledge=true
```

Alternatively, to use pre-acknowledge mode using the Jakarta Messaging API, create a Jakarta Messaging Session with the **ActiveMQSession.PRE_ACKNOWLEDGE** constant.

```
// messages will be acknowledge on the server *before* being delivered to the client
Session session = connection.createSession(false, ActiveMQJMSConstants.PRE_ACKNOWLEDGE);
```

CHAPTER 21. INTERCEPTORS

JBoss EAP messaging supports interceptors to intercept packets entering and exiting the server. Incoming and outgoing interceptors are called for every packet entering or exiting the server respectively. This allows custom code to be executed, such as for auditing or filtering packets. Interceptors can modify the packets they intercept. This makes interceptors powerful, but also potentially dangerous.

21.1. IMPLEMENTING INTERCEPTORS

An interceptor must implement the Interceptor interface:

```
package org.apache.artemis.activemq.api.core.interceptor;

public interface Interceptor
{
  boolean intercept(Packet packet, RemotingConnection connection) throws ActiveMQException;
}
```

The returned boolean value is important:

- if true is returned, the process continues normally
- if false is returned, the process is aborted, no other interceptors will be called and the packet will not be processed further by the server.

Interceptor classes should be added to JBoss EAP as a module. See Create a Custom Module in the JBoss EAP *Configuration Guide* for more information.

21.2. CONFIGURING INTERCEPTORS

After adding their module to JBoss EAP as a custom module, both incoming and outgoing interceptors are added to the messaging subsystem configuration by using the management CLI.

NOTE

You must start JBoss EAP in *administrator only* mode before the new interceptor configuration will be accepted. See Running JBoss EAP in Admin-only Mode in the JBoss EAP *Configuration Guide* for details. Restart the server in normal mode after the new configuration is processed.

Each interceptor should be added according to the example management CLI command below. The examples assume each interceptor has already been added to JBoss EAP as a custom module.

```
/subsystem=messaging-activemq/server=default:list-add(name=incoming-interceptors, value={name
=> "foo.bar.MyIncomingInterceptor", module=>"foo.bar.interceptors"})
```

Adding an outgoing interceptor follows a similar syntax, as the example below illustrates.

```
/subsystem=messaging-activemq/server=default:list-add(name=outgoing-interceptors, value={name
=> "foo.bar.MyOutgoingInterceptor", module=>"foo.bar.interceptors"})
```

CHAPTER 22. MESSAGE GROUPING

A message group is a group of messages that share certain characteristics:

- All messages in a message group are grouped under a common group ID. This means that they can be identified with a common group property.
- All messages in a message group are serially processed and consumed by the same consumer, irrespective of the number of customers on the queue. This means that a specific message group with a unique group id is always processed by one consumer when the consumer opens it. If the consumer closes the message group, then the entire message group is directed to another consumer in the queue.

Message groups are especially useful when there is a need for messages with a certain value of the property, such as group ID, to be processed serially by a single consumer.

IMPORTANT

Message grouping will not work as expected if the queue has paging enabled. Be sure to disable paging before configuring a queue for message grouping.

For information about configuring message grouping within a cluster of messaging servers, see Clustered Message Grouping in Part III, Configuring Multiple Messaging Systems .

22.1. CONFIGURING MESSAGE GROUPS USING THE CORE API

The property **_AMQ_GROUP_ID** is used to identify a message group using the Core API on the client side. To pick a random unique message group identifier, you can also set the **auto-group** property to **true** on the **SessionFactory**.

22.2. CONFIGURING MESSAGE GROUPS USING JAKARTA MESSAGING

The property **JMSXGroupID** is used to identify a message group for Jakarta Messaging clients. If you wish to send a message group with different messages to one consumer, you can set the same **JMSXGroupID** for different messages.

```
Message message = ...
message.setStringProperty("JMSXGroupID", "Group-0");
producer.send(message);

message = ...
message.setStringProperty("JMSXGroupID", "Group-0");
producer.send(message);
```

An alternative approach is to use the one of the following attributes of the **connection-factory** to be used by the client: **auto-group** or **group-id**.

When **auto-group** is set to **true**, the **connection-factory** will begin to use a random unique message group identifier for all messages sent through it. You can use the management CLI to set the **auto-group** attribute.

```
/subsystem=messaging-activemq/server=default/connection-
factory=RemoteConnectionFactory:write-attribute(name=auto-group,value=true)
```

The **group-id** attribute will set the property **JMSXGroupID** to the specified value for all messages sent through the connection factory. To set a specific **group-id** on the connection factory, use the management CLI.

```
/subsystem=messaging-activemq/server=default/connection-
factory=RemoteConnectionFactory:write-attribute(name=group-id,value="Group-0")
```

CHAPTER 23. DIVERTS

Diverts are objects configured in JBoss EAP messaging that help in diverting messages from one address to another. Diverts can be classified into the following types:

Exclusive

A message is only diverted to a new address and never sent to the old address.

Non-exclusive

A message is sent the old address, and a copy of it is also sent to the new address. Non-exclusive diverts can be used for splitting the flow of messages.

A divert will only divert a message to an address on the same server. If you want to divert to an address on a different server, a common pattern would be to divert to a local store-and-forward queue, then set up a bridge that consumes from that queue and forwards to an address on a different server.

Diverts are therefore a very sophisticated concept. When combined with bridges, diverts can be used to create interesting and complex routings. The set of diverts on a server can be thought of as a type of routing table for messages. Combining diverts with bridges allows you to create a distributed network of reliable routing connections between multiple geographically distributed servers, creating your global messaging mesh. See Configuring Core Bridges for more information on how to use bridges.

Diverts can be configured to apply a **Transformer** and an optional message filter. An optional message filter helps to divert only messages which match the specified filter. For more information on filters see Filter Expressions and Message Selectors .

A transformer is used for transforming messages to another form. When a transformer is specified, all diverted messages are transformed by the **Transformer**. All transformers must implement the **org.apache.activemq.artemis.core.server.cluster.Transformer** interface:

```
package org.apache.activemq.artemis.core.server.cluster;
import org.apache.activemq.artemis.core.server.ServerMessage;

public interface Transformer {
   ServerMessage transform(ServerMessage message);
}
```

To enable JBoss EAP messaging to instantiate an instance of your transformer implementation, you must include it in a JBoss EAP module, and then add the module as an exported dependency to the **org.apache.activemq.artemis** module. See Create a Custom Module in the JBoss EAP *Configuration Guide* for information on how to create a custom module. To add a dependency to the **org.apache.activemq.artemis** module, open the file ***EAP_HOME*/modules/system/layers/base/org/apache/activemq/artemis/main/module.xml** in a text editor and add your **<module>** to the list of **<dependencies>** as in the following example.

```
<module xmlns="urn:jboss:module:1.3" name="org.apache.activemq.artemis">
    <resources>
      ...
    </resources>
    <dependencies>
      ...
      <module name="YOUR_MODULE_NAME" export="true"/>
    </dependencies>
</module>
```

23.1. EXCLUSIVE DIVERTS

Below is in an example of an exclusive divert as it might appear in configuration:

```
<divert
  name="prices-divert"
  address="jms.topic.priceUpdates"
  forwarding-address="jms.queue.priceForwarding"
  filter="office='New York'"
  transformer-class-
name="org.apache.activemq.artemis.jms.example.AddForwardingTimeTransformer"
  exclusive="true"/>
```

In this example, a divert called **prices-divert** is configured that will divert any messages sent to the address **jms.topic.priceUpdates**, which maps to any messages sent to a Jakarta Messaging topic called **priceUpdates**, to another local address **jms.queue.priceForwarding**, corresponding to a local Jakarta Messaging queue called **priceForwarding**

We also specify a message filter string so that only messages with the message property **office** with a value of **New York** will be diverted. All other messages will continue to be routed to the normal address. The filter string is optional, if not specified then all messages will be considered matched.

Note that a transformer class is specified. In this example the transformer simply adds a header that records the time the divert happened.

This example is actually diverting messages to a local store and forward queue, which is configured with a bridge that forwards the message to an address on another server. See Configuring Core Bridges for more details.

23.2. NON-EXCLUSIVE DIVERTS

Below is an example of a non-exclusive divert. Non exclusive diverts can be configured in the same way as exclusive diverts with an optional filter and transformer.

```
<divert
  name="order-divert"
  address="jms.queue.orders"
  forwarding-address="jms.topic.spytopic"
  exclusive="false"/>
```

The above divert takes a copy of every message sent to the address **jms.queue.orders**, which maps to a Jakarta Messaging queue called **orders**, and sends it to a local address called **jms.topic.SpyTopic**, corresponding to a Jakarta Messaging topic called **SpyTopic**.

Creating diverts

Use the management CLI to create the type of divert you want:

```
/subsystem=messaging-activemq/server=default/divert=my-divert:add(divert-
address=news.in,forwarding-address=news.forward)
```

Non-exclusive diverts are created by default. To create an exclusive divert use the **exclusive** attribute:

```
/subsystem=messaging-activemq/server=default/divert=my-exclusive-divert:add(divert-
address=news.in,forwarding-address=news.forward,exclusive=true)
```

The below table captures a divert's attributes and their description. You can have the management CLI display this information using the following command:

```
/subsystem=messaging-activemq/server=default/divert=*:read-resource-description()
```

Attribute	Description
divert-address	Address to divert from. Required.
exclusive	Whether the divert is exclusive, meaning that the message is diverted to the new address, and does not go to the old address at all. The default is false.
filter	An optional filter string. If specified then only messages which match the filter expression will be diverted.
forwarding-address	Address to divert to. Required.
routing-name	Routing name of the divert.
transformer-class-name	The name of a class used to transform the message's body or properties before it is diverted.

CHAPTER 24. THREAD MANAGEMENT

Each JBoss EAP messaging server maintains a single thread pool for general use, and scheduled thread pool for scheduled use. A Java scheduled thread pool cannot be configured to use a standard thread pool, otherwise we could use a single thread pool for both scheduled and non scheduled activity.

Note that JBoss EAP uses the new, non-blocking NIO. By default, JBoss EAP messaging uses a number of threads equal to three times the number of cores, or hyper-threads, as reported by **.getRuntime().availableProcessors()** for processing incoming packets. To override this value, set the number of threads by specifying the **nio-remoting-threads** parameter in the transport configuration. See Configuring the Messaging Transports for more information.

24.1. SERVER SCHEDULED THREAD POOL

The server scheduled thread pool is used for most activities on the server side that require running periodically or with delays. It maps internally to a java.util.concurrent.ScheduledThreadPoolExecutor instance.

The maximum number of thread used by this pool is configured using the **scheduled-thread-pool-max-size** parameter. The default value is 5 threads. A small number of threads is usually sufficient for this pool. To change this value for the default JBoss EAP messaging server, use the following management CLI command:

```
/subsystem=messaging-activemq/server=default:write-attribute(name=scheduled-thread-pool-max-size,value=10)
```

24.2. SERVER GENERAL PURPOSE THREAD POOL

The general purpose thread pool is used for most asynchronous actions on the server side. It maps internally to a **java.util.concurrent.ThreadPoolExecutor** instance.

The maximum number of threads used by this pool is configured using the **thread-pool-max-size** attribute.

If **thread-pool-max-size** is set to **-1**, the thread pool has no upper bound and new threads are created on demand if there are not enough threads available to fulfill a request. If activity later subsides, then threads are timed out and closed.

If **thread-pool-max-size** is set to a positive integer greater than zero, the thread pool is bounded. If requests come in and there are no free threads available in the pool, requests will block until a thread becomes available. It is recommended that a bounded thread pool be used with caution since it can lead to deadlock situations if the upper bound is configured too low.

The default value for **thread-pool-max-size** is **30**. To set a new value for the default JBoss EAP messaging server, use the following management CLI command.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=thread-pool-max-size,value=40)
```

See the **ThreadPoolExecutor** Javadoc for more information on unbounded (cached), and bounded (fixed) thread pools.

24.3. MONITORING SERVER THREAD UTILIZATION

Check thread utilization to ensure you have configured the size of the thread pools appropriately.

To check thread utilization, issue the following CLI command:

```
/subsystem=messaging-activemq:read-resource(include-runtime)
```

The system returns results similar to the following

```
{
    "outcome" => "success",
    "result" => {
        "global-client-scheduled-thread-pool-active-count" => 0,
        "global-client-scheduled-thread-pool-completed-task-count" => 0L,
        "global-client-scheduled-thread-pool-current-thread-count" => 0,
        "global-client-scheduled-thread-pool-keepalive-time" => 10000000L,
        "global-client-scheduled-thread-pool-largest-thread-count" => 0,
        "global-client-scheduled-thread-pool-max-size" => undefined,
        "global-client-scheduled-thread-pool-task-count" => 0L,
        "global-client-thread-pool-active-count" => 0,
        "global-client-thread-pool-completed-task-count" => 2L,
        "global-client-thread-pool-current-thread-count" => 2,
        "global-client-thread-pool-keepalive-time" => 60000000000L,
        "global-client-thread-pool-largest-thread-count" => 2,
        "global-client-thread-pool-max-size" => undefined,
        "global-client-thread-pool-task-count" => 2L,
        "connection-factory" => undefined,
        "connector" => undefined,
        "discovery-group" => undefined,
        "external-jms-queue" => undefined,
        "external-jms-topic" => undefined,
        "http-connector" => undefined,
        "in-vm-connector" => undefined,
        "jms-bridge" => undefined,
        "pooled-connection-factory" => undefined,
        "remote-connector" => undefined,
        "server" => {"default" => undefined}
    }
}
```

Table 24.1. Thread Utilization Data

Utilization attribute	Description
global-client-scheduled-thread-pool-active-count	The number of scheduled pool threads in use by all ActiveMQ clients currently executing tasks.
global-client-scheduled-thread-pool-completed-task-count	The number of tasks using scheduled pool threads that have been executed by all ActiveMQ clients since the server was booted.
global-client-scheduled-thread-pool-current-thread-count	The current number of threads in the scheduled pool that are in use by all Active MQ clients.

Utilization attribute	Description
global-client-scheduled-thread-pool-keepalive-time	The amount of time to keep threads in the scheduled thread pool running when idle.
global-client-scheduled-thread-pool-largest-thread-count	The largest number of threads in the scheduled pool that have ever been used simultaneously by all Active MQ clients.
global-client-scheduled-thread-pool-max-size	Maximum number of threads in the the scheduled pool used by all ActiveMQ clients running inside this server.
global-client-scheduled-thread-pool-task-count	The total number of tasks in the scheduled thread pool that have ever been scheduled by all Active MQ clients.
global-client-thread-pool-active-count	The number of general-purpose pool threads being used by all ActiveMQ clients currently executing tasks.
global-client-thread-pool-completed-task-count	The number of tasks using general-purpose pool threads that have been executed by all ActiveMQ clients.
global-client-thread-pool-current-thread-count	The current number of threads in the general-purpose pool that are in use by all Active MQ clients.
global-client-thread-pool-keepalive-time	The amount of time that threads in the general-purpose thread pool should be kept running when idle.
global-client-thread-pool-largest-thread-count	The largest number of threads in the general-purpose pool that have ever been used simultaneously by all Active MQ clients.
global-client-thread-pool-max-size	Maximum number of threads in the general-purpose pool used by all ActiveMQ clients running inside this server.
global-client-thread-pool-task-count	The total number of tasks in the general-purpose thread pool that have ever been scheduled by all Active MQ clients.

24.4. EXPIRY REAPER THREAD

A single thread is also used on the server side to scan for expired messages in queues. We cannot use either of the thread pools for this since this thread needs to run at its own configurable priority.

24.5. ASYNCHRONOUS IO

Asynchronous IO has a thread pool for receiving and dispatching events out of the native layer. It is on a thread dump with the prefix **ArtemisMQ-AIO-poller-pool**. JBoss EAP messaging uses one thread per opened file on the journal (there is usually one).

There is also a single thread used to invoke writes on libaio. It is done to avoid context switching on libaio that would cause performance issues. This thread is found on a thread dump with the prefix **ArtemisMQ-AIO-writer-pool**.

24.6. CLIENT THREAD MANAGEMENT

JBoss EAP includes a client thread pool used for creating client connections. This pool is separate from the static pools mentioned earlier in this chapter and is used by JBoss EAP when it behaves like a client. For example, client thread pool clients are created as cluster connections with other nodes in the same cluster, or when the Artemis resource adapter connects to a remote Apache ActiveMQ Artemis messaging server integrated in a remote instance of JBoss EAP. There is a pool for scheduled client threads as well.

NOTE

With the release of JBoss EAP 7.1, client threads now timeout after 60 seconds of no activity.

Setting Client Thread Pool Size Using the Management CLI

Use the management CLI to configure the size of both the client thread pool and the client scheduled thread pool. Pool sizes set using the management CLI will have precedence over the sizings set using system properties.

The command below sets the client thread pool.

```
/subsystem=messaging-activemq:write-attribute(name=global-client-thread-pool-max-size,value=POOL_SIZE)
```

There is no default value defined for this attribute. If the attribute is not defined, the maximum size of the pool is determined to be eight (8) times the number of CPU core processors.

To review the current settings, use the following command.

```
/subsystem=messaging-activemq:read-attribute(name=global-client-thread-pool-max-size)
```

The client scheduled thread pool size is set using the following command.

```
/subsystem=messaging-activemq:write-attribute(name=global-client-scheduled-thread-pool-max-size,value=POOL_SIZE)
```

The following will display the current pool size for the client scheduled thread pool. The default value is **5**.

```
/subsystem=messaging-activemq:read-attribute(name=global-client-scheduled-thread-pool-max-size)
```

Setting Client Thread Pool Size Using System Properties

The following system properties can be used to set the size of the client's global and global scheduled thread pools respectively:

- **activemq.artemis.client.global.thread.pool.max.size**
- **activemq.artemis.client.global.scheduled.thread.pool.core.size**

The system properties can then be referenced in XML configuration, as in the example below.

NOTE

Pool sizes set using the management CLI will have precedence over sizes set by system properties.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <global-client thread-pool-max-size="${activemq.artemis.client.global.thread.pool.max.size}"
    scheduled-thread-pool-max-
size="${activemq.artemis.client.global.scheduled.thread.pool.core.size}" />
  <server ...>
  </server>
  ...
</subsystem>
```

Configuring a Client to Use Its Own Thread Pool

A client can configure each of its **ClientSessionFactory** instances to not use the pool provided by JBoss EAP, but instead to use its own client thread pool. Any sessions created from that **ClientSessionFactory** will use the newly created pool.

To configure a **ClientSessionFactory** instance to use its own pools, invoke the appropriate setter methods immediately after you created the factory. For example:

```
ServerLocator locator = ActiveMQClient.createServerLocatorWithoutHA(transportConfiguration);
locator.setUseGlobalPools(true);
locator.setThreadPoolMaxSize(-1);
locator.setScheduledThreadPoolMaxSize(10);
ClientSessionFactory myFactory = locator.createSessionFactory();
```

If you are using the Jakarta Messaging API, you can set the same parameters on the **ClientSessionFactory**. For example:

```
ActiveMQConnectionFactory myConnectionFactory =
ActiveMQJMSClient.createConnectionFactory(url, "ConnectionFactoryName");
myConnectionFactory.setUseGlobalPools(false);
myConnectionFactory.setScheduledThreadPoolMaxSize(10);
myConnectionFactory.setThreadPoolMaxSize(-1);
```

If you are using JNDI to instantiate **ActiveMQConnectionFactory** instances, you can also set these parameters using the management CLI, as in the examples given below for a standalone instance of JBoss EAP.

```
/subsystem=messaging-activemq/server=default/connection-factory=myConnectionFactory:write-
attribute(name=use-global-pools,value=false)

/subsystem=messaging-activemq/server=default/connection-factory=myConnectionFactory:write-
```

```
attribute(name=scheduled-thread-pool-max-size,value=10)

/subsystem=messaging-activemq/server=default/connection-factory=myConnectionFactory:write-
attribute(name=thread-pool-max-size,value=1)
```

Note that the management CLI will remind you that a reload of the instance is required after you execute each of the above commands.

CHAPTER 25. CONFIGURING DUPLICATE MESSAGE DETECTION

When a sender sends a message to another server, there can be a situation where the target server or the connection fails after sending the message, but before sending a response to the sender indicating that the process was successful. In these situations, it is very difficult for the sender to determine whether the message was sent successfully to the intended receiver. If the sender decides to resend the last message, it can result in a duplicate message being sent to the address.

You can configure duplicate message detection in JBoss EAP messaging so that your application does not need to provide the logic to filter duplicate messages.

25.1. USING DUPLICATE MESSAGE DETECTION FOR SENDING MESSAGES

To enable duplicate message detection for sent messages, you need to set the value of the **org.apache.activemq.artemis.api.core.Message.HDR_DUPLICATE_DETECTION_ID** property, which resolves to **_AMQ_DUPL_ID**, to a unique value. When the target server receives the messages, if the **_AMQ_DUPL_ID** property is set, it will check its memory cache to see if it has already received a message with the value of that header. If it has, then this message will be ignored. See Configuring the Duplicate ID Cache for more information.

The value of the **_AMQ_DUPL_ID** property can be of type **byte[]** or **SimpleString** if you are using the core API. If you are using Jakarta Messaging, it must be a **String**.

The following example shows how to set the property for core API.

```
SimpleString myUniqueID = "This is my unique id";   // Can use a UUID for this

ClientMessage message = session.createMessage(true);
message.setStringProperty(HDR_DUPLICATE_DETECTION_ID, myUniqueID);
```

The following example shows how to set the property for Jakarta Messaging clients.

```
String myUniqueID = "This is my unique id";   // Can use a UUID for this

Message jmsMessage = session.createMessage();
message.setStringProperty(HDR_DUPLICATE_DETECTION_ID.toString(), myUniqueID);
```

IMPORTANT

Duplicate messages are *not* detected when they are sent within the same transaction using the **HDR_DUPLICATE_DETECTION_ID** property.

25.2. CONFIGURING THE DUPLICATE ID CACHE

The server maintains caches of received values of the **_AMQ_DUPL_ID** property that is sent to each address. Each address maintains its own address cache.

The cache is fixed in terms of size. The maximum size of cache is configured using the **id-cache-size** attribute. The default value of this parameter is **20000** elements. If the cache has a maximum size of **n** elements, then the (**n + 1**)th ID stored will overwrite the element **0** in the cache. The value is set using the following management CLI command:

```
/subsystem=messaging-activemq/server=default:write-attribute(name=id-cache-size,value=SIZE)
```

The caches can also be configured to persist to disk. This can be configured by setting the **persist-id-cache** attribute using the following management CLI command.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=persist-id-cache,value=true)
```

If this value is set to **true**, then each ID will be persisted to permanent storage as they are received. The default value for this parameter is **true**.

NOTE

Set the size of the duplicate ID cache to a large size in order to ensure that resending of messages does not overwrite the previously sent messages stored in the cache.

CHAPTER 26. HANDLING SLOW CONSUMERS

A slow consumer with a server-side queue can pose a significant problem for server performance. As messages build up in the consumer's server-side queue, memory usage increases. Consequently, the server can enter paging mode, which can negatively impact performance. Another significant problem is that messages sent to a client's message buffer can remain waiting to be consumed if the **consumer-windows-size** attribute is greater than **0**. Criteria can be set so that consumers that do not consume messages quickly enough can be disconnected from the server.

In the case where the slow consumer is an MDB, the JBoss EAP server manages the connection. Once the MDB is disconnected, the messages are returned to the queue from which the MDB was consuming, the MDB automatically reconnects, and at this moment, messages are load balanced again to all of the MDBs on the queue. This same process holds for an MDB listening on a durable topic. In the case of a Jakarta Messaging consumer, if it is slow, then it is disconnected, and it reconnects only if the **reconnects-attempts** is set to **-1** or is greater than **0**.

In the case of a non-durable Jakarta Messaging subscriber or an MDB with non-durable subscription, the connection is disconnected, which results in the subscription being removed. If the non-durable subscriber reconnects, then a new non-durable subscription is created, and it starts to consume only new messages sent to the topic.

The calculation to determine whether or not a consumer is slow inspects only the number of messages that a particular consumer has acknowledged. It does not take into account whether flow control has been enabled on the consumer, or whether the consumer is streaming a large message, for example. Keep this in mind when configuring slow consumer detection.

Slow consumer checks are performed using the scheduled thread pool. Each queue on the server with slow consumer detection enabled will cause a new entry in the internal **java.util.concurrent.ScheduledThreadPoolExecutor** instance. If there are a high number of queues and the **slow-consumer-check-period** is relatively low, then there may be delays in executing some of the checks. However, this will not impact the accuracy of the calculations used by the detection algorithm. See Thread Management for more details about this thread pool.

Slow consumer handling is on a per **address-setting** basis. See Address Settings for more information on configuring an **address-setting**, and refer to the appendix for the list of **address-setting** attributes. There are three attributes used to configure the handling of slow consumers. They are:

slow-consumer-check-period

How often to check, in seconds, for slow consumers. The default is **5**.

slow-consumer-policy

Determines what happens when a slow consumer is identified. The valid options are **KILL** or **NOTIFY**:

- **KILL** will kill the consumer's connection, which will impact any client threads using that same connection.
- **NOTIFY** will send a **CONSUMER_SLOW** management notification to the client.

The default is **NOTIFY**.

slow-consumer-threshold

The minimum rate of message consumption allowed before a consumer is considered slow. The default is **-1**, which is unbounded.

Use the management CLI to read the current values for any of the attributes. For example, use the following command to read the current **slow-consumer-policy** for the **address-setting** with the name **myAddress**.

```
/subsystem=messaging-activemq/server=default/address-setting=myAddress:read-attribute(name=slow-consumer-policy)
```

Likewise, use the following example to set the same **slow-consumer-policy**.

```
/subsystem=messaging-activemq/server=default/address-setting=myAddress:write-attribute(name=slow-consumer-policy,value=<NEW_VALUE>)
```

PART III. CONFIGURING MULTI-NODE MESSAGING SYSTEMS

CHAPTER 27. CONFIGURING JAKARTA MESSAGING BRIDGES

JBoss EAP messaging includes a Jakarta Messaging bridge, which takes messages from a source destination and send them to a target destination, usually on a different server.

A Jakarta Messaging bridge supports destination mapping in which each link consists of a source and a target defined below:

- The source defines the destination from which the Jakarta Messaging bridge receives messages. The source consists of a connection factory for creating connections to a Jakarta Messaging provider and a message source destination in that provider.

- The target defines the destination to which the Jakarta Messaging bridge sends messages received from the source. The target consists of a connection factory for creating connections to a Jakarta Messaging provider and a message target destination in that provider.

If the source destination is a topic, the Jakarta Messaging bridge creates a subscription for it. If the **client-id** and **subscription-name** attributes are configured for the Jakarta Messaging bridge, the subscription is durable. This means that no messages are missed if the Jakarta Messaging bridge is stopped and then restarted.

The source and target servers do not have to be in the same cluster, which makes bridging suitable for reliably sending messages from one cluster to another, for instance across a WAN, and where the connection may be unreliable.

NOTE

Do not confuse a Jakarta Messaging bridge with a core bridge. A Jakarta Messaging bridge can be used to bridge any two Jakarta Messaging-1.1-compliant providers and uses the Jakarta Messaging API. A Configuring core bridges is used to bridge any two JBoss EAP messaging instances and uses the core API. Whenever possible, use a core bridge instead of a Jakarta Messaging bridge.

Example configuration of a JBoss EAP Jakarta Messaging bridge

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
  ...
  </server>
  <jms-bridge name="my-jms-bridge" module="org.apache.activemq.artemis" max-batch-time="100"
max-batch-size="10" max-retries="1" failure-retry-interval="500" quality-of-
service="AT_MOST_ONCE">
    <source destination="jms/queue/InQueue" connection-factory="ConnectionFactory">
      <source-context/>
    </source>
    <target destination="jms/queue/OutQueue" connection-factory="jms/RemoteConnectionFactory">
      <target-context>
        <property name="java.naming.factory.initial"
value="org.wildfly.naming.client.WildFlyInitialContextFactory"/>
        <property name="java.naming.provider.url" value="http-remoting://192.168.40.1:8080"/>
      </target-context>
    </target>
  </jms-bridge>
  ...
</subsystem>
```

In the above example configuration, the Jakarta Messaging bridge uses the **connection-factory** attribute for creating the following two connections:

- Source-context, which defines the original destination of the received messages.
- Target-context, which defines the target destination that receives the messages.

You can use the default implementation provided by Apache ActiveMQ Artemis or Red Hat AMQ that searches the connection factory by using Java Naming and Directory Interface (JNDI). For other application servers or Jakarta Messaging providers, you can provide a new implementation by implementing the interface **org.apache.activemq.artemis.jms.bridge.ConnectionFactoryFactory**.

Adding a Jakarta Messaging Bridge Using the Management CLI

A Jakarta Messaging bridge can be added using the following management CLI command. Note that the source and target destinations must already be defined in the configuration. See the table in the appendix for a full list of configurable attributes.

```
/subsystem=messaging-activemq/jms-bridge=my-jms-bridge:add(quality-of-
service=AT_MOST_ONCE,module=org.apache.activemq.artemis,failure-retry-interval=500,max-
retries=1,max-batch-size=10,max-batch-time=100,source-connection-
factory=ConnectionFactory,source-destination=jms/queue/InQueue,source-context={},target-
connection-factory=jms/RemoteConnectionFactory,target-destination=jms/queue/OutQueue,target-
context=
{java.naming.factory.initial=org.wildfly.naming.client.WildFlyInitialContextFactory,java.naming.provider.u
=http-remoting://192.168.40.1:8080})
```

You can review the configuration of a Jakarta Messaging bridge using the **read-resource** command in the management CLI as in the following example.

```
/subsystem=messaging-activemq/jms-bridge=my-jms-bridge:read-resource
```

Add configuration to a Jakarta Messaging bridge by using the **write-attribute**, as done in this example:

```
/subsystem=messaging-activemq/jms-bridge=my-jms-bridge:write-
attribute(name=ATTRIBUTE,value=VALUE)
```

Adding a Jakarta Messaging Bridge Using the Management Console

You can also use the management console to add a Jakarta Messaging bridge by following these steps.

1. Open the management console in a browser and navigate to **Configuration → Subsystems → Messaging (ActiveMQ) → JMS Bridge**.
2. Click the Add (**+**) button and provide the required information when prompted.
3. Click **Add** when finished.

27.1. QUALITY OF SERVICE

In JBoss EAP, **quality-of-service** is a configurable attribute that determines how messages are consumed and acknowledged. The valid values for **quality-of-service** and their descriptions are below. See the table in the appendix for a full list of Jakarta Messaging bridge attributes.

AT_MOST_ONCE

Messages will reach the destination from the source at the most one time. The messages are consumed from the source and acknowledged before sending to the destination. Therefore, there is a possibility that messages could be lost if a failure occurs between their removal from the source and their arrival at the destination. This mode is the default value.
This mode is available for both durable and non-durable messages.

DUPLICATES_OK

Messages are consumed from the source and then acknowledged after they have been successfully sent to the destination. Therefore, there is a possibility that messages could be sent again if a failure occurs after the initial message was sent but before it is acknowledged.
This mode is available for both durable and non-durable messages.

ONCE_AND_ONLY_ONCE

Messages will reach the destination from the source once and only once. If both the source and the destination are on the same server instance, this can be achieved by sending and acknowledging the messages in the same local transaction. If the source and destination are on different servers, this is achieved by enlisting the sending and consuming sessions in Jakarta Transactions. The transaction is controlled by a Transaction Manager which will need to be set using the **setTransactionManager()** method on the bridge.
This mode is only available for durable messages.

WARNING

When shutting down a server that has a deployed Jakarta Messaging bridge with the **quality-of-service** attribute set to **ONCE_AND_ONLY_ONCE**, be sure to shut the server down with the Jakarta Messaging bridge first to avoid unexpected errors.

It may possible to provide once and only once semantics by using the **DUPLICATES_OK** mode instead of **ONCE_AND_ONLY_ONCE** and then checking for duplicates at the destination and discarding them. See Configuring Duplicate Message Detection for more information. However, the cache would only be valid for a certain period of time. This approach therefore is not as watertight as using **ONCE_AND_ONLY_ONCE**, but it may be a good choice depending on your specific application needs.

27.2. TIMEOUTS AND THE JAKARTA MESSAGING BRIDGE

There is a possibility that the target or source server will not be available at some point in time. If this occurs, the bridge will try to reconnect a number of times equal to the value of **max-retries**. The wait between attempts is set by **failure-retry-interval**.

WARNING

Because each Jakarta Messaging bridge has its own **max-retries** parameter, you should use a connection factory that does not set the **reconnect-attempts** parameter, or sets it to **0**. This will avoid a potential collision that may result in longer reconnection times. Also note that any connection factory referenced by a Jakarta Messaging bridge with the **quality-of-service** set to **ONCE_AND_ONLY_ONCE** needs to have the **factory-type** set to **XA_GENERIC**, **XA_TOPIC**, or **XA_QUEUE**.

27.3. RESOLVING THE REMOTECONNECTIONFACTORY EXCEPTION

When configuring a Jakarta Messaging message bridge, you might get the **RemoteConnectionFactory** exception. To resolve this exception, perform one or all of the following steps:

- Add the URL prefix **java:/jboss/exported**, if needed, based on your client connection from within a Java 2 Platform Enterprise Edition (J2EE) component, a non-J2EE component or remote component.
- Use two Java Naming and Directory Interface names for your connection factory as shown below:

```
<jms-connection-factories>
 <connection-factory name="RemoteConnectionFactory">
  <connectors>
    <connector-ref connector-name="netty"/>
  </connectors>
  <entries>
    <entry name="java:jboss/exported/jms/RemoteConnectionFactory"/> <entry
name="jms/RemoteConnectionFactory"/>
  </entries>
 </connection-factory>
</jms-connection-factories>
```

- Use the following code snippet in the spring bean configuration file:

```
<jee:jndi-lookup id="jmsConnectionFactory" jndi-name="java:/ConnectionFactory" expected-
type="javax.jms.ConnectionFactory" lookup-on-startup="false" />
```

NOTE

The value of the **lookup-on-startup** and **jndi-name** attributes might change according to the application.

CHAPTER 28. CONFIGURING CORE BRIDGES

The function of a bridge is to consume messages from one destination and forward them to another one, typically on a different JBoss EAP messaging server.

The source and target servers do not have to be in the same cluster which makes bridging suitable for reliably sending messages from one cluster to another, for instance across a WAN, or internet and where the connection may be unreliable.

The bridge has built-in resilience to failure so if the target server connection is lost, for example, due to network failure, the bridge will retry connecting to the target until it comes back online. When it comes back online it will resume operation as normal.

Bridges are a way to reliably connect two separate JBoss EAP messaging servers together. With a core bridge both source and target servers must be JBoss EAP 7 messaging servers.

NOTE

Do not confuse a core bridge with a Jakarta Messaging bridge. A core bridge is used to bridge any two JBoss EAP messaging instances and uses the core API. A Jakarta Messaging bridge can be used to bridge any two Jakarta Messaging 2.0 compliant Jakarta Messaging providers and uses the Jakarta Messaging API. It is preferable to use a core bridge instead of a Jakarta Messaging bridge whenever possible.

Below is an example configuration of a JBoss EAP messaging core bridge.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <bridge name="my-core-bridge" static-connectors="bridge-connector" queue-
name="jms.queue.InQueue"/>
    ...
  </server>
</subsystem>
```

This core bridge can be added using the following management CLI command. Note that when defining a core bridge, you must define a **queue-name** and either **static-connectors** or **discovery-group**. See the table in the appendix for a full list of configurable attributes.

```
/subsystem=messaging-activemq/server=default/bridge=my-core-bridge:add(static-connectors=
[bridge-connector],queue-name=jms.queue.InQueue)
```

28.1. CONFIGURING A CORE BRIDGE FOR DUPLICATE DETECTION

Core bridges can be configured to automatically add a unique duplicate ID value, if there is not already one in the message, before forwarding the message to the target. To configure a core bridge for duplicate message detection set the **use-duplicate-detection** attribute to **true**, which is the default value.

```
/subsystem=messaging-activemq/server=default/bridge=my-core-bridge:write-attribute(name=use-
duplicate-detection,value=true)
```

CHAPTER 29. CLUSTERS OVERVIEW

JBoss EAP messaging clusters allow groups of JBoss EAP messaging servers to be grouped together in order to share message processing load. Each active node in the cluster is an active JBoss EAP messaging server which manages its own messages and handles its own connections.

WARNING

A mixed cluster consisting of different versions of JBoss EAP is not supported by the **messaging-activemq** subsystem. The servers that form the messaging cluster must all use the same version of JBoss EAP.

The cluster is formed by each node declaring cluster connections to other nodes in the JBoss EAP configuration file. When a node forms a cluster connection to another node, it internally creates a core bridge connection between itself and the other node. This is done transparently behind the scenes; you do not have to declare an explicit bridge for each node. These cluster connections allow messages to flow between the nodes of the cluster to balance the load.

An important part of clustering is server discovery where servers can broadcast their connection details so clients or other servers can connect to them with minimum configuration.

This section also discusses client-side load balancing, to balance client connections across the nodes of the cluster, and message redistribution, where JBoss EAP messaging will redistribute messages between nodes to avoid starvation.

WARNING

Once a cluster node has been configured, it is common to simply copy that configuration to other nodes to produce a symmetric cluster.

In fact, each node in the cluster must share the same configuration for the following elements in order to avoid unexpected errors:

- cluster-connection
- broadcast-group
- discovery-group
- address-settings, including queues and topics

However, care must be taken when copying the JBoss EAP messaging files. Do not copy the messaging data, the bindings, journal, and large-messages directories from one node to another. When a cluster node is started for the first time and initializes its journal files, it persists a special identifier to the journal directory. The identifier must be unique among nodes for the cluster to form properly.

29.1. SERVER DISCOVERY

Server discovery is a mechanism by which servers can propagate their connection details to:

- Messaging clients
  A messaging client wants to be able to connect to the servers of the cluster without having specific knowledge of which servers in the cluster are up at any one time.
- Other servers
  Servers in a cluster want to be able to create cluster connections to each other without having prior knowledge of all the other servers in the cluster.

This information, or cluster topology, is sent around normal JBoss EAP messaging connections to clients and to other servers over cluster connections. However, you need a way to establish the initial first connection. This can be done using dynamic discovery techniques like UDP and JGroups, or by providing a static list of initial connectors.

29.1.1. Broadcast Groups

A broadcast group is the means by which a server broadcasts connectors over the network. A connector defines a way in which a client, or other server, can make connections to the server.

The broadcast group takes a set of connectors and broadcasts them on the network. Depending on which broadcasting technique you configure the cluster, it uses either UDP or JGroups to broadcast connector pairs information.

Broadcast groups are defined in the **messaging-activemq** subsystem of the server configuration. There can be many broadcast groups per JBoss EAP messaging server.

Configure a Broadcast Group Using UDP
Below is an example configuration of a messaging server that defines a UDP broadcast group. Note that this configuration relies on a **messaging-group** socket binding.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <broadcast-group name="my-broadcast-group" connectors="http-connector" socket-
binding="messaging-group"/>
    ...
  </server>
</subsystem>
...
<socket-binding-group name="standard-sockets" default-interface="public" port-
offset="${jboss.socket.binding.port-offset:0}">
  ...
  <socket-binding name="messaging-group" interface="private" port="5432" multicast-
address="231.7.7.7" multicast-port="9876"/>
  ...
</socket-binding-group>
```

This configuration can be achieved using the following management CLI commands:

1. Add the **messaging-group** socket binding.

```
/socket-binding-group=standard-sockets/socket-binding=messaging-
group:add(interface=private,port=5432,multicast-address=231.7.7.7,multicast-port=9876)
```

2. Add the broadcast group.

```
/subsystem=messaging-activemq/server=default/broadcast-group=my-broadcast-
group:add(socket-binding=messaging-group,broadcast-period=2000,connectors=[http-
connector])
```

Configure a Broadcast Group Using JGroups

Below is an example configuration of a messaging server that defines broadcast group that uses the default JGroups broadcast group, which uses UDP. Note that to be able to use JGroups to broadcast, you must set a **jgroups-channel**.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <broadcast-group name="my-broadcast-group" connectors="http-connector" jgroups-
cluster="activemq-cluster"/>
    ...
  </server>
</subsystem>
```

This can be configured using the following management CLI command:

```
/subsystem=messaging-activemq/server=default/broadcast-group=my-broadcast-
group:add(connectors=[http-connector],jgroups-cluster=activemq-cluster)
```

Broadcast Group Attributes

The below table lists the configurable attributes for a broadcast group.

Attribute	Description
broadcast-period	The period in milliseconds between consecutive broadcasts.
connectors	The names of connectors that will be broadcast.
jgroups-channel	The name of a channel defined in the **jgroups** subsystem that is used in combination with the **jgroups-cluster** attribute to form a cluster. If undefined, the **default** channel will be used. Note that a**jgroups-channel** multiplexes group communication between distinct logical groups, which are identified by the **jgroups-cluster** attribute.
jgroups-cluster	The logical name used for the communication between a broadcast group and a discovery group. A discovery group intending to receive messages from a particular broadcast group must use the same cluster name used by the broadcast group.

Attribute	Description
jgroups-stack	The name of a stack defined in the **jgroups** subsystem that is used to form a cluster. This attribute is deprecated. Use **jgroups-channel** to form a cluster instead. Since each **jgroups-stack** is already associated with a **jgroups-channel**, you can use that channel or you can create a new **jgroups-channel** and associate it with the**jgroups-stack**. IMPORTANT If a **jgroups-stack** and a **jgroups-channel** are both specified, a new **jgroups-channel** is generated and is registered in the same namespace as the **jgroups-stack** and **jgroups-channel**. For this reason, the **jgroups-stack** and **jgroup-channel** names must be unique.
socket-binding	The broadcast group socket binding.

29.1.2. Discovery Groups

While the broadcast group defines how connector information is broadcasted from a server, a discovery group defines how connector information is received from a broadcast endpoint, for example, a UDP multicast address or JGroup channel.

A discovery group maintains a list of connectors, one for each broadcast by a different server. As it receives broadcasts on the broadcast endpoint from a particular server, it updates its entry in the list for that server. If it has not received a broadcast from a particular server for a length of time it will remove that server's entry from its list.

Discovery groups are used in two places in JBoss EAP messaging:

- By cluster connections so they know how to obtain an initial connection to download the topology.
- By messaging clients so they know how to obtain an initial connection to download the topology.

Although a discovery group will always accept broadcasts, its current list of available live and backup servers is only ever used when an initial connection is made. From then on, server discovery is done over the normal JBoss EAP messaging connections.

NOTE

Each discovery group must be configured with a broadcast endpoint (UDP or JGroups) that matches its broadcast group counterpart. For example, if the broadcast group is configured using UDP, the discovery group must also use UDP and the same multicast address.

29.1.2.1. Configure Discovery Groups on the Server

Discovery groups are defined in the **messaging-activemq** subsystem of the server configuration. There can be many discovery groups per JBoss EAP messaging server.

Configure a Discovery Group Using UDP

Below is an example configuration of a messaging server that defines a UDP discovery group. Note that this configuration relies on a **messaging-group** socket binding.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <discovery-group name="my-discovery-group" refresh-timeout="10000" socket-binding="messaging-group"/>
    ...
  </server>
</subsystem>
...
<socket-binding-group name="standard-sockets" default-interface="public" port-offset="${jboss.socket.binding.port-offset:0}">
  ...
  <socket-binding name="messaging-group" interface="private" port="5432" multicast-address="231.7.7.7" multicast-port="9876"/>
  ...
</socket-binding-group>
```

This configuration can be achieved using the following management CLI commands:

1. Add the **messaging-group** socket binding.

   ```
   /socket-binding-group=standard-sockets/socket-binding=messaging-group:add(interface=private,port=5432,multicast-address=231.7.7.7,multicast-port=9876)
   ```

2. Add the discovery group.

   ```
   /subsystem=messaging-activemq/server=default/discovery-group=my-discovery-group:add(socket-binding=messaging-group,refresh-timeout=10000)
   ```

Configure a Discovery Group Using JGroups

Below is an example configuration of a messaging server that defines a JGroups discovery group.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <discovery-group name="my-discovery-group" refresh-timeout="10000" jgroups-cluster="activemq-cluster"/>
    ...
  </server>
</subsystem>
```

This can be configured using the following management CLI command:

```
/subsystem=messaging-activemq/server=default/discovery-group=my-discovery-group:add(refresh-timeout=10000,jgroups-cluster=activemq-cluster)
```

Discovery Group Attributes

The below table lists the configurable attributes for a discovery group.

Attribute	Description
initial-wait-timeout	Period, in milliseconds, to wait for an initial broadcast to give us at least one node in the cluster.
jgroups-channel	The name of a channel defined in the **jgroups** subsystem that is used in combination with the **jgroups-cluster** attribute to form a cluster. If undefined, the **default** channel will be used. Note that a**jgroups-channel** multiplexes group communication between distinct logical groups, which are identified by the **jgroups-cluster** attribute.
jgroups-cluster	The logical name used for the communication between a broadcast group and a discovery group. A discovery group intending to receive messages from a particular broadcast group must use the same cluster name used by the broadcast group.
jgroups-stack	The name of a stack defined in the **jgroups** subsystem that is used to form a cluster. This attribute is deprecated. Use **jgroups-channel** to form a cluster instead. Since each **jgroups-stack** is already associated with a **jgroups-channel**, you can use that channel or you can create a new **jgroups-channel** and associate it with the**jgroups-stack**. IMPORTANT If a **jgroups-stack** and a **jgroups-channel** are both specified, a new **jgroups-channel** is generated and is registered in the same namespace as the **jgroups-stack** and **jgroups-channel**. For this reason, the **jgroups-stack** and **jgroup-channel** names must be unique.
refresh-timeout	Period the discovery group waits after receiving the last broadcast from a particular server before removing that server's connector pair entry from its list.
socket-binding	The discovery group socket binding.

WARNING

The JGroups attributes and UDP-specific attributes described above are exclusive of each other. Only one set can be specified in a discovery group configuration.

29.1.2.2. Configure Discovery Groups on the Client Side

You can use Jakarta Messaging or the core API to configure a JBoss EAP messaging client to discover a list of servers to which it can connect.

Configure Client Discovery using Jakarta Messaging

Clients using Jakarta Messaging can look up the relevant **ConnectionFactory** with JNDI. The **entries** attribute of a **connection-factory** or a **pooled-connection-factory** specifies the JNDI name under which the factory will be exposed. Below is an example of a **ConnectionFactory** configured for a remote client to lookup with JNDI:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <connection-factory name="RemoteConnectionFactory"
entries="java:jboss/exported/jms/RemoteConnectionFactory" connectors="http-connector"/>
    ...
  </server>
</subsystem>
```

NOTE

It is important to remember that only JNDI names bound in the **java:jboss/exported** namespace are available to remote clients. If a **connection-factory** has an entry bound in the **java:jboss/exported** namespace a remote client would look up the **connection-factory** using the text after **java:jboss/exported**. For example, the **RemoteConnectionFactory** is bound by default to **java:jboss/exported/jms/RemoteConnectionFactory** which means a remote client would look-up this **connection-factory** using **jms/RemoteConnectionFactory**. A **pooled-connection-factory** should not have any entry bound in the **java:jboss/exported** namespace because a **pooled-connection-factory** is not suitable for remote clients.

Since Jakarta Messaging 2.0, a default Jakarta Messaging connection factory is accessible to Jakarta EE applications under the JNDI name **java:comp/DefaultJMSConnectionFactory**. The JBoss EAP **messaging-activemq** subsystem defines a **pooled-connection-factory** that is used to provide this default connection factory. Any parameter change on this **pooled-connection-factory** will be taken into account by any Jakarta EE application looking the default Jakarta Messaging provider under the JNDI name **java:comp/DefaultJMSConnectionFactory**. Below is the default pooled connection factory as defined in the ***-full** and ***-full-ha** profiles.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <pooled-connection-factory name="activemq-ra" transaction="xa" entries="java:/JmsXA
java:jboss/DefaultJMSConnectionFactory" connectors="in-vm"/>
    ...
  </server>
</subsystem>
```

Configure Client Discovery using the Core API

If you are using the core API to directly instantiate **ClientSessionFactory** instances, then you can specify the discovery group parameters directly when creating the session factory. For example:

```
final String groupAddress = "231.7.7.7";
final int groupPort = 9876;

ServerLocator factory =
  ActiveMQClient.createServerLocatorWithHA(new DiscoveryGroupConfiguration(
      groupAddress,
      groupPort,
      new UDPBroadcastGroupConfiguration(groupAddress, groupPort, null, -1)));
ClientSessionFactory factory = locator.createSessionFactory();
ClientSession session1 = factory.createSession();
ClientSession session2 = factory.createSession();
```

You can use the **setDiscoveryRefreshTimeout()** setter method on the **DiscoveryGroupConfiguration** to set the **refresh-timeout** value, which defaults to **10000** milliseconds.

You can also use the **setDiscoveryInitialWaitTimeout()** setter method on the **DiscoveryGroupConfiguration** to set the **initial-wait-timeout** value, which determines how long the session factory will wait before creating the first session. The default value is **10000** milliseconds.

29.1.3. Static Discovery

In situations where you can not or do not want to use UDP on your network, you can configure a connection with an initial list of one or more servers.

This does not mean that you have to know where all your servers are going to be hosted. You can configure these servers to connect to a reliable server, and have their connection details propagated by way of that server.

Configuring a Cluster Connection
For cluster connections there, is no additional configuration needed, you just need to make sure that any connectors are defined in the usual manner. These are then referenced by the cluster connection configuration.

Configuring a Client Connection
A static list of possible servers can also be used by a client.

Configuring Client Discovery Using Jakarta Messaging
The recommended way to use static discovery with Jakarta Messaging is to configure a **connection-factory** with multiple connectors (each pointing to a unique node in the cluster) and have the client look up the ConnectionFactory using JNDI. Below is a snippet of configuration showing just such a **connection-factory**:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <connection-factory name="MyConnectionFactory" entries="..." connectors="http-connector http-
node1 http-node2"/>
    ...
  </server>
</subsystem>
```

In the above example, **http-connector** is an HTTP connector (**<http-connector>**) pointing to the local server, **http-node1** is an HTTP connector pointing to server **node1**, and so on. See the Connectors and Acceptors section for configuring connectors in the server configuration.

Configuring Client Discovery Using the Core API
If you are using the core API, create a unique **TransportConfiguration** for each server in the cluster and pass them into the method responsible for creating the **ServerLocator**, as in the below example code.

```
HashMap<String, Object> map = new HashMap<String, Object>();
map.put("host", "myhost");
map.put("port", "8080");

HashMap<String, Object> map2 = new HashMap<String, Object>();
map2.put("host", "myhost2");
map2.put("port", "8080");

TransportConfiguration server1 = new
TransportConfiguration(NettyConnectorFactory.class.getName(), map);
TransportConfiguration server2 = new
TransportConfiguration(NettyConnectorFactory.class.getName(), map2);

ServerLocator locator = ActiveMQClient.createServerLocatorWithHA(server1, server2);
ClientSessionFactory factory = locator.createSessionFactory();
ClientSession session = factory.createSession();
```

29.1.4. Default JGroups values

Previously, you had to review the **jgroups-defaults.xml** file to find the default JGroups values, which was time-consuming. For your review convenience, Red Hat listed the following default JGroups values:

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="urn:org:jgroups">
   <UDP
      ip_ttl="2"
      mcast_recv_buf_size="25m"
      mcast_send_buf_size="1m"
      ucast_recv_buf_size="20m"
      ucast_send_buf_size="1m"
      port_range="0"
   />
   <TCP
      send_buf_size="640k"
      sock_conn_timeout="300"
      port_range="0"
   />
   <TCP_NIO2
      send_buf_size="640k"
      sock_conn_timeout="300"
      port_range="0"
   />
   <TCPPING port_range="0" num_discovery_runs="4"/>
   <MPING ip_ttl="2"/>
   <kubernetes.KUBE_PING port_range="0"/>
   <MERGE3
      min_interval="10000"
      max_interval="30000"
   />
   <FD max_tries="5"
      msg_counts_as_heartbeat="false"
      timeout="3000"
   />
   <FD_ALL
      interval="15000"
      timeout="60000"
      timeout_check_interval="5000"/>
```

```
    <FD_SOCK/>
    <VERIFY_SUSPECT timeout="1000"/>
    <pbcast.NAKACK2
       xmit_interval="100"
       xmit_table_num_rows="50"
    />
    <UNICAST3
       xmit_interval="100"
       xmit_table_num_rows="50"
    />
    <pbcast.STABLE
       stability_delay="500"
       desired_avg_gossip="5000"
       max_bytes="1m"
    />
    <pbcast.GMS print_local_addr="false"/>
    <UFC max_credits="2m"/>
    <MFC max_credits="2m"/>
    <FRAG2 frag_size="30k"/>
</config>
```

29.2. SERVER-SIDE MESSAGE LOAD BALANCING

If a cluster connection is defined between nodes of a cluster, then JBoss EAP messaging will load balance messages arriving at a particular node from a client.

A messaging cluster connection can be configured to load balance messages in a round robin fashion, irrespective of whether there are any matching consumers on other nodes. It can also be configured to distribute to other nodes only if matching consumers exist. See the Message Redistribution section for more information.

Configuring the Cluster Connection

A cluster connection group servers into clusters so that messages can be load balanced between the nodes of the cluster. A cluster connection is defined in the JBoss EAP server configuration using the **cluster-connection** element.

WARNING

Red Hat supports using only one **cluster-connection** within the **messaging-activemq** subsystem.

Below is the default **cluster-connection** as defined in the ***-full** and ***-full-ha** profiles. See Cluster Connection Attributes for the complete list of attributes.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <cluster-connection name="my-cluster" discovery-group="dg-group1" connector-name="http-
connector" address="jms"/>
```

```
    ...
  </server>
</subsystem>
```

In the case shown above the cluster connection will load balance messages sent to addresses that start with "jms". This cluster connection will, in effect, apply to all Jakarta Messaging queues and topics since they map to core queues that start with the substring "jms".

NOTE

When a packet is sent using a cluster connection and is at a blocked state and waiting for acknowledgements, the **call-timeout** attribute specifies how long it will wait for the reply before throwing an exception. The default value is **30000**. In certain cases, for example, if the remote Jakarta Messaging broker is disconnected from network and the transaction is incomplete, the thread could remain stuck until connection is re-established. To avoid this situation, it is recommended to use the **call-failover-timeout** attribute along with the **call-timeout** attribute. The **call-failover-timeout** attribute is used when a call is made during a failover attempt. The default value is **-1**, which means no timeout. For more information on Client Failover, see Automatic Client Failover.

NOTE

Alternatively, if you would like the cluster connection to use a static list of servers for discovery then you can use the **static-connectors** attribute. For example:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <cluster-connection name="my-cluster" static-connectors="server0-connector
server1-connector" .../>
    ...
  </server>
</subsystem>
```

In this example, there are two servers defined where we know that at least one will be available. There may be many more servers in the cluster, but these will be discovered using one of these connectors once an initial connection has been made.

Configuring a Cluster Connection for Duplicate Detection

The cluster connection internally uses a core bridge to move messages between nodes of the cluster. To configure a cluster connection for duplicate message detection, set the **use-duplicate-detection** attribute to **true**, which is the default value.

```
/subsystem=messaging-activemq/server=default/cluster-connection=my-cluster:write-
attribute(name=use-duplicate-detection,value=true)
```

Cluster User Credentials

When creating connections between nodes of a cluster to form a cluster connection, JBoss EAP messaging uses a cluster user and password.

You can set the cluster user and password by using the following management CLI commands.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=cluster-
user,value="NewClusterUser")
```

```
/subsystem=messaging-activemq/server=default:write-attribute(name=cluster-
password,value="NewClusterPassword123")
```

This adds the following XML content to the **messaging-activemq** subsystem in the JBoss EAP configuration file.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <cluster user="NewClusterUser" password="NewClusterPassword123"/>
    ...
  </server>
</subsystem>
```

WARNING

The default value for **cluster-user** is **ACTIVEMQ.CLUSTER.ADMIN.USER** and the default value for **cluster-password** is **CHANGE ME!!**. It is imperative that these values are changed from their default, or remote clients will be able to make connections to the server using the default values. If they are not changed from the default, JBoss EAP messaging will detect this and display a warning upon every startup.

NOTE

You can also use the **cluster-credential-reference** attribute to reference a credential store instead of setting a cluster password.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=cluster-
credential-reference,value={clear-text=SecretStorePassword})
```

29.3. CLIENT-SIDE LOAD BALANCING

With JBoss EAP messaging client-side load balancing, subsequent sessions created using a single session factory can be connected to different nodes of the cluster. This allows sessions to spread smoothly across the nodes of a cluster and not be clumped on any particular node.

The recommended way to declare a load balancing policy to be used by the client factory is to set the **connection-load-balancing-policy-class-name** attribute of the **<connection-factory>** resource. JBoss EAP messaging provides the following out-of-the-box load balancing policies, and you can also implement your own.

Round robin

With this policy, the first node is chosen randomly then each subsequent node is chosen sequentially in the same order.
For example, nodes might be chosen in the order **B**, **C**, **D**, **A**, **B**, **C**, **D**, **A**, **B** or **D**, **A**, **B**, **C**, **D**, **A**, **B**, **C**.

Use **org.apache.activemq.artemis.api.core.client.loadbalance.RoundRobinConnectionLoadBalancingPolicy** as the **connection-load-balancing-policy-class-name** .

Random

With this policy, each node is chosen randomly.
Use **org.apache.activemq.artemis.api.core.client.loadbalance.RandomConnectionLoadBalancingPolicy** as the **connection-load-balancing-policy-class-name** .

Random Sticky

With this policy, the first node is chosen randomly and then reused for subsequent connections.
Use **org.apache.activemq.artemis.api.core.client.loadbalance.RandomStickyConnectionLoadBalancingPolicy** as the **connection-load-balancing-policy-class-name** .

First Element

With this policy, the first, or 0th, node is always returned.
Use **org.apache.activemq.artemis.api.core.client.loadbalance.FirstElementConnectionLoadBalancingPolicy** as the **connection-load-balancing-policy-class-name** .

You can also implement your own policy by implementing the interface **org.apache.activemq.artemis.api.core.client.loadbalance.ConnectionLoadBalancingPolicy**

29.4. MESSAGE REDISTRIBUTION

With message redistribution, JBoss EAP messaging can be configured to automatically redistribute messages from queues which have no consumers back to other nodes in the cluster which do have matching consumers. To enable this functionality, cluster connection's **message-load-balancing-type** must be set to **ON_DEMAND**, which is the default value. You can set this using the following management CLI command.

```
/subsystem=messaging-activemq/server=default/cluster-connection=my-cluster:write-attribute(name=message-load-balancing-type,value=ON_DEMAND)
```

Message redistribution can be configured to kick in immediately after the last consumer on a queue is closed, or to wait a configurable delay after the last consumer on a queue is closed before redistributing. This is configured using the **redistribution-delay** attribute.

You use the **redistribution-delay** attribute to set how many milliseconds to wait after the last consumer is closed on a queue before redistributing messages from that queue to other nodes of the cluster that have matching consumers. A value of **-1**, which is the default value, means that messages will never be redistributed. A value of **0** means that messages will be immediately redistributed.

The **address-setting** in the default JBoss EAP configuration sets a **redistribution-delay** value of **1000**, meaning that it will wait 1000 milliseconds before redistributing messages.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <address-setting name="#" redistribution-delay="1000" message-counter-history-day-limit="10"
```

```
page-size-bytes="2097152" max-size-bytes="10485760" expiry-address="jms.queue.ExpiryQueue"
dead-letter-address="jms.queue.DLQ"/>
    ...
  </server>
</subsystem>
```

It often makes sense to introduce a delay before redistributing as it is a common case that a consumer closes but another one quickly is created on the same queue. In this case, you may not want to redistribute immediately since the new consumer will arrive shortly.

Below is an example of an **address-setting** that sets a **redistribution-delay** of **0** for any queue or topic that is bound to an address that starts with "jms.". In this case, messages will be redistributed immediately.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  <server name="default">
    ...
    <address-setting name="jms.#" redistribution-delay="0"/>
    ...
  </server>
</subsystem>
```

This address setting can be added using the following management CLI command.

```
/subsystem=messaging-activemq/server=default/address-setting=jms.#:add(redistribution-
delay=1000)
```

29.5. CLUSTERED MESSAGE GROUPING

IMPORTANT

This feature is not supported.

Clustered grouping follows a different approach relative to normal message grouping. In a cluster, message groups with specific group ids can arrive on any of the nodes. It is important for a node to determine which group ids are bound to which consumer on which node. Each node is responsible for routing message groups correctly to the node which has the consumer processing those group ids, irrespective of where the message groups arrive by default. Once messages with a given group id are sent to a specific consumer connected to the given node in the cluster, then those messages are never sent to another node even if the consumer is disconnected.

This situation is addressed by a grouping handler. Each node has a grouping handler and this grouping handler (along with other handlers) is responsible for routing the message groups to the correct node. There are two types of grouping handlers: **LOCAL** and **REMOTE**.

The local handler is responsible for deciding the route that a message group should take. The remote handlers communicate with the local handler and work accordingly. Each cluster should choose a specific node to have a local grouping handler and all the other nodes should have remote handlers.

WARNING

If message grouping is used in a cluster, it will break if a node configured as a remote grouping handler fails. Setting up a backup for the remote grouping handler will not correct this.

The node that initially receives a message group takes the routing decision based on regular cluster routing conditions (round-robin queue availability). The node proposes this decision to the respective grouping handler which then routes the messages to the proposed queue if it accepts the proposal.

If the grouping handler rejects the proposal, it proposes some other route and the routing takes place accordingly. The other nodes follow suite and forward the message groups to the chosen queue. After a message arrives on a queue, it is pinned to a customer on that queue.

You can configure grouping handlers using the management CLI. The following command adds a **LOCAL** grouping handler with the address **news.europe.#**.

```
/subsystem=messaging-activemq/server=default/grouping-handler=my-group-handler:add(grouping-handler-address="news.europe.#",type=LOCAL)
```

This will require a server reload.

```
reload
```

The below table lists the configurable attributes for a **grouping-handler**.

Attribute	Description
group-timeout	With a **REMOTE** handler, this value specifies how often the**REMOTE** will notify the **LOCAL** that the route was used. With a**LOCAL** handler, if a route is not used for the time specified, it is removed, and a new path would need to be established. The value is in milliseconds.
grouping-handler-address	A reference to a cluster connection and the address it uses.
reaper-period	How often the reaper will be run to check for timed out group bindings (only valid for **LOCAL** handlers).
timeout	How long to wait for a handling decision to be made; an exception will be thrown during the send if this timeout is reached, ensuring that strict ordering is kept.
type	Whether the handler is the single local handler for the cluster, which makes handling decisions, or a remote handler which converses with the local handler. Possible values are **LOCAL** or **REMOTE**.

29.5.1. Best Practices for Clustered Message Grouping

Some best practices for clustered grouping are as follows:

- If you create and close consumers regularly, make sure that your consumers are distributed evenly across the different nodes. Once a queue is pinned, messages are automatically transferred to that queue regardless of removing customers from it.
- If you wish to remove a queue that has a message group bound to it, make sure the queue is deleted by the session that is sending the messages. Doing this will ensure that other nodes will not try to route messages to this queue after it is removed.
- As a failover mechanism, always replicate the node that has the local grouping handler.

29.6. STARTING AND STOPPING MESSAGING CLUSTERS

When you configure JBoss EAP 7.4 servers to form an ActiveMQ Artemis cluster, there can be other servers and clients that are connected to the running clustered servers. It is recommended to shutdown the connected clients and servers first, before shutting down the JBoss EAP 7.4 servers that are running in the cluster. This must be done in sequence and not in parallel in order to provide enough time for the servers to close all connections and avoid failures during closing that might lead to inconsistent states. ActiveMQ Artemis does not support automatic scale down of cluster nodes and expects that all cluster nodes will be restarted.

The same holds true when starting the servers. You must first start the JBoss EAP 7.4 servers in the ActiveMQ Artemis cluster. When startup is complete, you can then start the other servers and clients that connect to the cluster.

CHAPTER 30. HIGH AVAILABILITY

High availability is the ability for the system to continue functioning after failure of one or more of the servers.

A part of high availability is failover which is the ability for client connections to migrate from one server to another in event of server failure so client applications can continue to operate.

NOTE

Only persistent message data will survive failover. Any non persistent message data will not be available after failover.

30.1. LIVE / BACKUP PAIRS

JBoss EAP 7 messaging allows servers to be linked together as live - backup pairs where each live server has a backup. Live servers receive messages from clients, while a backup server is not operational until failover occurs. A backup server can be owned by only one live server, and it will remain in passive mode, waiting to take over the live server's work.

NOTE

There is a one-to-one relation between a live server and a backup server. A live server can have only *one* backup server, and a backup server can be owned by only *one* live server.

When a live server crashes or is brought down in the correct mode, the backup server currently in passive mode will become the new live server. If the new live server is configured to allow automatic failback, it will detect the old live server coming back up and automatically stop, allowing the old live server to start receiving messages again.

NOTE

If you deploy just one pair of live / backup servers, you cannot effectively use a load balancer in front of the pair because the backup instance is not actively processing messages. Moreover, services such as JNDI and the Undertow web server are not active on the backup server either. For these reasons, deploying JEE applications to an instance of JBoss EAP being used as a backup messaging server is not supported.

30.1.1. Journal Synchronization

When HA is configured with a replicated journal, it takes time for the backup to synchronize with live server.

To check whether synchronization is complete, submit the following command in the CLI:

```
/subsystem=messaging-activemq/server=default/ha-policy=replication-master:read-
attribute(name=synchronized-with-backup)
```

If the result is **true**, synchronization is complete.

To check whether it is safe to shut down the live server, submit the following command in the CLI:

```
/subsystem=messaging-activemq/server=default/ha-policy=replication-slave:read-attribute(name=synchronized-with-live)
```

If the result is **true**, it is safe to shut down the live server.

30.2. HA POLICIES

JBoss EAP messaging supports two different strategies for backing up a server: replication and shared store. Use the **ha-policy** attribute of the **server** configuration element to assign the policy of your choice to the given server. There are four valid values for **ha-policy**:

- **replication-master**
- **replication-slave**
- **shared-store-master**
- **shared-store-slave**

As you can see, the value specifies whether the server uses a data replication or a shared store ha policy, and whether it takes the role of master or slave.

Use the management CLI to add an **ha-policy** to the server of your choice.

NOTE

The examples below assume you are running JBoss EAP using the **standalone-full-ha** configuration profile.

```
/subsystem=messaging-activemq/server=SERVER/ha-policy=POLICY:add
```

For example, use the following command to add the **replication-master** policy to the **default** server.

```
/subsystem=messaging-activemq/server=default/ha-policy=replication-master:add
```

The **replication-master** policy is configured with the default values. Values to override the default configuration can be included when you add the policy. The management CLI command to read the current configuration uses the following basic syntax.

```
/subsystem=messaging-activemq/server=SERVER/ha-policy=POLICY:read-resource
```

For example, use the following command to read the current configuration for the **replication-master** policy that was added above to the **default** server. The output is also is also included to highlight the default configuration.

```
/subsystem=messaging-activemq/server=default/ha-policy=replication-master:read-resource

{
    "outcome" => "success",
    "result" => {
        "check-for-live-server" => true,
        "cluster-name" => undefined,
        "group-name" => undefined,
        "initial-replication-sync-timeout" => 30000L
    }
}
```

See Data Replication and Shared Store for details on the configuration options available for each policy.

30.3. DATA REPLICATION

When using replication, the live and the backup server pairs do not share the same data directories, all data synchronization is done over the network. Therefore all (persistent) data received by the live server will be duplicated to the backup.

If the live server is cleanly shut down, the backup server will activate and clients will failover to backup. This behavior is pre-determined and is therefore not configurable when using data replication.

The backup server will first need to synchronize all existing data from the live server before replacing it. Unlike shared storage, therefore, a replicating backup will not be fully operational immediately after startup. The time it will take for the synchronization to happen depends on the amount of data to be synchronized and the network speed. Also note that clients are blocked for the duration of **initial-replication-sync-timeout** when the backup is started. After this timeout elapses, clients will be unblocked, even if synchronization is not completed.

After a successful failover, the backup's journal will start holding newer data than the data on the live server. You can configure the original live server to perform a failback and become the live server once restarted. A failback will synchronize data between the backup and the live server before the live server comes back online.

In cases were both servers are shut down, the administrator will have to determine which server's journal has the latest data. If the backup journal has the latest data, copy that journal to the live server. Otherwise, whenever it activates again, the backup will replicate the stale journal data from the live server and will delete its own journal data. If the live server's data is the latest, no action is needed and the servers can be started normally.

IMPORTANT

Due to higher latencies and a potentially unreliable network between data centers, the configuration and use of replicated journals for high availability between data centers is neither recommended nor supported.

The replicating live and backup pair must be part of a cluster. The **cluster-connection** configuration element defines how a backup server finds its live match.

Replication requires at least three live/backup pairs to reduce the risk of network isolation, although you cannot eliminate this risk. If you use at least three live/backup pairs, the cluster can use quorum voting to avoid using two live brokers.

When you configure **cluster-connection**, remember the following details:

- Both the live and backup server must be part of the same cluster. Notice that even a simple live/backup replicating pair requires a cluster configuration.
- The cluster user and password must match on each server in the pair.

Specify a pair of live/backup servers by configuring the **group-name** attribute in both the **<master>** and the **<slave>** elements. A backup server only connects to a live server that shares the same **group-name**.

As an example of using a **group-name**, suppose you have three live servers and three backup servers. Because each live server must pair with its own backup, assign the following group names:

- live1 and backup1 use the **group-name** of **pair1**.
- live2 and backup2 use the **group-name** of **pair2**.
- live3 and backup3 use the **group-name** of **pair3**.

In this example, server **backup1** searches for the live server with the same **group-name**, **pair1**, which in this case is the server **live1**.

Much like in the shared store case, when the live server stops or crashes, its replicating, paired backup will become active and take over its duties. Specifically, the paired backup will become active when it loses connection to its live server. This can be problematic because this can also happen because of a temporary network problem. In order to address this issue, the paired backup will try to determine whether it still can connect to the other servers in the cluster. If it can connect to more than half the servers, it will become active. If it loses communication to its live server plus more than half the other servers in the cluster, the paired backup will wait and try reconnecting with the live server. This reduces the risk of a "split brain" situation where both the backup and live servers are processing messages without the other knowing it.

IMPORTANT

This is an important distinction from a shared store backup, where the backup will activate and start to serve client requests if it does not find a live server and the file lock on the journal was released. Note also that in replication the backup server does not know whether any data it might have is up to date, so it really cannot decide to activate automatically. To activate a replicating backup server using the data it has, the administrator must change its configuration to make it a live server by changing slave to master.

Additional resources

- Configuring Cluster Connections

30.3.1. Configuring Data Replication

Below are two examples showing the basic configuration for both a live and a backup server residing in the cluster named **my-cluster** and in the backup group named **group1**.

The steps below use the management CLI to provide a basic configuration for both a live and a backup server residing in the cluster named **my-cluster** and in the backup group named **group1**.

NOTE

The examples below assume you are running JBoss EAP using the **standalone-full-ha** configuration profile.

Management CLI Commands to Configure a Live Server for Data Replication

1. Add the **ha-policy** to the Live Server

   ```
   /subsystem=messaging-activemq/server=default/ha-policy=replication-master:add(check-for-live-server=true,cluster-name=my-cluster,group-name=group1)
   ```

   The **check-for-live-server** attribute tells the live server to check to make sure that no other server has its given id within the cluster. The default value for this attribute was **false** in JBoss EAP 7.0. In JBoss EAP 7.1 and later, the default value is **true**.

2. Add the **ha-policy** to the Backup Server

   ```
   /subsystem=messaging-activemq/server=default/ha-policy=replication-slave:add(cluster-name=my-cluster,group-name=group1)
   ```

3. Confirm a shared **cluster-connection** exists.
   Proper communication between the live and backup servers requires a **cluster-connection**. Use the following management CLI command to confirm that the same **cluster-connection** is configured on both the live and backup servers. The example uses the default **cluster-connection** found in the **standalone-full-ha** configuration profile, which should be sufficient for most use cases. See Configuring Cluster Connections for details on how to configure a cluster connection.

   Use the following management CLI command to confirm that both the live and backup server are using the same cluster-connection.

   ```
   /subsystem=messaging-activemq/server=default/cluster-connection=my-cluster:read-resource
   ```

   If the **cluster-connection** exists, the output will provide the current configuration. Otherwise an error message will be displayed.

See All Replication Configuration for details on all configuration attributes.

30.3.2. All Replication Configuration

You can use the management CLI to add configuration to a policy after it has been added. The commands to do so follow the basic syntax below.

```
/subsystem=messaging-activemq/server=default/ha-policy=POLICY:write-attribute(name=ATTRIBUTE,value=VALUE)
```

For example, to set the value of the **restart-backup** attribute to **true**, use the following command.

```
/subsystem=messaging-activemq/server=default/ha-policy=replication-slave:write-attribute(name=restart-backup,value=true)
```

The following tables provide the HA configuration attributes for the **replication-master** node and **replication-slave** configuration elements.

Table 30.1. Attributes for replication-master

Attribute	Description
check-for-live-server	Set to **true** to tell this server to check the cluster for another server using the same server ID when starting up. The default value for JBoss EAP 7.0 is **false**. The default value for JBoss EAP 7.1 and later is **true**.
cluster-name	Name of the cluster used for replication.
group-name	If set, backup servers will only pair with live servers with the matching **group-name**.
initial-replication-sync-timeout	How long to wait in milliseconds until the initiation replication is synchronized. Default is **30000**.
synchronized-with-backup	Indicates whether the journals on the live server and the replication server have been synchronized.

Table 30.2. Attributes for **replication-slave**

Attribute	Description
allow-failback	Whether this server will automatically stop when another places a request to take over its place. A typical use case is when live server requests to resume active processing after a restart or failure recovery. A backup server with **allow-failback** set to **true** would yield to the live server once it rejoined the cluster and requested to resume processing. Default is **true**.
cluster-name	Name of the cluster used for replication.
group-name	If set, backup servers will pair only with live servers with the matching **group-name**.
initial-replication-sync-timeout	How long to wait in milliseconds until the initiation replication is synchronized. Default is **30000**.
max-saved-replicated-journal-size	Specifies how many times a replicated backup server can restart after moving its files on start. After reaching the maximum, the server will stop permanently after if fails back. Default is **2**.
restart-backup	Set to **true** to tell this backup server to restart once it has been stopped because of failback. Default is **true**.

Attribute	Description
synchronized-with-live	Indicates whether the journals on the replication server have been synchronized with the live server, meaning it is safe to shut down the live server.

30.3.3. Preventing Cluster Connection Timeouts

Each live and backup pair uses a **cluster-connection** to communicate. The **call-timeout** attribute of a **cluster-connection** sets the amount of a time a server will wait for a response after making a call to another server on the cluster. The default value for **call-timeout** is 30 seconds, which is sufficient for most use cases. However, there are situations where the backup server might be unable to process replication packets coming from the live server. This may happen, for example, when the initial pre-creation of journal files takes too much time, due to slow disk operations or to a large value for **journal-min-files**. If timeouts like this occur you will see a line in your logs similar to the one below.

> AMQ222207: The backup server is not responding promptly introducing latency beyond the limit. Replication server being disconnected now.

WARNING

If a line like the one above appears in your logs that means that the replication process has stopped. You must restart the backup server to reinitiate replication.

To prevent cluster connection timeouts, consider the following options:

- Increase the **call-timeout** of the **cluster-connection**. See Configuring Cluster Connections for more information.
- Decrease the value of **journal-min-files**. See Configuring Persistence for more information.

30.3.4. Removing Old Journal Directories

A backup server will move its journals to a new location when it starts to synchronize with a live server. By default the journal directories are located in **data/activemq** directory under **EAP_HOME/standalone**. For domains, each server will have its own **serverX/data/activemq** directory located under **EAP_HOME/domain/servers**. The directories are named **bindings**, **journal**, **largemessages** and **paging**. See Configuring Persistence and Configuring Paging for more information about these directories.

Once moved, the new directories are renamed **oldreplica.X**, where **X** is a digit suffix. If another synchronization starts due to a new failover then the suffix for the "moved" directories will be increased by 1. For example, on the first synchronization the journal directories will be moved to **oldreplica.1**, on the second, **oldreplica.2**, and so on. The original directories will store the data synchronized from the live server.

By default a backup server is configured to manage two occurrences of failing over and failing back. After that a cleanup process triggers that removes the **oldreplica.X** directories. You can change the number of failover occurrences that trigger the cleanup process using the **max-saved-replicated-journal-size** attribute on the backup server.

NOTE

Live servers will have **max-saved-replicated-journal-size** set to **2**. This value cannot be changed

30.3.5. Updating Dedicated Live and Backup Servers

If the live and backup servers are deployed in a dedicated topology, where each server is running in its own instance of JBoss EAP, follow the steps below to ensure a smooth update and restart of the cluster.

1. Cleanly shut down the backup servers.
2. Cleanly shut down the live servers.
3. Update the configuration of the live and backup servers.
4. Start the live servers.
5. Start the backup servers.

30.3.6. Detecting network isolation of the broker

To detect network isolation of the broker, you can ping a configurable list of hosts. Use one of the following parameters to configure how the status of the broker on the network is detected:

- **network-check-NIC**: Denotes the Network Interface Controller (NIC) to be used in the **InetAddress.isReachable** method to check network availability.
- **network-check-period**: Denotes a frequency in milliseconds that defines how often the network status is checked.
- **network-check-timeout**: Denotes a waiting time period before a network connection is expired.
- **network-check-list**: Denotes the list of IP addresses that are pinged to detect the network status.
- **network-check-URL-list**: Denotes the list of http URIs that are used to validate the network.
- **network-check-ping-command**: Denotes the ping command and its parameters that are used to detect the network status on an IPv4 network.
- **network-check-ping6-command**: Denotes the ping command and its parameters that are used to detect the network status on an IPv6 network.

Procedure

- Use the following command to ping a configurable list of hosts to detect network isolation of the broker:

```
/subsystem=messaging-activemq/server=default:write-attribute(name=<parameter-name>,
value="<ip-address>")
```

Example

To check the network status by pinging the IP address **10.0.0.1**, issue the following command:

```
/subsystem=messaging-activemq/server=default:write-attribute(name=network-check-list,
value="10.0.0.1")
```

30.3.7. Limitations of Data Replication: Split Brain Processing

A "split brain" situation occurs when both a live server and its backup are active at the same time. Both servers can serve clients and process messages without the other knowing it. In this situation there is no longer any message replication between the live and backup servers. A split situation can happen if there is network failure between the two servers.

For example, if the connection between a live server and a network router is broken, the backup server will lose the connection to the live server. However, because the backup can still can connect to more than half the servers in the cluster, it becomes active. Recall that a backup will also activate if there is just one live-backup pair and the backup server loses connectivity to the live server. When both servers are active within the cluster, two undesired situations can happen:

1. Remote clients fail over to the backup server, but local clients such as MDBs will use the live server. Both nodes will have completely different journals, resulting in split brain processing.

2. The broken connection to the live server is fixed after remote clients have already failed over to the backup server. Any new clients will be connected to the live server while old clients continue to use the backup, which also results in a split brain scenario.

Customers should implement a reliable network between each pair of live and backup servers to reduce the risk of split brain processing when using data replication. For example, use duplicated Network Interface Cards and other network redundancies.

30.4. SHARED STORE

This style of high availability differs from data replication in that it requires a shared file system which is accessible by both the live and backup node. This means that the server pairs use the same location for their paging, message journal, bindings journal, and large messages in their configuration.

NOTE

Using a shared store is not supported on Windows. It is supported on Red Hat Enterprise Linux when using Red Hat versions of GFS2 or NFSv4. In addition, GFS2 is supported only with an ASYNCIO journal type, while NFSv4 is supported with both ASYNCIO and NIO journal types.

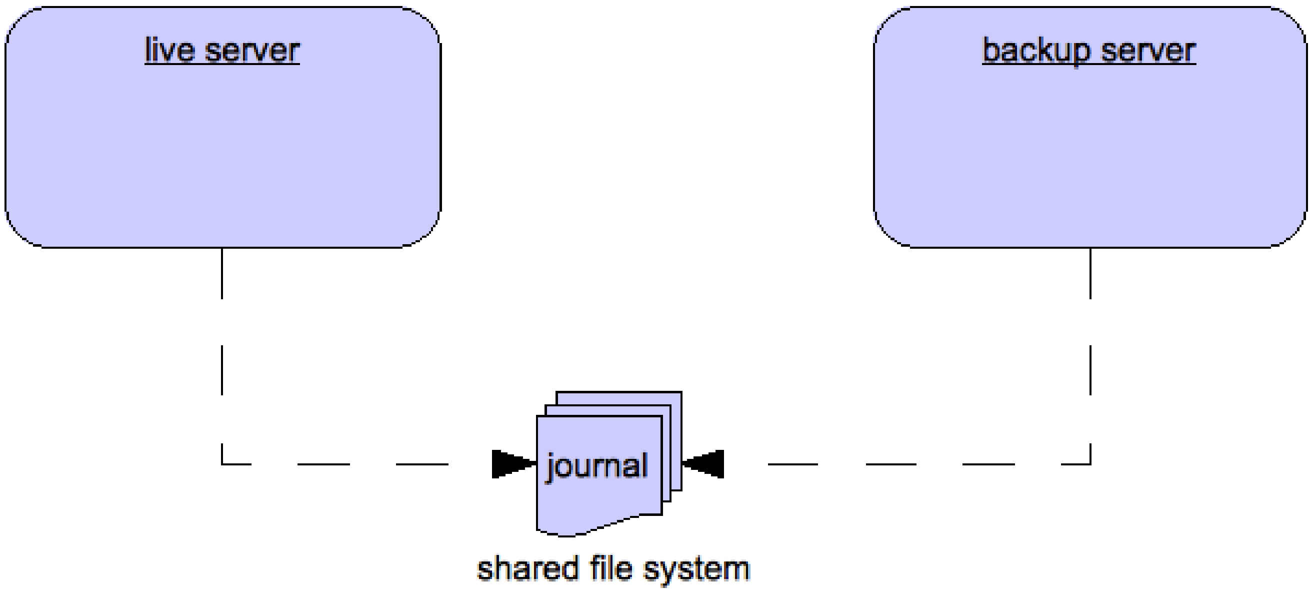
Also, each participating server in the pair, live and backup, will need to have a **cluster-connection** defined, even if not part of a cluster, because the **cluster-connection** defines how the backup server announces its presence to its live server and any other nodes. See Configuring Cluster Connections for details on how this is done.

When failover occurs and a backup server takes over, it will need to load the persistent storage from the shared file system before clients can connect to it. This style of high availability differs from data replication in that it requires a shared file system which is accessible by both the live and backup pair. Typically this will be some kind of high performance Storage Area Network, or SAN. Red Hat does not recommend using Network Attached Storage, known as a NAS, for your storage solution.

The advantage of shared store high availability is that no replication occurs between the live and backup nodes, this means it does not suffer any performance penalties due to the overhead of replication during normal operation.

The disadvantage of shared store replication is that when the backup server activates it needs to load the journal from the shared store which can take some time depending on the amount of data in the store. Also, it requires a shared storage solution supported by JBoss EAP.

If you require the highest performance during normal operation, Red Hat recommends having access to a highly performant SAN and accept the slightly slower failover costs. Exact costs will depend on the amount of data.



30.4.1. Configuring a Shared Store

NOTE

The examples below assume you are running JBoss EAP using the **standalone-full-ha** configuration profile.

1. Add the **ha-policy** to the Live Server.

```
/subsystem=messaging-activemq/server=default/ha-policy=shared-store-master:add
```

2. Add the **ha-policy** to the Backup Server.

```
/subsystem=messaging-activemq/server=default/ha-policy=shared-store-slave:add
```

3. Confirm a shared **cluster-connection** exists.
Proper communication between the live and backup servers requires a **cluster-connection**. Use the following management CLI command to confirm that the same **cluster-connection** is configured on both the live and backup servers. The example uses the default **cluster-connection** found in the **standalone-full-ha** configuration profile, which should be sufficient for most use cases. See Configuring Cluster Connections for details on how to configure a cluster connection.

```
/subsystem=messaging-activemq/server=default/cluster-connection=my-cluster:read-
resource
```

If the **cluster-connection** exists, the output will provide the current configuration. Otherwise an error message will be displayed.

See All Shared Store Configuration for details on all configuration attributes for shared store policies.

30.4.2. All Shared Store Configuration

Use the management CLI to add configuration to a policy after it has been added. The commands to do so follow the basic syntax below.

```
/subsystem=messaging-activemq/server=default/ha-policy=POLICY:write-
attribute(name=ATTRIBUTE,value=VALUE)
```

For example, to set the value of the **restart-backup** attribute to **true**, use the following command.

```
/subsystem=messaging-activemq/server=default/ha-policy=shared-store-slave:write-
attribute(name=restart-backup,value=true)
```

Table 30.3. Attributes of the shared-store-master Configuration Element

Attribute	Description
failover-on-server-shutdown	Set to **true** to tell this server to failover when it is normally shut down. Default is **false**.

Table 30.4. Attributes of the shared-store-slave Configuration Element

Attribute	Description
allow-failback	Set to **true** to tell this server to automatically stop when another places a request to take over its place. The use case is when a regular server stops and its backup takes over its duties, later the main server restarts and requests the server (the former backup) to stop operating. Default is **true**.
failover-on-server-shutdown	Set to **true** to tell this server to failover when it is normally shut down. Default is **false**.
restart-backup	Set to **true** to tell this server to restart once it has been stopped because of failback or scaling down. Default is **true**.

30.5. FAILING BACK TO A LIVE SERVER

After a live server has failed and a backup taken has taken over its duties, you may want to restart the live server and have clients fail back to it.

In case of a shared store, simply restart the original live server and kill the new live server by killing the process itself. Alternatively, you can set **allow-fail-back** to **true** on the slave which will force it to automatically stop once the master is back online. The management CLI command to set **allow-fail-back** looks like the following:

```
/subsystem=messaging-activemq/server=default/ha-policy=shared-store-slave:write-attribute(name=allow-fail-back,value=true)
```

In replication HA mode you need make sure the **check-for-live-server** attribute is set to **true** in the master configuration. Starting with JBoss EAP 7.1, this is the default value.

```
/subsystem=messaging-activemq/server=default/ha-policy=replication-master:write-attribute(name=check-for-live-server,value=true)
```

If set to **true**, a live server will search the cluster during startup for another server using its nodeID. If it finds one, it will contact this server and try to "fail-back". Since this is a remote replication scenario, the original live server will have to synchronize its data with the backup running with its ID. Once they are in sync, it will request the backup server to shut down so it can take over active processing. This behavior allows the original live server to determine whether there was a fail-over, and if so whether the server that took its duties is still running or not.

WARNING

Be aware that if you restart a live server after the failover to backup has occurred, then the **check-for-live-server** attribute must be set to **true**. If not, then the live server will start at once without checking that its backup server is running. This results in a situation in which the live and backup are running at the same time, causing the delivery of duplicate messages to all newly connected clients.

For shared stores, it is also possible to cause failover to occur on normal server shut down, to enable this set **failover-on-server-shutdown** to **true** in the HA configuration on either the master or slave like so:

```
/subsystem=messaging-activemq/server=default/ha-policy=shared-store-slave:write-attribute(name=failover-on-server-shutdown,value=true)
```

You can also force the running backup server to shut down when the original live server comes back up, allowing the original live server to take over automatically, by setting **allow-failback** to **true**.

```
/subsystem=messaging-activemq/server=default/ha-policy=shared-store-slave:write-attribute(name=allow-failback,value=true)
```

30.6. COLOCATED BACKUP SERVERS

JBoss EAP also makes it possible to colocate backup messaging servers in the same JVM as another live server. Take for example a simple two node cluster of standalone servers where each live server colocates the backup for the other. You can use either a shared store or a replicated HA policy when colocating servers in this way. There are two important things to remember when configuring messaging servers for colocation.

First, each **server** element in the configuration will need its own **remote-connector** and **remote-acceptor** or **http-connector** and **http-acceptor**. For example, a live server with a **remote-acceptor** can be configured to listen on port **5445**, while a **remote-acceptor** from a colocated backup uses port **5446**. The ports are defined in **socket-binding** elements that must be added to the default **socket-binding-group**. In the case of **http-acceptors**, the live and colocated backup can share the same **http-listener**. Cluster-related configuration elements in each **server** configuration will use the **remote-connector** or **http-connector** used by the server. The relevant configuration is included in each of the examples that follow.

Second, remember to properly configure paths for journal related directories. For example, in a shared store colocated topology, both the live server and its backup, colocated on another live server, must be configured to share directory locations for the binding and message journals, for large messages, and for paging.

30.6.1. Configuring Manual Creation of a Colocated HA Topology

The example management CLI commands used in the steps below illustrate how to configure a simple two node cluster employing a colocated topology. The example configures a two node colocated cluster. A live server and a backup server will live on each node. The colocated backup on *node one* is paired with the live server colocated on *node two*, and the backup server on *node two* is be paired with the live server on *node one*. Examples are included for both a shared store and a data replication HA policy.

NOTE

The examples below assume you are running JBoss EAP using the **full-ha** configuration profile.

1. Modify the default server on each instance to use an HA policy. The default server on each node will become the live server. The instructions you follow depend on whether you have configured a shared store policy or a data replication policy.

   - *Instructions for a shared store policy:* Use the following management CLI command to add the preferred HA policy.

     ```
     /subsystem=messaging-activemq/server=default/ha-policy=shared-store-master:add
     ```

   - *Instructions for a data replication policy:* The default server on each node should be configured with a unique **group-name**. In the following example, the first command is executed on *node one*, and the second on *node two*.

     ```
     /subsystem=messaging-activemq/server=default/ha-policy=replication-
     master:add(cluster-name=my-cluster,group-name=group1,check-for-live-server=true)

     /subsystem=messaging-activemq/server=default/ha-policy=replication-
     master:add(cluster-name=my-cluster,group-name=group2,check-for-live-server=true)
     ```

2. Colocate a new backup server with each live server.

   a. Add a new server to each instance of JBoss EAP to colocate with the default live server. The new server will backup the default server on the other node. Use the following management CLI command to create a new server named **backup**.

     ```
     /subsystem=messaging-activemq/server=backup:add
     ```

b. Next, configure the new server to use the preferred HA policy. The instructions you follow depend on whether you have configured a shared store policy or a data replication policy.

   - *Instructions for a shared store policy:* Use the following management CLI command to add the HA policy:

     ```
     /subsystem=messaging-activemq/server=backup/ha-policy=shared-store-slave:add
     ```

   - *Instructions for a data replication policy:* Configure the backup servers to use the **group-name** of the live server on the other node. In the following example, the first command is executed on *node one*, and the second command is executed on *node two*.

     ```
     /subsystem=messaging-activemq/server=backup/ha-policy=replication-slave:add(cluster-name=my-cluster,group-name=group2)

     /subsystem=messaging-activemq/server=backup/ha-policy=replication-slave:add(cluster-name=my-cluster,group-name=group1)
     ```

3. Configure the directory locations for all servers.
   Once the servers are configured for HA, you must configure the locations for the binding journal, message journal, and large messages directory. If you plan to use paging, you must also configure the paging directory. The instructions you follow depend on whether you have configured a shared store policy or a data replication policy.

   - *Instructions for a shared store policy:* The **path** values for the live server on *node one* should point to the same location on a supported file system as the backup server on *node two*. The same is true for the live server on *node two* and its backup on *node one*.

     a. Use the following management CLI commands to configure the directory locations for *node one*:

        ```
        /subsystem=messaging-activemq/server=default/path=bindings-directory:write-attribute(name=path,value=/PATH/TO/shared/bindings-A)

        /subsystem=messaging-activemq/server=default/path=journal-directory:write-attribute(name=path,value=/PATH/TO/shared/journal-A)

        /subsystem=messaging-activemq/server=default/path=large-messages-directory:write-attribute(name=path,value=/PATH/TO/shared/largemessages-A)

        /subsystem=messaging-activemq/server=default/path=paging-directory:write-attribute(name=path,value=/PATH/TO/shared/paging-A)

        /subsystem=messaging-activemq/server=backup/path=bindings-directory:write-attribute(name=path,value=/PATH/TO/shared/bindings-B)

        /subsystem=messaging-activemq/server=backup/path=journal-directory:write-attribute(name=path,value=/PATH/TO/shared/journal-B)

        /subsystem=messaging-activemq/server=backup/path=large-messages-directory:write-attribute(name=path,value=/PATH/TO/shared/largemessages-B)

        /subsystem=messaging-activemq/server=backup/path=paging-directory:write-attribute(name=path,value=/PATH/TO/shared/paging-B)
        ```

b. Use the following management CLI commands to configure the directory locations for *node two*:

```
/subsystem=messaging-activemq/server=default/path=bindings-directory:write-attribute(name=path,value=/PATH/TO/shared/bindings-B)

/subsystem=messaging-activemq/server=default/path=journal-directory:write-attribute(name=path,value=/PATH/TO/shared/journal-B)

/subsystem=messaging-activemq/server=default/path=large-messages-directory:write-attribute(name=path,value=/PATH/TO/shared/largemessages-B)

/subsystem=messaging-activemq/server=default/path=paging-directory:write-attribute(name=path,value=/PATH/TO/shared/paging-B)

/subsystem=messaging-activemq/server=backup/path=bindings-directory:write-attribute(name=path,value=/PATH/TO/shared/bindings-A)

/subsystem=messaging-activemq/server=backup/path=journal-directory:write-attribute(name=path,value=/PATH/TO/shared/journal-A)

/subsystem=messaging-activemq/server=backup/path=large-messages-directory:write-attribute(name=path,value=/PATH/TO/shared/largemessages-A)

/subsystem=messaging-activemq/server=backup/path=paging-directory:write-attribute(name=path,value=/PATH/TO/shared/paging-A)
```

- *Instructions for a data replication policy:* Each server uses its own directories and does not share them with any other server. In the example commands below, each value for a **path** location is assumed to be a unique location on a file system. There is no need to change the directory locations for the live servers since they will use the default locations. However, the backup servers still must be configured with unique locations.

  a. Use the following management CLI commands to configure the directory locations for *node one*:

```
/subsystem=messaging-activemq/server=backup/path=bindings-directory:write-attribute(name=path,value=activemq/bindings-B)

/subsystem=messaging-activemq/server=backup/path=journal-directory:write-attribute(name=path,value=activemq/journal-B)

/subsystem=messaging-activemq/server=backup/path=large-messages-directory:write-attribute(name=path,value=activemq/largemessages-B)

/subsystem=messaging-activemq/server=backup/path=paging-directory:write-attribute(name=path,value=activemq/paging-B)
```

  b. Use the following management CLI commands to configure the directory locations for *node two*:

```
/subsystem=messaging-activemq/server=backup/path=bindings-directory:write-attribute(name=path,value=activemq/bindings-B)

/subsystem=messaging-activemq/server=backup/path=journal-directory:write-
```

```
attribute(name=path,value=activemq/journal-B)

/subsystem=messaging-activemq/server=backup/path=large-messages-
directory:write-attribute(name=path,value=activemq/largemessages-B)

/subsystem=messaging-activemq/server=backup/path=paging-directory:write-
attribute(name=path,value=activemq/paging-B)
```

4. Add a new acceptor and connector to the backup servers.
   Each backup server must be configured with an **http-connector** and an **http-acceptor** that uses the default **http-listener**. This allows a server to receive and send communications over the HTTP port. The following example adds an **http-acceptor** and an **http-connector** to the backup server.

   ```
   /subsystem=messaging-activemq/server=backup/http-acceptor=http-acceptor:add(http-
   listener=default)

   /subsystem=messaging-activemq/server=backup/http-connector=http-
   connector:add(endpoint=http-acceptor,socket-binding=http)
   ```

5. Configure the **cluster-connection** for the backup servers.
   Each messaging server needs a **cluster-connection**, a **broadcast-group**, and a **discovery-group** for proper communication. Use the following management CLI commands to configure these elements.

   ```
   /subsystem=messaging-activemq/server=backup/broadcast-group=bg-
   group1:add(connectors=[http-connector],jgroups-cluster=activemq-cluster)

   /subsystem=messaging-activemq/server=backup/discovery-group=dg-group1:add(jgroups-
   cluster=activemq-cluster)

   /subsystem=messaging-activemq/server=backup/cluster-connection=my-
   cluster:add(connector-name=http-connector,cluster-connection-address=jms,discovery-
   group=dg-group1)
   ```

The colocated server configuration is now completed.

30.7. FAILOVER MODES

JBoss EAP messaging defines two types of client failover:

- Automatic client failover
- Application-level client failover

JBoss EAP messaging also provides 100% transparent automatic reattachment of connections to the same server (e.g. in case of transient network problems). This is similar to failover, except it is reconnecting to the same server and is discussed in Client Reconnection and Session Reattachment.

During failover, if the client has consumers on any non persistent or temporary queues, those queues will be automatically recreated during failover on the backup node, since the backup node will not have any knowledge of non persistent queues.

30.7.1. Automatic Client Failover

JBoss EAP messaging clients can be configured to receive knowledge of all live and backup servers, so that in the event of a connection failure at the client - live server connection, the client will detect the failure and reconnect to the backup server. The backup server will then automatically recreate any sessions and consumers that existed on each connection before failover, thus saving the user from having to hand-code manual reconnection logic.

A JBoss EAP messaging client detects connection failure when it has not received packets from the server within the time given by **client-failure-check-period** as explained in Detecting Dead Connections.

If the client does not receive data in the allotted time, it will assume the connection has failed and attempt failover. If the socket is closed by the operating system, the server process might be killed rather than the server hardware itself crashing for example, the client will failover straight away.

JBoss EAP messaging clients can be configured to discover the list of live-backup server pairs in a number of different ways. They can be configured with explicit endpoints, for example, but the most common way is for the client to receive information about the cluster topology when it first connects to the cluster. See Server Discovery for more information.

The default HA configuration includes a **cluster-connection** that uses the recommended **http-connector** for cluster communication. This is the same **http-connector** that remote clients use when making connections to the server using the default **RemoteConnectionFactory**. While it is not recommended, you can use a different connector. If you use your own connector, make sure it is included as part of the configuration for both the **connection-factory** to be used by the remote client and the **cluster-connection** used by the cluster nodes. See Configuring the Messaging Transports and Cluster Connections for more information on connectors and cluster connections.

WARNING

The **connector** defined in the **connection-factory** to be used by a Jakarta Messaging client must be the same one defined in the **cluster-connection** used by the cluster. Otherwise, the client will not be able to update its topology of the underlying live/backup pairs and therefore will not know the location of the backup server.

Use CLI commands to review the configuration for both the **connection-factory** and the **cluster-connection**. For example, to read the current configuration for the **connection-factory** named **RemoteConnectionFactory** use the following command.

```
/subsystem=messaging-activemq/server=default/connection-
factory=RemoteConnectionFactory:read-resource
```

Likewise, the command below reads the configuration for the **cluster-connection** named **my-cluster**.

```
/subsystem=messaging-activemq/server=default/cluster-connection=my-cluster:read-resource
```

To enable automatic client failover, the client must be configured to allow non-zero reconnection attempts. See Client Reconnection and Session Reattachment for more information. By default, failover will occur only after at least one connection has been made to the live server. In other words, failover will not occur if the client fails to make an initial connection to the live server. If it does fail its initial attempt, a client would simply retry connecting to the live server according to the **reconnect-attempts** property and fail after the configured number of attempts.

```
/subsystem=messaging-activemq/server=default/connection-
factory=RemoteConnectionFactory:write-attribute(name=reconnect-attempts,value=<NEW_VALUE>)
```

An exception to this rule is the case where there is only one pair of live - backup servers, and no other live server, and a remote MDB is connected to the live server when it is cleanly shut down. If the MDB has configured **@ActivationConfigProperty(propertyName = "rebalanceConnections", propertyValue = "true")**, it tries to rebalance its connection to another live server and will not failover to the backup.

Failing Over on the Initial Connection
Since the client does not learn about the full topology until after the first connection is made, there is a window of time where it does not know about the backup. If a failure happens at this point the client can only try reconnecting to the original live server. To configure how many attempts the client will make you can set the property **initialConnectAttempts** on the **ClientSessionFactoryImpl** or **ActiveMQConnectionFactory**.

Alternatively in the server configuration, you can set the **initial-connect-attempts** attribute of the connection factory used by the client. The default for this is **0**, that is, try only once. Once the number of attempts has been made, an exception will be thrown.

```
/subsystem=messaging-activemq/server=default/connection-
factory=RemoteConnectionFactory:write-attribute(name=initial-connect-attempts,value=
<NEW_VALUE>)
```

About Server Replication
JBoss EAP messaging does not replicate full server state between live and backup servers. When the new session is automatically recreated on the backup, it won't have any knowledge of the messages already sent or acknowledged during that session. Any in-flight sends or acknowledgements at the time of failover may also be lost.

By replicating full server state, JBoss EAP messaging could theoretically provide a 100% transparent seamless failover, avoiding any lost messages or acknowledgements. However, doing so comes at a great cost: replicating the full server state, including the queues and session. This would require replication of the entire server state machine. That is, every operation on the live server would have to replicated on the replica servers in the exact same global order to ensure a consistent replica state. This is extremely hard to do in a performant and scalable way, especially considering that multiple threads are changing the live server state concurrently.

It is possible to provide full state machine replication using techniques such as virtual synchrony, but this does not scale well and effectively serializes all operations to a single thread, dramatically reducing concurrency. Other techniques for multi-threaded active replication exist such as replicating lock states or replicating thread scheduling, but this is very hard to achieve at a Java level.

Consequently, it was not worth reducing performance and concurrency for the sake of 100% transparent failover. Even without 100% transparent failover, it is simple to guarantee once and only once delivery, even in the case of failure, by using a combination of duplicate detection and retrying of transactions. However this is not 100% transparent to the client code.

30.7.1.1. Handling Blocking Calls During Failover

[illegible]

If the client code is in a blocking call to the server, i.e. it is waiting for a response to continue its execution, during a failover, the new session will not have any knowledge of the call that was in progress. The blocked call might otherwise hang forever, waiting for a response that will never come.

To prevent this, JBoss EAP messaging will unblock any blocking calls that were in progress at the time of failover by making them throw a **javax.jms.JMSException**, if using Jakarta Messaging, or an **ActiveMQException** with error code **ActiveMQException.UNBLOCKED** if using the core API. It is up to the client code to catch this exception and retry any operations if desired.

If the method being unblocked is a call to commit(), or prepare(), then the transaction will be automatically rolled back and JBoss EAP messaging will throw a **javax.jms.TransactionRolledBackException**, if using Jakarta Messaging, or a **ActiveMQException** with error code **ActiveMQException.TRANSACTION_ROLLED_BACK** if using the core API.

30.7.1.2. Handling Failover With Transactions

If the session is transactional and messages have already been sent or acknowledged in the current transaction, then the server cannot be sure whether messages or acknowledgements were lost during the failover.

Consequently the transaction will be marked as rollback-only, and any subsequent attempt to commit it will throw a **javax.jms.TransactionRolledBackException**,if using Jakarta Messaging. or a **ActiveMQException** with error code **ActiveMQException.TRANSACTION_ROLLED_BACK** if using the core API.

WARNING

The caveat to this rule is when XA is used either via Jakarta Messaging or through the core API. If a two phase commit is used and **prepare()** has already been called then rolling back could cause a **HeuristicMixedException**. Because of this the commit will throw a **XAException.XA_RETRY** exception. This informs the Transaction Manager that it should retry the commit at some later point in time, a side effect of this is that any non persistent messages will be lost. To avoid this from happening, be sure to use persistent messages when using XA. With acknowledgements this is not an issue since they are flushed to the server before **prepare()** gets called.

It is up to the user to catch the exception and perform any client side local rollback code as necessary. There is no need to manually rollback the session since it is already rolled back. The user can then just retry the transactional operations again on the same session.

If failover occurs when a commit call is being executed, the server, as previously described, will unblock the call to prevent a hang, since no response will come back. In this case it is not easy for the client to determine whether the transaction commit was actually processed on the live server before failure occurred.

NOTE

If XA is being used either via Jakarta Messaging or through the core API then an **XAException.XA_RETRY** is thrown. This is to inform Transaction Managers that a retry should occur at some point. At some later point in time the Transaction Manager will retry the commit. If the original commit has not occurred, it will still exist and be committed. If it does not exist, then it is assumed to have been committed, although the transaction manager may log a warning.

To remedy this, the client can enable duplicate detection in the transaction, and retry the transaction operations again after the call is unblocked. See Duplicate Message Detection for information on how detection is configured on the server. If the transaction had indeed been committed on the live server successfully before failover, duplicate detection will ensure that any durable messages resent in the transaction will be ignored on the server to prevent them getting sent more than once when the transaction is retried.

30.7.1.3. Getting Notified of Connection Failure

Jakarta Messaging provides a standard mechanism for sending asynchronously notifications of a connection failure: **java.jms.ExceptionListener**. Please consult the Jakarta Messaging javadoc for more information on this class. The core API also provides a similar feature in the form of the class **org.apache.activemq.artemis.core.client.SessionFailureListener**.

Any **ExceptionListener** or **SessionFailureListener** instance will always be called by JBoss EAP in case of a connection failure, whether the connection was successfully failed over, reconnected, or reattached. However, you can find out if the reconnect or reattach has happened by inspecting the value for the **failedOver** flag passed into **connectionFailed()** on **SessionfailureListener** or the error code on the **javax.jms.JMSException** which will be one of the following:

JMSException error codes

Error code	Description
FAILOVER	Failover has occurred and we have successfully reattached or reconnected.
DISCONNECT	No failover has occurred and we are disconnected.

30.7.2. Application-Level Failover

In some cases you may not want automatic client failover, and prefer to handle any connection failure yourself, and code your own manually reconnection logic in your own failure handler. We define this as application-level failover, since the failover is handled at the user application level.

To implement application-level failover if you're using Jakarta Messaging set an **ExceptionListener** class on the Jakarta Messaging connection. The **ExceptionListener** will be called by JBoss EAP messaging in the event that connection failure is detected. In your **ExceptionListener**, you would close your old Jakarta Messaging connections, potentially look up new connection factory instances from JNDI and creating new connections.

If you are using the core API, then the procedure is very similar: you would set a **FailureListener** on the core **ClientSession** instances.

30.8. DETECTING DEAD CONNECTIONS

This section discusses connection time to live (TTL) and explains how JBoss EAP messaging handles crashed clients and clients that have exited without cleanly closing their resources.

Cleaning up Dead Connection Resources on the Server

Before a JBoss EAP client application exits, it should close its resources in a controlled manner, using a **finally** block.

Below is an example of a core client appropriately closing its session and session factory in a **finally** block:

```
ServerLocator locator = null;
ClientSessionFactory sf = null;
ClientSession session = null;

try {
  locator = ActiveMQClient.createServerLocatorWithoutHA(..);

  sf = locator.createClientSessionFactory();;

  session = sf.createSession(...);

  ... do some stuff with the session...
}
finally {
  if (session != null) {
    session.close();
  }

  if (sf != null) {
    sf.close();
  }

  if(locator != null) {
    locator.close();
  }
}
```

And here is an example of a well behaved Jakarta Messaging client application:

```
Connection jmsConnection = null;

try {
  ConnectionFactory jmsConnectionFactory =
ActiveMQJMSClient.createConnectionFactoryWithoutHA(...);

  jmsConnection = jmsConnectionFactory.createConnection();

  ... do some stuff with the connection...
}
finally {
  if (connection != null) {
        connection.close();
      }
   }
```

Unfortunately sometimes clients crash and do not have a chance to clean up their resources. If this occurs, it can leave server side resources hanging on the server. If these resources are not removed they would cause a resource leak on the server, and over time this likely would result in the server running out of memory or other resources.

When looking to clean up dead client resources, it is important to be aware of the fact that sometimes the network between the client and the server can fail and then come back, allowing the client to reconnect. Because JBoss EAP supports client reconnection, it is important that it not clean up "dead" server side resources too soon, or clients will be prevented any client from reconnecting and regaining their old sessions on the server.

JBoss EAP makes all of this configurable. For each **ClientSessionFactory** configured, a Time-To-Live, or TTL, property can be used to set how long the server will keep a connection alive in milliseconds in the absence of any data from the client. The client will automatically send "ping" packets periodically to prevent the server from closing its connection. If the server does not receive any packets on a connection for the length of the TTL time, it will automatically close all the sessions on the server that relate to that connection.

If you are using Jakarta Messaging, the connection TTL is defined by the **ConnectionTTL** attribute on a **ActiveMQConnectionFactory** instance, or if you are deploying Jakarta Messaging connection factory instances direct into JNDI on the server side, you can specify it in the xml config, using the parameter **connectionTtl**.

The default value for **ConnectionTTL** on an network-based connection, such as an **http-connector**, is **60000**, i.e. 1 minute. The default value for connection TTL on a internal connection, e.g. an **in-vm** connection, is **-1**. A value of **-1** for **ConnectionTTL** means the server will never time out the connection on the server side.

If you do not want clients to specify their own connection TTL, you can set a global value on the server side. This can be done by specifying the **connection-ttl-override** attribute in the server configuration. The default value for **connection-ttl-override** is **-1** which means "do not override", i.e. let clients use their own values.

Closing Core Sessions or Jakarta Messaging Connections

It is important that all core client sessions and Jakarta Messaging connections are always closed explicitly in a **finally** block when you are finished using them.

If you fail to do so, JBoss EAP will detect this at garbage collection time. It will then close the connection and log a warning similar to the following:

```
[Finalizer] 20:14:43,244 WARNING [org.apache.activemq.artemis.core.client.impl.DelegatingSession]
I'm closing a ClientSession you left open. Please make sure you close all ClientSessions explicitly
before let
ting them go out of scope!
[Finalizer] 20:14:43,244 WARNING [org.apache.activemq.artemis.core.client.impl.DelegatingSession]
The session you didn't close was created here:
java.lang.Exception
   at org.apache.activemq.artemis.core.client.impl.DelegatingSession.<init>
(DelegatingSession.java:83)
   at org.acme.yourproject.YourClass (YourClass.java:666)
```

Note that if you are using Jakarta Messaging the warning will involve a Jakarta Messaging connection, not a client session. Also, the log will tell you the exact line of code where the unclosed Jakarta Messaging connection or core client session was instantiated. This will enable you to pinpoint the error in your code and correct it appropriately.

Detecting Failure from the Client Side

As long as the client is receiving data from the server it will consider the connection to be alive. If the client does not receive any packets for **client-failure-check-period** milliseconds, it will consider the connection failed and will either initiate failover, or call any **FailureListener** instances, or **ExceptionListener** instances if you are using Jakarta Messaging, depending on how the client has been configured.

If you are using Jakarta Messaging the behavior is defined by the **ClientFailureCheckPeriod** attribute on a **ActiveMQConnectionFactory** instance.

The default value for client failure check period on a network connection, for example an HTTP connection, is **30000**, or 30 seconds. The default value for client failure check period on an in-vm connection, is **-1**. A value of **-1** means the client will never fail the connection on the client side if no data is received from the server. Whatever the type of connection, the check period is typically much lower than the value for connection TTL on the server so that clients can reconnect in case of transitory failure.

Configuring Asynchronous Connection Execution

Most packets received on the server side are executed on the **remoting** thread. These packets represent short-running operations and are always executed on the **remoting** thread for performance reasons.

However, by default some kinds of packets are executed using a thread from a thread pool so that the **remoting** thread is not tied up for too long. Please note that processing operations asynchronously on another thread adds a little more latency. These packets are:

```
org.apache.activemq.artemis.core.protocol.core.impl.wireformat.RollbackMessage

org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionCloseMessage

org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionCommitMessage

org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionXACommitMessage

org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionXAPrepareMessage

org.apache.activemq.artemis.core.protocol.core.impl.wireformat.SessionXARollbackMessage
```

To disable asynchronous connection execution, set the parameter **async-connection-execution-enabled** to **false**. The default value is **true**.

30.9. CLIENT RECONNECTION AND SESSION REATTACHMENT

JBoss EAP messaging clients can be configured to automatically reconnect or reattach to the server in the event that a failure is detected in the connection between the client and the server.

Transparent Session Reattachment

If the failure was due to some transient cause such as a temporary network outage, and the target server was not restarted, the sessions will still exist on the server, assuming the client has not been disconnected for more than the value of **connection-ttl**. See Detecting Dead Connections.

In this scenario, JBoss EAP will automatically reattach the client sessions to the server sessions when the re-connection is made. This is done 100% transparently and the client can continue exactly as if nothing had happened.

As JBoss EAP messaging clients send commands to their servers they store each sent command in an in-memory buffer. When a connection fails and the client subsequently attempts to reattach to the same server, as part of the reattachment protocol, the server gives the client the id of the last command it successfully received.

If the client has sent more commands than were received before failover it can replay any sent commands from its buffer so that the client and server can reconcile their states.

The size in bytes of this buffer is set by the **confirmationWindowSize** property. When the server has received **confirmationWindowSize** bytes of commands and processed them it will send back a command confirmation to the client, and the client can then free up space in the buffer.

If you are using the Jakarta Messaging service on the server to load your Jakarta Messaging connection factory instances into JNDI, then this property can be configured in the server configuration, by setting the **confirmation-window-size** attribute of the chosen **connection-factory**. If you are using Jakarta Messaging but not using JNDI then you can set these values directly on the **ActiveMQConnectionFactory** instance using the appropriate setter method, **setConfirmationWindowSize**. If you are using the core API, the **ServerLocator** instance has a **setConfirmationWindowSize** method exposed as well.

Setting **confirmationWindowSize** to **-1**, which is also the default, disables any buffering and prevents any reattachment from occurring, forcing a reconnect instead.

Session Reconnection

Alternatively, the server might have actually been restarted after crashing or it might have been stopped. In such a case any sessions will no longer exist on the server and it will not be possible to 100% transparently reattach to them.

In this case, JBoss EAP will automatically reconnect the connection and recreate any sessions and consumers on the server corresponding to the sessions and consumers on the client. This process is exactly the same as what happens when failing over to a backup server.

Client reconnection is also used internally by components such as core bridges to allow them to reconnect to their target servers.

See the section on Automatic Client Failover to get a full understanding of how transacted and non-transacted sessions are reconnected during a reconnect and what you need to do to maintain once and only once delivery guarantees.

Configuring Reconnection Attributes

Client reconnection is configured by setting the following properties:

- retryInterval. This optional parameter sets the period in milliseconds between subsequent reconnection attempts, if the connection to the target server has failed. The default value is **2000** milliseconds.

- retryIntervalMultiplier. This optional parameter sets a multiplier to apply to the time since the last retry to compute the time to the next retry. This allows you to implement an exponential backoff between retry attempts.
  For example, if you set **retryInterval** to **1000** ms and set retryIntervalMultiplier to **2.0**, then, if the first reconnect attempt fails, the client will wait **1000** ms then **2000** ms then **4000** ms between subsequent reconnection attempts.

The default value is **1.0** meaning each reconnect attempt is spaced at equal intervals.

- maxRetryInterval. This optional parameter sets the maximum retry interval that will be used. When setting **retryIntervalMultiplier** it would otherwise be possible that subsequent retries exponentially increase to ridiculously large values. By setting this parameter you can set an upper limit on that value. The default value is **2000** milliseconds.

- reconnectAttempts. This optional parameter sets the total number of reconnect attempts to make before giving up and shutting down. A value of **-1** signifies an unlimited number of attempts. The default value is **0**.

If you are using Jakarta Messaging and JNDI on the client to look up your Jakarta Messaging connection factory instances then you can specify these parameters in the JNDI context environment. For example, your **jndi.properties** file might look like the following.

```
java.naming.factory.initial = ActiveMQInitialContextFactory
connection.ConnectionFactory=tcp://localhost:8080?
retryInterval=1000&retryIntervalMultiplier=1.5&maxRetryInterval=60000&reconnectAttempts=1000
```

If you are using Jakarta Messaging, but instantiating your Jakarta Messaging connection factory directly, you can specify the parameters using the appropriate setter methods on the **ActiveMQConnectionFactory** immediately after creating it.

If you are using the core API and instantiating the **ServerLocator** instance directly you can also specify the parameters using the appropriate setter methods on the ServerLocator immediately after creating it.

If your client does manage to reconnect but the session is no longer available on the server, for instance if the server has been restarted or it has timed out, then the client will not be able to reattach, and any **ExceptionListener** or **FailureListener** instances registered on the connection or session will be called.

ExceptionListeners and SessionFailureListeners

Note that when a client reconnects or reattaches, any registered Jakarta Messaging **ExceptionListener** or core API **SessionFailureListener** will be called.

CHAPTER 31. RESOURCE ADAPTERS

A Jakarta Connectors Resource Adapter lets your applications communicate with any messaging provider. It configures how Jakarta EE components such as MDBs and other Jakarta Enterprise Beans, and even Servlets, can send or receive messages.

31.1. ABOUT THE INTEGRATED ARTEMIS RESOURCE ADAPTER

JBoss EAP 7 includes an integrated Artemis resource adapter, which uses the **pooled-connection-factory** element to configure the outbound and inbound connections of the resource adapter.

Outbound Connection

Outbound connections are defined using the **pooled-connection-factory** element, which is then used in Jakarta EE deployments by Jakarta Enterprise Beans and servlets to send messages to and receive messages from queues or topics. Because connections created from connection factories are created in the scope of the application server, they can use application server features like the following:

- Connection pooling
- Authentication using the security domains defined by the application server
- Participation in XA transactions using the transaction manager

This is a major difference with a **pooled-connection-factory** as these features are not available with a basic **connection-factory** like **InVmConnectionFactory** or **RemoteConnectionFactory**. Also, be aware that with a connection factory defined using **pooled-connection-factory**, it is not possible to do a lookup using JNDI from an external standalone Jakarta Messaging client.

Inbound Connections

Inbound connections are used only by message-driven beans (MDBs) to receive message from a queue or a topic. MDBs are stateless session beans that listen on a queue or topic. They must implement the public **onMessage(Message message)** method, which is called when a message is sent to a queue or a topic. The Artemis resource adapter is responsible for receiving the message from the queue or the topic and passing it to the **onMessage(Message message)** method. For this purpose it configures the inbound connection, which defines the location of the integrated Artemis server and some additional elements.

Each MDB session bean uses a thread from the client thread pool to consume the message from the destination. If the maximum pool size is not defined, it is determined to be eight (8) times the number of CPU core processors. For systems with many MDB sessions, such as test suites, this can potentially lead to thread exhaustion and force MDBs to wait for a free thread from the pool. You can increase the maximum pool size of client thread pool using the management CLI. The following command sets the maximum client thread pool size to **128**.

```
/subsystem=messaging-activemq:write-attribute(name=global-client-thread-pool-max-size,value=128)
```

For information about how to configure the client thread pool size, see Client Thread Management. For more information about MDBs, see Message Driven Beans in *Developing Jakarta Enterprise Beans Applications* for JBoss EAP.

31.2. USING THE INTEGRATED ARTEMIS RESOURCE ADAPTER FOR REMOTE CONNECTIONS

JBoss EAP includes a resource adapter to make connections to its integrated ActiveMQ Artemis messaging server. By default the **pooled-connection-factory** defined in the **messaging-activemq** subsystem uses the adapter to make the connections. However, you can use the same resource adapter to make connections to an Artemis server running inside a remote instance of JBoss EAP as well.

IMPORTANT

The **activemq-ra** pooled connection factory, which is configured by default in the **messaging-activemq** subsystem, has the **java:jboss/DefaultJMSConnectionFactory** entry assigned. This entry is required by the **messaging-activemq** subsystem. If you decide to remove the **activemq-ra** pooled connection factory, you must assign this entry to a different connection factory. Otherwise you will see the following error in the server log on deployment.

```
WFLYCTL0412: Required services that are not installed:" =>
["jboss.naming.context.java.jboss.DefaultJMSConnectionFactory"]
```

To connect to an Artemis server running inside a remote instance of JBoss EAP, create a new **pooled-connection-factory** by following the steps below.

1. Create an outbound-socket-binding pointing to the remote messaging server:

```
/socket-binding-group=standard-sockets/remote-destination-outbound-socket-
binding=remote-server:add(host=<server host>, port=8080)
```

2. Create a remote-connector referencing the outbound-socket-binding created in step 1.

```
/subsystem=messaging-activemq/server=default/http-connector=remote-http-
connector:add(socket-binding=remote-server,endpoint=http-acceptor)
```

3. Create a pooled-connection-factory referencing the remote-connector created in step 2.

```
/subsystem=messaging-activemq/server=default/pooled-connection-factory=remote-
artemis:add(connectors=[remote-http-connector], entries=[java:/jms/remoteCF])
```

NOTE

Artemis 1.x required a prefix on destination names (jms.topic for topics and jms.queue for queues). Artemis 2.x does not require prefixes, but for compatibility with Artemis 1.x, EAP still adds the prefix and directs Artemis to run in compatibility mode. If you connect to a remote Artemis 2.x server, it may not be in compatibility mode and you may not need the prefixes. When using destinations without a prefix, you can configure the connection factory not to include the prefixes by setting the attribute **enable-amq1-prefix** to false.

Configuring an MDB to use a pooled-connection-factory

After the **pooled-connection-factory** is configured to connect to a remote Artemis server, Message-Driven Beans (MDB) wanting to read messages from the remote server must be annotated with the **@ResourceAdapter** annotation using the name of the **pooled-connection-factory** resource.

```
import org.jboss.ejb3.annotation.ResourceAdapter;

@ResourceAdapter("remote-artemis")
```

```
@MessageDriven(name = "MyMDB", activationConfig = { ... })
public class MyMDB implements MessageListener {
   public void onMessage(Message message) {
      ...
   }
}
```

If the MDB needs to send messages to the remote server, it must inject the **pooled-connection-factory** by looking it up using one of its JNDI **entries**.

```
@Inject
@JMSConnectionFactory("java:/jms/remoteCF")
private JMSContext context;
```

Configuring the Jakarta Messaging destination

An MDB must also specify the destination from which it will consume messages. The standard way to do this is to define a **destinationLookup** activation config property that corresponds to a JNDI lookup on the local server.

```
@ResourceAdapter("remote-artemis")
@MessageDriven(name = "MyMDB", activationConfig = {
   @ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "myQueue"),
   ...
})
public class MyMDB implements MessageListener {
   ...
}
```

If the local server does not include a JNDI binding for the remote Artemis server, specify the name of the destination, as configured in the remote Artemis server, using the **destination** activation config property and set the **useJNDI** activation config property to **false**. This instructs the Artemis resource adapter to automatically create the Jakarta Messaging destination without requiring a JNDI lookup.

```
@ResourceAdapter("remote-artemis")
@MessageDriven(name = "MyMDB", activationConfig = {
   @ActivationConfigProperty(propertyName = "useJNDI", propertyValue = "false"),
   @ActivationConfigProperty(propertyName = "destination", propertyValue = "myQueue"),
   ...
})
public class MyMDB implements MessageListener {
   ...
}
```

In the above example, the activation config properties configure the MDB to consume messages from the Jakarta Messaging Queue named **myQueue** hosted on the remote Artemis server. In most cases, the MDB does not need to lookup other destinations to process the consumed messages, and it can use the **JMSReplyTo** destination if it is defined on the message.

If the MDB needs any other Jakarta Messaging destinations defined on the remote server, it must use client-side JNDI. See Connecting to a Server for more information.

31.3. CONFIGURING THE ARTEMIS RESOURCE ADAPTER TO CONNECT TO RED HAT AMQ

You can configure the integrated Artemis resource adapter to connect to a remote installation of Red Hat AMQ 7, which then becomes the Jakarta Messaging provider for your JBoss EAP 7.4 applications. This allows JBoss EAP to be a client for the remote Red Hat AMQ 7 server.

If you require support for other messaging protocols, such as AMQP or STOMP, you must configure Red Hat AMQ 7 as your messaging broker. The Artemis resource adapter integrated in the JBoss EAP server can then be used to process messages for the deployed applications.

Limitations of the Integrated Resource Adapter

Dynamic Creation of Queues and Topics

Be aware that the Artemis resource adapter that is integrated in JBoss EAP 7.4 does not support dynamic creation of queues and topics in the Red Hat AMQ 7 broker. You must configure all queue and topic destinations directly on the remote Red Hat AMQ 7 server.

Creation of Connection Factories

Although Red Hat AMQ allows connection factories to be configured using both the **pooled-connection-factory** and the **external-context**, there is a difference in the way each connection factory is created. When the **external-context** is used to create the connection factory, it creates simple Jakarta Messaging connection factory as defined in the Jakarta Messaging specification. The newly created connection factory is equivalent to the **RemoteConnectionFactory**, which is defined by default in **messaging-activemq** subsystem. This connection factory is independent of the other components in the application server, meaning it is not aware of, nor is it able to use, other components like the transaction manager or the security manager. For this reason, only the **pooled-connection-factory** can be used to create connection factories in JBoss EAP 7. The **external-context** can only be used to register Jakarta Messaging destinations, which are already configured on the remote AMQ 7 broker, into the JNDI tree of the JBoss EAP 7 server so that local deployments can look them up or inject them.

Connection factories created by configuring the **external-context** or the **connection-factory** elements cannot be used to connect to the remote AMQ 7 broker as they do not use the Artemis resource adapter. Only connection factories created by configuring the **pooled-connection-factory** element are supported for use when connecting to the remote AMQ7 broker.

Configure JBoss EAP to Use a Remote Red Hat AMQ Server

You can use the management CLI to configure JBoss EAP to use a remote installation of Red Hat AMQ 7 as the messaging provider by following the steps below:

1. Configure the queue in the Red Hat AMQ 7 **broker.xml** deployment descriptor file.

```
<configuration xmlns="urn:activemq"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="urn:activemq /schema/artemis-configuration.xsd">

   <core xmlns="urn:activemq:core" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
        xsi:schemaLocation="urn:activemq:core ">
        ...
        <acceptors>
           <acceptor name="netty-acceptor">tcp://localhost:61616?
anycastPrefix=jms.queue.;multicastPrefix=jms.topic.
           </acceptor>
        </acceptors>
        <addresses>
           <address name="MyQueue">
              <anycast>
                 <queue name="MyQueue" />
              </anycast>
            </address>
            <address name="MyOtherQueue">
                <anycast>
                    <queue name="MyOtherQueue" />
                </anycast>
            </address>
            <address name="MyTopic">
                <multicast/>
            </address>
        <addresses>
        ...
    </core>
</configuration>
```

NOTE

The Artemis resource adapter that is included with JBoss EAP uses the ActiveMQ Artemis Jakarta Messaging Client 2.x. This client requires **anycastPrefix** and **multicastPrefix** prefixing on the address. It also expects the queue name to be the same as the address name.

2. Create the remote connector.

```
/subsystem=messaging-activemq/remote-connector=netty-remote-throughput:add(socket-binding=messaging-remote-throughput)
```

This creates the following **remote-connector** in the **messaging-activemq** subsystem.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
    ...
    <remote-connector name="netty-remote-throughput" socket-binding="messaging-remote-throughput"/>
    ...
</subsystem>
```

3. Add the remote destination outbound socket binding.

```
/socket-binding-group=standard-sockets/remote-destination-outbound-socket-binding=messaging-remote-throughput:add(host=localhost, port=61616)
```

This creates the following **remote-destination** in the **outbound-socket-binding** element configuration.

```
<outbound-socket-binding name="messaging-remote-throughput">
    <remote-destination host="localhost" port="61616"/>
</outbound-socket-binding>
```

4. Add a pooled connection factory for the remote connector.

```
/subsystem=messaging-activemq/pooled-connection-factory=activemq-ra-remote:add(transaction=xa,entries=[java:/RemoteJmsXA, java:jboss/RemoteJmsXA],connectors=[netty-remote-throughput])
```

This creates the following **pooled-connection-factory** in the **messaging-activemq** subsystem.

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
  ...
  <pooled-connection-factory name="activemq-ra-remote" entries="java:/RemoteJmsXA
java:jboss/RemoteJmsXA" connectors="netty-remote-throughput"/>
  ...
</subsystem>
```

NOTE

Artemis 1.x required a prefix on destination names (jms.topic for topics and jms.queue for queues). Artemis 2.x does not require prefixes, but for compatibility with Artemis 1.x, EAP still adds the prefix and directs Artemis to run in compatibility mode. If you connect to a remote Artemis 2.x server, it may not be in compatibility mode and you may not need the prefixes. When using destinations without a prefix, you can configure the connection factory not to include the prefixes by setting the attribute **enable-amq1-prefix** to `false`.

5. Create the **external-context** bindings for the queues and topics.

```
/subsystem=naming/binding=java\:global\/remoteContext:add(binding-type=external-context,
class=javax.naming.InitialContext, module=org.apache.activemq.artemis, environment=
[java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory,
java.naming.provider.url=tcp://127.0.0.1:61616, queue.MyQueue=MyQueue,
queue.MyOtherQueue=MyOtherQueue, topic.MyTopic=MyTopic])
```

This creates the following **external-context** bindings in the **naming** subsystem.

```
<subsystem xmlns="urn:jboss:domain:naming:2.0">
  ...
  <bindings>
    <external-context name="java:global/remoteContext"
module="org.apache.activemq.artemis" class="javax.naming.InitialContext">
      <environment>
        <property name="java.naming.factory.initial"
value="org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory"/>
        <property name="java.naming.provider.url" value="tcp://127.0.0.1:61616"/>
        <property name="queue.MyQueue" value="MyQueue"/>
        <property name="queue.MyOtherQueue" value="MyOtherQueue"/>
        <property name="topic.MyTopic" value="MyTopic"/>
      </environment>
    </external-context>
  </bindings>
  ...
</subsystem>
```

6. Create the lookup entry for the Jakarta Messaging queues and topics by setting the JNDI name to the Red Hat AMQ 7 address name value. This creates a mapping between the JNDI name and the Red Hat AMQ 7 address name.

```
/subsystem=naming/binding=java\:\/MyQueue:add(lookup=java:global/remoteContext/MyQueue
,binding-type=lookup)
/subsystem=naming/binding=java\:\/MyOtherQueue:add(lookup=java:global/remoteContext/My
OtherQueue,binding-type=lookup)
/subsystem=naming/binding=java\:\/MyTopic:add(lookup=java:global/remoteContext/MyTopic,bi
nding-type=lookup)
```

This creates the following **lookup** configurations in the **naming** subsystem.

```
<subsystem xmlns="urn:jboss:domain:naming:2.0">
   ...
   <lookup name="java:/MyQueue" lookup="java:global/remoteContext/MyQueue"/>
   <lookup name="java:/MyOtherQueue"
lookup="java:global/remoteContext/MyOtherQueue"/>
   <lookup name="java:/MyTopic" lookup="java:global/remoteContext/MyTopic"/>
   ...
</subsystem>
```

Alternatively, define the **/subsystem=messaging-activemq/external-jms-queue** or the **/subsystem=messaging-activemq/external-jms-topic** resource instead of configuring naming subsystem. For example:

```
/subsystem=messaging-activemq/external-jms-queue=MyQueue:add(entries=
[java:/MyQueue])
```

This creates the following resource:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:4.0">
   ...
   <external-jms-queue name="MyQueue" entries="java:/MyQueue"/>
   ...
</subsystem>
```

NOTE

The **external-jms-queue** resource does not provide operations for queue management and statistics.

JBoss EAP is now configured to use the remote installation of Red Hat AMQ 7 as the messaging provider.

31.4. JAKARTA MESSAGING RESOURCES CONFIGURATION FOR A REMOTE ARTEMIS-BASED BROKER

From the management CLI, you can configure Jakarta Messaging resources for a remote Artemis-based broker, such as Red Hat AMQ 7, using the **@JMSConnectionFactoryDefinition** annotation or the **@JMSDestinationDefinition** annotation. You can also configure the resources from the management console.

The remote ActiveMQ server resources do not require a local instance of Artemis. This helps reduce the memory and CPU footprint of the JBoss EAP image.

31.4.1. Jakarta Messaging Resources Configuration Using the JMSConnectionFactoryDefinition Annotation

JBoss EAP uses the **@JMSConnectionFactoryDefinition** annotation to define a connection factory. This connection factory can connect to a local or a remote Artemis broker. The **resourceAdapter** element of the **@JMSConnectionFactoryDefinition** annotation then refers to the name of the **pooled-connection-factory** defined in the **messaging-subsystem** which can connect to a remote Artemis broker. The **resourceAdapter** element defines which resource adapter is used for creating a connection factory or in which resource adapter a connection factory is defined.

When the **resourceAdapter** element is not defined in the **@JMSConnectionFactoryDefinition** annotation, the **messaging-activemq** subsystem uses the JNDI name of the connection factory by default. This is known as default binding. The default binding is defined using the **jms-connection-factory** attribute at **/subsystem=ee/service=default-bindings**. If the **resourceAdapter** element is specified or one can be defined from the default binding for the **jms-connection-factory** and if it is a **pooled-connection-factory** to a remote broker, you can use it to connect to the remote broker.

If the **resourceAdapter** is not defined in the **messaging-activemq** subsystem or one cannot be obtained from the default binding for the **jms-connection-factory**, the task of creating the Jakarta Messaging resources is delegated to the **resource-adapters** subsystem based on the resource adapter's **admin-objects** and **connection-definitions** resources.

The following sections provide examples of how to configure and use the **@JMSConnectionFactoryDefinition** annotation.

Configuring @JMSConnectionFactoryDefinition Using the Default Resource Adapter

1. Create a socket binding to remote instance:

```
/socket-binding-group=standard-sockets/remote-destination-outbound-socket-
binding=messaging-remote-throughput:add(host=127.0.0.2, port=5445)
```

2. Create a connector:

```
/subsystem=messaging-activemq/remote-connector=remote-amq:add(socket-
binding="messaging-remote-throughput")
```

3. Create a pooled connection factory:

```
/subsystem=messaging-activemq/pooled-connection-factory=activemq-ra-
remote:add(entries=["java:/jms/remote-amq/JmsConnectionFactory"],connectors=["remote-
amq"]
```

4. Define the default Jakarta Messaging connection factory for the **ee** subsystem:

```
/subsystem=ee/service=default-bindings:write-attribute(name=jms-connection-factory,
value="java:/jms/remote-amq/JmsConnectionFactory")
```

5. Use the **@JMSConnectionFactoryDefinition** annotation in your application code:

```
@JMSConnectionFactoryDefinition(name="java:/jms/remote-amq/JmsConnectionFactory")
```

Configuring @JMSConnectionFactoryDefinition Using a Remote Artemis Broker

1. Create a connector:

```
/subsystem=messaging-activemq/remote-connector=remote-amq:add(socket-
binding="messaging-remote-throughput")
```

2. Create a pooled connection factory:

```
/subsystem=messaging-activemq/pooled-connection-factory=activemq-ra-
remote:add(entries=["java:/jms/remote-amq/JmsConnectionFactory"],connectors=["remote-
amq"])
```

3. Define the default Jakarta Messaging connection factory for the **ee** subsystem:

```
/subsystem=ee/service=default-bindings:write-attribute(name=jms-connection-factory,
value="java:/jms/remote-amq/JmsConnectionFactory")
```

4. Use the **@JMSConnectionFactoryDefinition** annotation in your application code:

```
@JMSConnectionFactoryDefinition(
        name="java:app/myCF"
        resourceAdapter="myPCF"
)
```

Configuring @JMSConnectionFactoryDefinition Using a Third-party JMS Resource Adapter

1. Create a connector:

```
/subsystem=messaging-activemq/remote-connector=remote-amq:add(socket-
binding="messaging-remote-throughput")
```

2. Create a pooled connection factory:

```
/subsystem=messaging-activemq/pooled-connection-factory=activemq-ra-
remote:add(entries=["java:/jms/remote-amq/JmsConnectionFactory"],connectors=["remote-
amq"])
```

3. Define the default Jakarta Messaging connection factory for the **ee** subsystem:

```
/subsystem=ee/service=default-bindings:write-attribute(name=jms-connection-factory,
value="java:/jms/remote-amq/JmsConnectionFactory")
```

4. Use the **@JMSConnectionFactoryDefinition** annotation in your application code:

```
@JMSConnectionFactoryDefinition(
        name="java:app/myCF"
        resourceAdapter="wsmq"
)
```

31.4.2. Jakarta Messaging Resources Configuration Using the JMSDestinationDefinition Annotation

You can use the server resources to create the required destinations for a **pooled-connection-factory** to a local broker.

If the **resourceAdapter** element points to a **pooled-connection-factory** name and it is defined in a local broker, for example, **/subsystem/messaging-activemq/server=default**, then it creates destinations in the local Artemis broker.

NOTE

If you need to create destinations in a remote Artemis-based broker, then the **pooled-connection-factory** must be defined in the **messaging-activemq** subsystem.

If the **resourceAdapter** set in the **@JMSDestinationDefinition** annotation matches the **resourceAdapter** element defined for the **server** in the **messaging-activemq** subsystem, then the destination is created in this broker, irrespective of whether the connector in the **pooled-connection-factory** points to a local or a remote Artemis broker.

Configuring Jakarta Messaging Resources Using the JMSDestinationDefinition Annotation

1. Create a connector:

```
/subsystem=messaging-activemq/remote-connector=remote-amq:add(socket-
binding="messaging-remote-throughput")
```

2. Create a pooled connection factory:

```
/subsystem=messaging-activemq/pooled-connection-factory=activemq-ra-
remote:add(entries=["java:/jms/remote-amq/JmsConnectionFactory"],connectors=["remote-
amq"])
```

3. Define the default Jakarta Messaging connection factory for the **ee** subsystem:

```
/subsystem=ee/service=default-bindings:write-attribute(name=jms-connection-factory,
value="java:/jms/remote-amq/JmsConnectionFactory")
```

4. Use the **@JMSDestinationDefinition** annotation in your application code:

```
@JMSDestinationDefinition(
   name = "java:/jms/queue/MessageBeanQueue",
   interfaceName = "javax.jms.Queue",
   destinationName = "MessageBeanQueue"
   properties= {
      "management-address=remote-activemq.management"
   }
)
```

31.4.3. Configuring Remote ActiveMQ Server Resources Using the Management Console

You can configure the following remote ActiveMQ server resources from the management console:

- Generic Connector
- In VM Connector
- HTTP Connector
- Remote Connector
- Discovery Group

- Connection Factory
- Pooled Connection Factory
- External JMS Queue
- External JMS Topic

To configure the remote ActiveMQ server resources from the management console:

1. Access the management console and navigate to **Configuration → Subsystems → Messaging (ActiveMQ) → Remote ActiveMQ Server** and click **View**.
2. In the navigation pane, click the resource you want to configure.

31.5. DEPLOYING A RED HAT JBOSS A-MQ RESOURCE ADAPTER

You can deploy the resource adapter provided by the Red Hat JBoss A-MQ product and have, for example, Red Hat JBoss A-MQ 6.3.0, become the external Jakarta Messaging provider for JBoss EAP.

See Install the ActiveMQ Resource Adapter in *Integrating with JBoss Enterprise Application Platform* , which is in the Red Hat JBoss A-MQ documentation suite, for details on how to deploy and configure a Red Hat JBoss A-MQ resource adapter.

NOTE

Be aware that the product name changed from Red Hat JBoss A-MQ in the 6.x releases to Red Hat AMQ in the 7.x releases.

31.5.1. Issues with the Red Hat JBoss A-MQ 6 Resource Adapter

- JBoss EAP will track and monitor applications, looking for unclosed resources. While useful in many cases, such monitoring might cause unexpected behavior when an application tries to re-use a closed instance of **UserTransaction** in a single method. Add the attribute **tracking="false"** to the **<connection-definition/>** element when configuring the Red Hat JBoss A-MQ resource adapter if your applications re-use connections in this way.

```
<connection-definition class-name="..." tracking="false" ... />
```

- The Red Hat JBoss A-MQ 6 resource adapter does not implement **XAResourceWrapper** from the Narayana API, which is used by JBoss EAP. Consequently, when the Transaction Manager sends a commit to all the XA transaction participants and then crashes while waiting for a reply, it will go on indefinitely logging warnings until records of the committed transaction are removed from its object store.

- The Red Hat JBoss A-MQ 6 resource adapter returns the code **XAER_RMERR** when an error, such as a network disconnection, occurs during the call of the commit method protocol. This behavior breaks the XA specification since the correct return code should be **XAER_RMFAIL** or **XAER_RETRY**. Consequently, the transaction is left in an unknown state on the message broker side, which can cause data inconsistency in some cases. A message will be logged similar to the one below when the unexpected error code is returned.

```
WARN [com.arjuna.ats.jtax] ...: XAResourceRecord.rollback caused an XA error:
ARJUNA016099: Unknown error code:0 from resource ... in transaction ...:
javax.transaction.xa.XAException: Transaction ... has not been started.
```

- Red Hat JBoss A-MQ 6.x supports the JMS 1.1 specification that is included with Java EE 6. It does not support the Jakarta Messaging 2.0 specification that was introduced in Jakarta EE 8 and is supported in JBoss EAP 7. If you need to send messages to a remote Red Hat JBoss A-MQ broker, you must use the JMS 1.1 API within your application code. For more information about Red Hat JBoss A-MQ 6.x supported standards, see Red Hat JBoss A-MQ Supported Standards and Protocols.

31.6. DEPLOYING THE IBM MQ RESOURCE ADAPTER

About IBM MQ

IBM MQ is the Messaging Oriented Middleware (MOM) product offering from IBM that allows applications on distributed systems to communicate with each other. This is accomplished through the use of messages and message queues. IBM MQ is responsible for delivering messages to the message queues and for transferring data to other queue managers using message channels. For more information about IBM MQ, see IBM MQ on the IBM products website.

Summary

IBM MQ can be configured as an external Jakarta Messaging provider for JBoss EAP 7.4. This section covers the steps to deploy and configure the IBM MQ resource adapter in JBoss EAP. This deployment and configuration can be accomplished by using the management CLI tool or the web-based management console. See JBoss EAP supported configurations for the most current information about the supported configurations of IBM MQ.

NOTE

You must restart your system after configuring your IBM MQ resource adapter for the configuration changes to take effect.

Prerequisites

Before you get started, you must verify the version of the IBM MQ resource adapter and understand its configuration properties.

- The IBM MQ resource adapter is supplied as a Resource Archive (RAR) file called **wmq.jmsra.rar**. You can obtain the **wmq.jmsra.rar** file from **/opt/mqm/java/lib/jca/wmq.jmsra.rar**. See JBoss EAP supported configurations for information about the specific versions that are supported for each release of JBoss EAP.

- You must know the following IBM MQ configuration values. Refer to the IBM MQ product documentation for details about these values.

  - *MQ_QUEUE_MANAGER*: The name of the IBM MQ queue manager
  - *MQ_HOST_NAME*: The host name used to connect to the IBM MQ queue manager
  - *MQ_CHANNEL_NAME*: The server channel used to connect to the IBM MQ queue manager
  - *MQ_QUEUE_NAME*: The name of the destination queue
  - *MQ_TOPIC_NAME*: The name of the destination topic
  - *MQ_PORT*: The port used to connect to the IBM MQ queue manager
  - *MQ_CLIENT*: The transport type

- For outbound connections, you must also be familiar with the following configuration value:

- *MQ_CONNECTIONFACTORY_NAME*: The name of the connection factory instance that will provide the connection to the remote system

Procedure

NOTE

The following are default configurations provided by IBM and are subject to change. Please refer to IBM MQ documentation for more information.

1. First, deploy the resource adapter manually by copying the **wmq.jmsra.rar** file to the ***EAP_HOME*/standalone/deployments/** directory.

2. Next, use the management CLI to add the resource adapter and configure it.

```
/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar:add(archive=wmq.jmsra.rar,
transaction-support=XATransaction)
```

Note that the **transaction-support** element was set to **XATransaction**. When using transactions, be sure to supply the security domain of the XA recovery datasource, as in the example below.

```
/subsystem=resource-adapters/resource-adapter=test/connection-definitions=test:write-
attribute(name=recovery-security-domain,value=myDomain)
```

For more information about XA Recovery see Configuring XA Recovery in the JBoss EAP *Configuration Guide*.

For non-transactional deployments, change the value of **transaction-support** to **NoTransaction**.

```
/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar:add(archive=wmq.jmsra.rar,
transaction-support=NoTransaction)
```

3. Now that the resource adapter is created, you can add the necessary configuration elements to it.

a. Add an **admin-object** for queues and configure its properties.

```
/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar/admin-objects=queue-
ao:add(class-name=com.ibm.mq.connector.outbound.MQQueueProxy, jndi-
name=java:jboss/MQ_QUEUE_NAME)

/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar/admin-objects=queue-
ao/config-properties=baseQueueName:add(value=MQ_QUEUE_NAME)

/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar/admin-objects=queue-
ao/config-properties=baseQueueManagerName:add(value=MQ_QUEUE_MANAGER)
```

b. Add an **admin-object** for topics and configure its properties.

```
/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar/admin-objects=topic-
ao:add(class-name=com.ibm.mq.connector.outbound.MQTopicProxy, jndi-
name=java:jboss/MQ_TOPIC_NAME)
```

```
/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar/admin-objects=topic-ao/config-properties=baseTopicName:add(value=MQ_TOPIC_NAME)

/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar/admin-objects=topic-ao/config-properties=brokerPubQueueManager:add(value=MQ_QUEUE_MANAGER)
```

c. Add a connection definition for a managed connection factory and configure its properties.

```
/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar/connection-definitions=mq-cd:add(class-name=com.ibm.mq.connector.outbound.ManagedConnectionFactoryImpl, jndi-name=java:jboss/MQ_CONNECTIONFACTORY_NAME, tracking=false)

/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar/connection-definitions=mq-cd/config-properties=hostName:add(value=MQ_HOST_NAME)

/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar/connection-definitions=mq-cd/config-properties=port:add(value=MQ_PORT)

/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar/connection-definitions=mq-cd/config-properties=channel:add(value=MQ_CHANNEL_NAME)

/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar/connection-definitions=mq-cd/config-properties=transportType:add(value=MQ_CLIENT)

/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar/connection-definitions=mq-cd/config-properties=queueManager:add(value=MQ_QUEUE_MANAGER)
```

4. If you want to change the default provider for the EJB3 messaging system in JBoss EAP from JBoss EAP 7 messaging to IBM MQ, use the management CLI to modify the **ejb3** subsystem as follows:

```
/subsystem=ejb3:write-attribute(name=default-resource-adapter-name,value=wmq.jmsra.rar)
```

5. Configure the **@ActivationConfigProperty** and **@ResourceAdapter** annotations in the MDB code as follows:

```
@MessageDriven(name="IbmMqMdb", activationConfig = {
   @ActivationConfigProperty(propertyName = "destinationType",propertyValue = "javax.jms.Queue"),
   @ActivationConfigProperty(propertyName = "useJNDI", propertyValue = "false"),
   @ActivationConfigProperty(propertyName = "hostName", propertyValue = "MQ_HOST_NAME"),
   @ActivationConfigProperty(propertyName = "port", propertyValue = "MQ_PORT"),
   @ActivationConfigProperty(propertyName = "channel", propertyValue = "MQ_CHANNEL_NAME"),
   @ActivationConfigProperty(propertyName = "queueManager", propertyValue = "MQ_QUEUE_MANAGER"),
   @ActivationConfigProperty(propertyName = "destination", propertyValue = "MQ_QUEUE_NAME"),
   @ActivationConfigProperty(propertyName = "transportType", propertyValue = "MQ_CLIENT")
})
```

```
@ResourceAdapter(value = "wmq.jmsra-VERSION.rar")
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class IbmMqMdb implements MessageListener {
}
```

Be sure to replace the *VERSION* in the **@ResourceAdapter** value with the actual version in the name of the RAR.

6. Activate your resource adapter:

```
/subsystem=resource-adapters/resource-adapter=wmq.jmsra.rar:activate()
```

31.6.1. Limitations and Known Issues with the IBM MQ Resource Adapters

The following table lists known issues with the IBM MQ resource adapters. A checkmark (✔) in the version column indicates the issue is a problem for that version of the resource adapter.

Table 31.1. Known Issues with the IBM MQ Resource Adapters

JIRA	Description of Issue	IBM MQ 8	IBM MQ 9
JBEAP-503	The IBM MQ resource adapter returns different String values for the **Queue.toString()** and **QueueBrowser.getQueue().toString()** methods. **Queue** is instance of the **com.ibm.mq.connector.outbound.MQQueueProxy** class, which is different from the **com.ibm.mq.jms.MQQueue** class that is returned by the **QueueBrowser.htmlQueueBrowser.getQueue()** method. These classes contain different implementations of the **toString()** method. Be aware that you cannot rely on these **toString()** methods to return the same value.	✔	✔

JIRA	Description of Issue	IBM MQ 8	IBM MQ 9
JBEAP-511, JBEAP-550, JBEAP-3686	The following restrictions apply to message property names for IBM MQ. ● In the **activation-config** section of the deployment descriptor, you must not configure the **destinationName** property using special characters such as _, **&**, or **|**. Use of these characters causes the MDB deployment to fail with a **com.ibm.msg.client.jms.DetailedInvalidDestinationException** exception. ● In the **activation-config** section of the deployment descriptor, you must not configure the **destinationName** property using the **java:/** prefix. Use of this prefix causes the MDB deployment to fail with a **com.ibm.msg.client.jms.DetailedInvalidDestinationException** exception. ● A property must not begin with "JMS" or "usr.JMS" as they are reserved for use by IBM MQ Jakarta Messaging classes. Exceptions are noted on the IBM Knowledge Center website. See Property name restrictions for IBM MQ, Version 8.0and Property name restrictions for IBM MQ, Version 9.0 on the IBM Knowledge Center website for the complete list of message property name restrictions for each version of the resource adapter.	✔	✔
JBEAP-549	When specifying the **destination** property name value for an MDB using the **@ActivationConfigProperty** annotation, you must use all upper case letters. For example: `@ActivationConfigProperty(propertyName = "destination", propertyValue = "QUEUE")`	✔	✔

JBEAP-624	If the IBM MQ resource adapter is used to create a connection factory in a Jakarta EE deployment using the **@JMSConnectionFactoryDefinition** annotation, you must specify the **resourceAdapter** property. Otherwise, the deployment will fail.
	
	`@JMSConnectionFactoryDefinition(`
	`    name = "java:/jms/WMQConnectionFactory",`
	`    interfaceName = "javax.jms.ConnectionFactory",`
	`    resourceAdapter = "wmq.jmsra",`
	`    properties = {`
	`        "channel=<channel>",`
	`        "hostName=<hostname_wmq_broker>",`
	`        "transportType=<transport_type>",`
	`        "queueManager=<queue_manager>"`
	`    }`
	`)`
		✔	✔

JIRA	Description of Issue	IBM MQ 8	IBM MQ 9
JBEAP-2339	The IBM MQ resource adapter is able to read messages from queues and topics even before the connection has started. This means a consumer can consume messages before the connection is started. To avoid hitting this issue, use connection factories created by the remote IBM MQ broker using the **external-context** and not connection factories created by IBM MQ resource adapter.	✔	✔

JBEAP-3685	Once **<transaction-support>XATransaction</transaction-support>** is set, a **JMSContext** is always **JMSContext.SESSION_TRANSACTED**, whether it was created using injection or manually.
	
	In the following code example, the **@JMSSessionMode(JMSContext.DUPS_OK_ACKNOWLEDGE)** is ignored and the **JMSContext** remains at **JMSContext.SESSION_TRANSACTED**.
	
	`@Inject`
	`@JMSConnectionFactory("jms/CF")`
	`@JMSPasswordCredential(userName="myusername", password="mypassword")`
	`@JMSSessionMode(JMSContext.DUPS_OK_ACKNOWLEDGE)`
	`transient JMSContext context3;`
		✔	✔

JBEAP-14633	According to the Jakarta Messaging specification, the **QueueSession** interface cannot be used to create objects specific to the publish/subscribe domain and certain methods that inherit from **Session** should throw an **javax.jms.IllegalStateException**. One such method is such **QueueSession.createTemporaryTopic()**. Instead of throwing an **javax.jms.IllegalStateException**, the IBM MQ resource adapter throws a **java.lang.NullPointerException**.	✔	✔
JBEAP-14634	The **MQTopicProxy.getTopicName()** returns different topic name than was set by the IBM MQ broker. For example, if the topic name was set to **topic://MYTOPIC?XMSC_WMQ_BROKER_PUBQ_QMGR=QM**, the **MQTopicProxy** returns **topic://MYTOPIC**.	✔	✔
JBEAP-14636	The default **autoStart** setting for the **JMSContext** is **false**, meaning the underlying connection used by the **JMSContext** is not started automatically when a consumer is created. This setting should default to **true**.	✔	✔

JIRA	Description of Issue	IBM MQ 8	IBM MQ 9

JBEAP-14640	The IBM MQ resource adapter throws **DetailedJMSException** instead of a **JMSSecurityException** when invalid credentials are used and logs the following error to the server console.
	`WARN [org.jboss.jca.core.connectionmanager.pool.strategy.PoolByCri] (EJB default - 7) IJ000604: Throwable while attempting to get a new connection: null: com.ibm.mq.connector.DetailedResourceException: MQJCA1011: Failed to allocate a {JMS} connection., error code: MQJCA1011 An internal error caused an attempt to allocate a connection to fail. See the linked exception for details of the failure.`
	The following is an example of code that can cause this issue.
	`QueueConnection qc = queueConnectionFactory.createQueueConnection("invalidUserName", "invalidPassword");`
		✔	✔

JBEAP-14642	Due to an invalid class cast conversion by the resource adapter in the **MQMessageProducer.send(Destination destination, Message message)** and **MQMessageProducer.send(Destination destination, Message message, int deliveryMode, int priority, long timeToLive, CompletionListener completionListener)** methods, the IBM MQ resource adapter throws a **JMSException** and logs the following error message to the server console. `SVR-ERROR: Expected JMSException, received com.ibm.mq.connector.outbound.MQQueueProxy cannot be cast to com.ibm.mq.jms.MQDestination` This is because the JNDI name used in the queue or topic lookup is **com.ibm.mq.connector.outbound.MQQueueProxy/MQTopicProxy**.	✔	✔
JBEAP-14643	The **setDeliveryDelay(expDeliveryDelay)** method on the **JMSProducer** interface does not change the setting. After calling this method, it remains at the default setting of **0**.	✔	✔
JBEAP-14670	If work is done on a **QueueSession** that was created prior to a **UserTransaction.begin()**, that work is not considered part of the transaction. This means that any message sent to the queue using this session is not committed by a **UserTransaction.commit()**, and after a **UserTransaction.rollback()**, the message remains on the queue.	✔	✔

JIRA	Description of Issue	IBM MQ 8	IBM MQ 9
JBEAP-14675	If you close a connection and then immediately create a **JMSContext** with the same **clientID**, the IBM MQ resource adapter intermittently logs the following error to the server console. `ERROR [io.undertow.request] (default task-1) UT005023: Exception handling request to /jmsServlet-1.0-SNAPSHOT/: com.ibm.msg.client.jms.DetailedJMSRuntimeException: MQJCA0002: An exception occurred in the IBM MQ layer. See the linked exception for details. A call to IBM MQ classes for Java(tm) caused an exception to be thrown.` This issue does not occur when there is a delay in creating the new **JMSContext** after the connection with the same **clientID** is closed.	✔	✔
JBEAP-15535	If a stateful session bean tries to send a message to a topic while in a container managed transaction (CMT), the message send fails with the following message. `SVR-ERROR: com.ibm.msg.client.jms.DetailedJMSException: JMSWMQ2007: Failed to send a message to destination 'MDB_NAME TOPIC_NAME'` The stack trace shows it to be caused by the following exception. `com.ibm.mq.MQException: JMSCMQ0001: IBM MQ call failed with compcode '2' ('MQCC_FAILED') reason '2072' ('MQRC_SYNCPOINT_NOT_AVAILABLE')`	✔	✔
JBEAP-20758	When you deploy either the IBM MQ 8 or IBM MQ 9 resource adapter, **wmq.jmsra.rar**, on JBoss EAP, the following error message displays on the server console: `WARN  [org.jboss.as.connector.deployers.RADeployer] (MSC service thread 1-8) IJ020017: Invalid archive: file:/<path-to-jboss>/jboss-eap-7.4/standalone/tmp/vfs/temp/tempa02bdd5ee254e590/content-135e13d4f38704fc/contents/` The IBM MQ v9.0.0.4 resource adapter was tested as part of the Jakarta Messaging providers tests for JBoss EAP 7.4. You can choose to ignore this warning message or you can disable the archive validation by setting the **enabled** attribute to **false**. For example: `/subsystem=jca/archive-validation=archive-validation:write-attribute(name=enabled, value=false)`	✔	✔

31.7. DEPLOYING A GENERIC JAKARTA MESSAGING RESOURCE ADAPTER

JBoss EAP can be configured to work with third-party Jakarta Messaging providers; however, not all Jakarta Messaging providers produce a Jakarta Messaging Jakarta Connectors resource adapter for integration with Jakarta application platforms. This procedure covers the steps required to configure the generic Jakarta Messaging resource adapter included in JBoss EAP to connect to a Jakarta Messaging provider. In this procedure, Tibco EMS 8 is used as an example Jakarta Messaging provider. Other Jakarta Messaging providers may require different configuration.

IMPORTANT

Before using the generic Jakarta Messaging resource adapter, check with the Jakarta Messaging provider to see if they have their own resource adapter that can be used with JBoss EAP. The generic Jakarta Messaging Jakarta Connectors resource adapter should only be used when a Jakarta Messaging provider does not provide its own resource adapter.

Before you can configure a generic resource adapter, you will need to do the following:

- Your Jakarta Messaging provider server must already be configured and ready for use. Any binaries required for the provider's Jakarta Messaging implementation will be needed.
- You will need to know the values of the following Jakarta Messaging provider properties to be able to look up its Jakarta Messaging resources, such as connection factories, queues or topics.
    - **java.naming.factory.initial**
    - **java.naming.provider.url**
    - **java.naming.factory.url.pkgs**

In the example XML used in this procedure, these parameters are written as **PROVIDER_FACTORY_INITIAL**, **PROVIDER_URL**, and **PROVIDER_CONNECTION_FACTORY** respectively. Replace these placeholders with the Jakarta Messaging provider values for your environment.

31.7.1. Configure a Generic Jakarta Messaging Resource Adapter for Use with a Third-party Jakarta Messaging Provider

1. Create and configure the resource adapter module.
   Create a JBoss EAP module that contains all the libraries required to connect and communicate with the Jakarta Messaging provider. This module will be named *org.jboss.genericjms.provider*.
    - Create the following directory structure: ***EAP_HOME*/modules/org/jboss/genericjms/provider/main**
    - Copy the binaries required for the provider's Jakarta Messaging implementation to ***EAP_HOME*/modules/org/jboss/genericjms/provider/main**.

   NOTE

   For Tibco EMS, the binaries required are **tibjms.jar** and **tibcrypt.jar** from the Tibco installation's **lib** directory.

- Create a **module.xml** file in ***EAP_HOME*/modules/org/jboss/genericjms/provider/main** as below, listing the JAR files from the previous steps as resources:

```
<module xmlns="urn:jboss:module:1.5" name="org.jboss.genericjms.provider">
  <resources>
    <!-- all jars required by the Jakarta Messaging provider, in this case Tibco -->
    <resource-root path="tibjms.jar"/>
    <resource-root path="tibcrypt.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
    <module name="javax.jms.api"/>
  </dependencies>
</module>
```

- Add the module to the **ee** subsystem using the following CLI command:

```
/subsystem=ee:list-add(name=global-modules, value={"name" =>
"org.jboss.genericjms.provider", "slot" =>"main"}
```

2. Create and configure a Java Naming and Directory Interface external context to the Jakarta Messaging provider.
The Jakarta Messaging resources, such as connection factories and destinations, are looked up in the Jakarta Messaging provider. Add an external context in the JBoss EAP instance so that any *local* lookup for this resource will automatically look up the resource on the remote Jakarta Messaging provider.

NOTE

In this procedure, ***EAP_HOME*/standalone/configuration/standalone-full.xml** is used as the JBoss EAP configuration file.

Use the management CLI to create an external Java Naming and Directory Interface context and include its configuration properties. The properties in the example below should be replaced by the correct value to connect to the remote Jakarta Messaging provider. For example, some Jakarta Messaging providers, such as Tibco EMS, do not support the Java Naming and Directory Interface **lookup(Name)** method. In these cases, add the **org.jboss.as.naming.lookup.by.string** property with a value of **true** to work around this issue. Check the adapter's documentation for information on required properties and their values.

```
/subsystem=naming/binding="java:global/remoteJMS":add(binding-type=external-
context,module=org.jboss.genericjms.provider,class=javax.naming.InitialContext,environment=
[java.naming.factory.initial=com.tibco.tibjms.naming.TibjmsInitialContextFactory,java.naming.prc
vider.url=tcp://<hostname>:7222,org.jboss.as.naming.lookup.by.string=true])
```

With the external context configured properly, any Java Naming and Directory Interface lookup to a resource starting with **java:global/remoteJMS/** will be done on the remote Jakarta Messaging provider. As an example, if a message-driven bean performs a Java Naming and Directory Interface lookup for **java:global/remoteJMS/Queue1**, the external context will connect to the remote Jakarta Messaging provider and perform a lookup for the **Queue1** resource.

Alternatively, you can make a Java Naming and Directory Interface lookup to the remote server without using an **external-context** when looking up the Java Naming and Directory Interface name. To do so, use the CLI to create a new binding that references the **external-context**, as in the example below.

```
/subsystem=naming/binding=java\:\/jms\/queue\/myQueue:add(binding-type=lookup,
lookup=java:global/remoteJMS/jms/queue/myQueue)
```

In the example above, an application that does a Java Naming and Directory Interface lookup for **java:/jms/queue/myQueue** will locate the queue named **myQueue** on the remote server.

3. Create the generic Jakarta Messaging resource adapter.
   Use the management CLI to create the resource adapter

```
/subsystem=resource-adapters/resource-adapter=generic-
ra:add(module=org.jboss.genericjms,transaction-support=XATransaction)
```

4. Configure the generic Jakarta Messaging resource adapter.
   Use the management CLI to configure the resource adapter's **connection-definition** and other elements.

```
/subsystem=resource-adapters/resource-adapter=generic-ra/connection-definitions=tibco-
cd:add(class-name=org.jboss.resource.adapter.jms.JmsManagedConnectionFactory, jndi-
name=java:/jms/XAQCF)

/subsystem=resource-adapters/resource-adapter=generic-ra/connection-definitions=tibco-
cd/config-properties=ConnectionFactory:add(value=XAQCF)

/subsystem=resource-adapters/resource-adapter=generic-ra/connection-definitions=tibco-
cd/config-
properties=JndiParameters:add(value="java.naming.factory.initial=com.tibco.tibjms.naming.Tibj
msInitialContextFactory;java.naming.provider.url=tcp://<hostname>:7222")

/subsystem=resource-adapters/resource-adapter=generic-ra/connection-definitions=tibco-
cd:write-attribute(name=security-application,value=true)
```

5. Configure the default message-driven bean pool in the **ejb3** subsystem to use the generic resource adapter.

```
/subsystem=ejb3:write-attribute(name=default-resource-adapter-name, value=generic-ra)
```

The generic Jakarta Messaging resource adapter is now configured and ready for use. Below is an example of using the resource adapter when creating a new message-driven bean.

Example: Code Using the Generic Resource Adapter

```
@MessageDriven(name = "HelloWorldQueueMDB", activationConfig = {
  // The generic Jakarta Messaging resource adapter requires the  Java Naming and Directory
Interface bindings
  // for the actual remote connection factory and destination
  @ActivationConfigProperty(propertyName = "connectionFactory", propertyValue =
"java:global/remoteJMS/XAQCF"),
  @ActivationConfigProperty(propertyName = "destination", propertyValue =
"java:global/remoteJMS/Queue1"),
  @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue"),
  @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue = "Auto-
acknowledge") })
  public class HelloWorldQueueMDB implements MessageListener {
  public void onMessage(Message message) {
  // called every time a message is received from the _Queue1_ queue on the Jakarta Messaging
provider.
  }
}
```

IMPORTANT

When using the generic Jakarta Messaging resource adapter, ensure you set the session to be transacted, to avoid a potential **NullPointerException** error. The error occurs because the generic Jakarta Messaging resource adapter attempts processing of parameters, when the Jakarta EE specification states that they are *not* to be processed. This is accomplished by doing the following: **connection.createSession(true, Session.SESSION_TRANSACTED);**

You can also use the pooled connection factory from the resource adapter:

```
@Resource(lookup = "java:/jms/XAQCF")
private ConnectionFactory cf;
```

It is not possible to inject a resource from an external context directly but it is possible to inject an external context and then perform a lookup. For example, a lookup for a queue deployed in a Tibco EMS broker would be as follows.

```
@Resource(lookup = "java:global/remoteJMS")
private Context context;
...
Queue queue = (Queue) context.lookup("Queue1")
```

31.8. USING THE RESOURCE ANNOTATION

Using the **@Resource** annotation, Jakarta Enterprise Beans can directly inject Jakarta Messaging resources or connection factories. You can specify the following parameters using the **@Resource** annotations:

- **lookup**
- **name**
- **mappedName**

To inject a resource, you must specify the Java Naming and Directory Interface (JNDI) name of the resource in one of these parameters.

31.8.1. Injecting Jakarta Messaging Resources

1. Define your queue as shown below:

```
<jms-queue name="OutQueue" entries="jms/queue/OutQueue
java:jboss/exported/jms/queue/OutQueue"/>
```

2. Inject this queue by specifying its Java Naming and Directory Interface name in the **lookup**, **name**, or **mappedName** parameter of the **@Resource** annotation. For example:

```
@Resource(lookup = "java:jboss/exported/jms/queue/OutQueue")
public Queue myOutQueue;
```

31.8.2. Injecting Connection Factories

1. Define your connection factory as shown below. The example shows a **JmsXA** pooled connection factory.

```
<pooled-connection-factory name="activemq-ra" entries="java:/JmsXA
java:jboss/DefaultJMSConnectionFactory" connectors="in-vm" transaction="xa"/>
```

2. Inject the default **activemq-ra** pooled connection factory as shown below:

```
@Resource(lookup = "java:/JmsXA")
private ConnectionFactory cf;
```

31.8.3. The Limitations and Known Issues for the Generic Jakarta Messaging Resource Adapter

- The Jakarta Messaging API does not provide a programmatic way to create the Jakarta Messaging resources, only the features that are defined in the Jakarta Messaging 2.0 specification are supported. For more information about the specification, see Jakarta Messaging 2.0 specification.
  - EE.5.18.4 Jakarta Messaging Connection Factory Resource Definition
    This is the ability for an application to define a Jakarta Messaging **ConnectionFactory** resource.
  - EE.5.18.5 Jakarta Messaging Destination Definition
    This is the ability for an application to define a Jakarta Messaging **Destination** resource.

CHAPTER 32. BACKWARD AND FORWARD COMPATIBILITY

JBoss EAP supports both backward and forward compatibility with legacy versions of JBoss EAP that were using HornetQ as their messaging brokers, such as JBoss EAP 6. These two compatibility modes are provided by the JBoss EAP built-in messaging server, ActiveMQ Artemis, that supports the HornetQ's core protocol.

- Forward compatibility: Legacy Jakarta Messaging clients using HornetQ can connect to a JBoss EAP 7 server running ActiveMQ Artemis.
- Backward compatibility: JBoss EAP 7 Jakarta Messaging clients using JBoss EAP messaging can connect to the legacy JBoss EAP 6 server running HornetQ.

32.1. FORWARD COMPATIBILITY

Forward compatibility requires no code changes to legacy JBoss EAP 6 Jakarta Messaging clients. Support is provided by the JBoss EAP **messaging-activemq** subsystem and its resources. To enable support of forward compatibility make the following changes to the configuration of the JBoss EAP 7 server. Example management CLI commands for a standalone server are provided for each step.

- Create a **socket-binding** the listens on port 4447 for remote legacy clients.

```
/socket-binding-group=standard-sockets/socket-binding=legacy-remoting:add(port=4447)
```

- Create a legacy **remote-connector** that will use the **socket-binding** created in the previous step. This is required for JNDI lookups.

```
/subsystem=remoting/connector=legacy-remoting-connector:add(socket-binding=legacy-
remoting)
```

- Set up a legacy messaging **socket-binding** that listens on port 5445.

```
/socket-binding-group=standard-sockets/socket-binding=legacy-messaging:add(port=5445)
```

- Set up a **remote-connector** and a **remote-acceptor** in the **messaging-activemq** subsystem that use the binding from the previous step.

```
/subsystem=messaging-activemq/server=default/remote-connector=legacy-messaging-
connector:add(socket-binding=legacy-messaging)

/subsystem=messaging-activemq/server=default/remote-acceptor=legacy-messaging-
acceptor:add(socket-binding=legacy-messaging)
```

- Create a legacy HornetQ Jakarta Messaging ConnectionFactory in the **legacy-connection-factory** element of the **messaging-activemq** subsystem.

```
/subsystem=messaging-activemq/server=default/legacy-connection-factory=legacy-
discovery:add(entries=[java:jboss/exported/jms/LegacyRemoteConnectionFactory],
connectors=[legacy-messaging-connector])
```

- Create legacy HornetQ Jakarta Messaging destinations and include **legacy-entries** attributes to the **jms-queue** or **jms-topic** resources.

```
jms-queue add --queue-address=myQueue --entries=[java:jboss/exported/jms/myQueue-new] --legacy-entries=[java:jboss/exported/jms/myQueue]

jms-topic add --topic-address=myTopic --entries=[java:jboss/exported/jms/myTopic-new] --legacy-entries=[java:jboss/exported/jms/myTopic]
```

You can add **legacy-entries** to an existing queue or topic by following the below example.

```
/subsystem=messaging-activemq/server=default/jms-queue=myQueue:write-attribute(name=legacy-entries,value=[java:jboss/exported/jms/myQueue])
```

While the **entries** attributes are used by JBoss EAP messaging Jakarta Messaging clients, the **legacy-entries** are used by the legacy HornetQ Jakarta Messaging clients. Legacy Jakarta Messaging clients look up this legacy Jakarta Messaging resource to communicate with JBoss EAP 7.

NOTE

To avoid any code change in the legacy Jakarta Messaging clients, the legacy JNDI entries configured in the **messaging-activemq** subsystem must match the lookup expected by the legacy Jakarta Messaging client.

Management CLI migrate Operation

When you run the management CLI **migrate** operation to update your **messaging** subsystem configuration, if the boolean argument **add-legacy-entries** is set to **true**, the **messaging-activemq** subsystem creates the **legacy-connection-factory** resource and adds **legacy-entries** to the **jms-queue** and **jms-topic** resources. The legacy entries in the migrated **messaging-activemq** subsystem will correspond to the entries specified in the legacy **messaging** subsystem and the regular entries are created with a **-new** suffix.

If the boolean argument **add-legacy-entries** is set to **false** when you run the **migrate** operation, no legacy resources are created in the **messaging-activemq** subsystem and legacy Jakarta Messaging clients will not be able to communicate with the JBoss EAP 7 servers.

32.2. BACKWARD COMPATIBILITY

Backward compatibility requires no configuration change in the legacy JBoss EAP 7 servers. JBoss EAP 7 Jakarta Messaging clients do not look up resources on the legacy server, but instead use client-side JNDI to create Jakarta Messaging resources. JBoss EAP 7 Jakarta Messaging clients can then use these resources to communicate with the legacy server using the HornetQ core protocol.

WARNING

JBoss EAP 7 client connections to a JBoss EAP 5 server are currently not supported.

JBoss EAP messaging supports client-side JNDI to create Jakarta Messaging **ConnectionFactory** and **Destination** resources.

For example, if a JBoss EAP 7 Jakarta Messaging client wants to communicate with a legacy server using a Jakarta Messaging queue named "myQueue", it must use the following properties to configure its JNDI **InitialContext**:

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.ActiveMQInitialContextFactory
connectionFactory.jms/ConnectionFactory=tcp://<legacy server address>:5445? \

protocolManagerFactoryStr=org.apache.activemq.artemis.core.protocol.hornetq.client.HornetQClientPr
otocolManagerFactory
queue.jms/myQueue=myQueue
```

The client can then use the **jms/ConnectionFactory** name to create the Jakarta Messaging **ConnectionFactory** and use the **jms/myQueue** to create the Jakarta Messaging **Queue**. Note that the property **protocolManagerFactoryStr=org.apache.activemq.artemis.core.protocol.hornetq.client.HornetQClientProtocolManagerFactory** is mandatory when specifying the URL of the legacy connection factory. This allows the JBoss EAP messaging Jakarta Messaging client to communicate with the HornetQ broker in the legacy server.

PART IV. PERFORMANCE TUNING

CHAPTER 33. MONITORING MESSAGING STATISTICS

When statistics collection is enabled for a messaging server in the **messaging-activemq** subsystem, you can view runtime statistics for resources on the messaging server.

33.1. ENABLING MESSAGING STATISTICS

Because it can negatively impact performance, statistics collection for the **messaging-activemq** subsystem is *not* enabled by default. You do not need to enable queue statistics to obtain basic information, such as the number of messages on a queue or the number of messages added to a queue. Those statistics are available using queue attributes without requiring that you set **statistics-enabled** to **true**.

You can enable additional statistics collection using the management CLI or the management console.

Enable Messaging Statistics Using the Management CLI

The following management CLI command enables the collection of statistics for the **default** messaging server.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=statistics-enabled,value=true)
```

Pooled connection factory statistics are enabled separately from the other messaging server statistics. Use the following command to enable statistics for a pooled connection factory.

```
/subsystem=messaging-activemq/server=default/pooled-connection-factory=activemq-ra:write-
attribute(name=statistics-enabled,value=true)
```

Reload the server for the changes to take effect.

Enable Messaging Statistics Using the Management Console

Use the following steps to enable statistics collection for a messaging server using the management console.

1. Navigate to **Configuration** → **Subsystems** → **Messaging (ActiveMQ)** → **Server**.
2. Select the server and click **View**.
3. Click **Edit** under the **Statistics** tab.
4. Set the **Statistics Enabled** field to **ON** and click **Save**.

Pooled connection factory statistics are enabled separately from the other messaging server statistics. Use the following steps to enable statistics collection for a pooled connection factory.

1. Navigate to **Configuration** → **Subsystems** → **Messaging (ActiveMQ)** → **Server**.
2. Select the server, select **Connections**, and click **View**.
3. Select the **Pooled Connection Factory** tab.
4. Select the pooled connection factory and click **Edit** under the **Attributes** tab.
5. Set the **Statistics Enabled** field to **ON** and click **Save**.
6. Reload the server for the changes to take effect.

33.2. VIEWING MESSAGING STATISTICS

You can view runtime statistics for a messaging server using the management CLI or management console.

View Messaging Statistics Using the Management CLI

You can view messaging statistics using the following management CLI commands. Be sure to include the **include-runtime=true** argument as statistics are runtime information.

- View statistics for a queue.

```
/subsystem=messaging-activemq/server=default/jms-queue=DLQ:read-resource(include-
runtime=true)
{
    "outcome" => "success",
    "result" => {
        "consumer-count" => 0,
        "dead-letter-address" => "jms.queue.DLQ",
        "delivering-count" => 0,
        "durable" => true,
        ...
    }
}
```

- View statistics for a topic.

```
/subsystem=messaging-activemq/server=default/jms-topic=testTopic:read-resource(include-
runtime=true)
{
    "outcome" => "success",
    "result" => {
        "delivering-count" => 0,
        "durable-message-count" => 0,
        "durable-subscription-count" => 0,
        ...
    }
}
```

- View statistics for a pooled connection factory.

```
/subsystem=messaging-activemq/server=default/pooled-connection-factory=activemq-
ra/statistics=pool:read-resource(include-runtime=true)
{
    "outcome" => "success",
    "result" => {
        "ActiveCount" => 1,
        "AvailableCount" => 20,
        "AverageBlockingTime" => 0L,
        "AverageCreationTime" => 13L,
        "AverageGetTime" => 14L,
        ...
    }
}
```

NOTE

Pooled connection factory statistics are enabled separately from the other messaging server statistics. See Enabling Messaging Statistics for instructions.

View Messaging Statistics Using the Management Console

To view messaging statistics from the management console, navigate to the **Messaging (ActiveMQ)** subsystem from the **Runtime** tab, and select the server. Select a destination to view its statistics.

NOTE

The **Prepared Transactions** page is where you can view, commit, and roll back prepared transactions. See Managing Messaging Journal Prepared Transactions for more information.

See Messaging Statistics for a detailed list of all available statistics.

33.3. CONFIGURING MESSAGE COUNTERS

You can configure the following message counter attributes for a messaging server.

- **message-counter-max-day-history**: The number of days the message counter history is kept.
- **message-counter-sample-period**: How often, in milliseconds, the queue is sampled.

The management CLI command to configure these options uses the following syntax. Be sure to replace ***STATISTICS_NAME*** and ***STATISTICS_VALUE*** with the statistic name and value you want to configure.

```
/subsystem=messaging-activemq/server=default::write-
attribute(name=STATISTICS_NAME,value=STATISTICS_VALUE)
```

For example, use the following commands to set the **message-counter-max-day-history** to five days and the **message-counter-sample-period** to two seconds.

```
/subsystem=messaging-activemq/server=default:write-attribute(name=message-counter-max-day-
history,value=5)
/subsystem=messaging-activemq/server=default:write-attribute(name=message-counter-sample-
period,value=2000)
```

33.4. VIEWING THE MESSAGE COUNTER AND HISTORY FOR A QUEUE

You can view the message counter and message counter history for a queue using the following management CLI operations.

- **list-message-counter-as-json**
- **list-message-counter-as-html**
- **list-message-counter-history-as-json**
- **list-message-counter-history-as-html**

The management CLI command to use display these values uses the following syntax. Be sure to replace ***QUEUE_NAME*** and ***OPERATION_NAME*** with the queue name and operation you want to use.

```
/subsystem=messaging-activemq/server=default/jms-queue=QUEUE_NAME:OPERATION_NAME
```

For example, use the following command to view the message counter for the **TestQueue** queue in JSON format.

```
/subsystem=messaging-activemq/server=default/jms-queue=TestQueue:list-message-counter-as-json
{
    "outcome" => "success",
    "result" => "
{\"destinationName\":\"TestQueue\",\"destinationSubscription\":null,\"destinationDurable\":true,\"count\":
0,\"countDelta\":0,\"messageCount\":0,\"messageCountDelta\":0,\"lastAddTimestamp\":\"12/31/69
7:00:00 PM\",\"updateTimestamp\":\"2/20/18 2:24:05 PM\"}"
}
```

33.5. RESET THE MESSAGE COUNTER FOR A QUEUE

You can reset the message counter for a queue using the **reset-message-counter** management CLI operation.

```
/subsystem=messaging-activemq/server=default/jms-queue=TestQueue:reset-message-counter
{
    "outcome" => "success",
    "result" => undefined
}
```

33.6. RUNTIME OPERATIONS USING THE MANAGEMENT CONSOLE

Using the management console you can:

- Perform forced failover to another messaging server
- Reset all message counters for a messaging server
- Reset all message counters history for a messaging server
- View information related to a messaging server
- Close connections for a messaging server
- Roll back transactions
- Commit transactions

Performing Forced Failover to Another Messaging Server

1. Access the management console and navigate to **Server** using either of the following:
    - **Runtime→ Browse By → Hosts → Host → Server**
    - **Runtime→ Browse By → Server Groups → Server Group → Server**

2. Click **Messaging ActiveMQ → Server**
3. Click the arrow button next to **View** and click **Force Failover**.
4. On the **Force Failover** window, click **Yes**.

Resetting All Message Counters for a Messaging Server

1. Access the management console and navigate to **Server** using either of the following:
   - **Runtime→ Browse By → Hosts → Host → Server**
   - **Runtime→ Browse By → Server Groups → Server Group → Server**
2. Click **Messaging ActiveMQ → Server**
3. Click the arrow button next to **View** and click **Reset**.
4. On the **Reset** window, click the toggle button next to **Reset all message counters** to enable the functionality.
   The button now displays **ON** in a blue background.
5. Click **Reset**.

Resetting Message Counters History for a Messaging Server

1. Access the management console and navigate to **Server** using either of the following:
   - **Runtime→ Browse By → Hosts → Host → Server**
   - **Runtime→ Browse By → Server Groups → Server Group → Server**
2. Click **Messaging ActiveMQ → Server**
3. Click the arrow button next to **View** and click **Reset**.
4. On the **Reset** window, click the toggle button next to **Reset all message counters history** to enable the functionality.
   The button now displays **ON** in a blue background.
5. Click **Reset**.

Viewing Information Related to a Messaging Server

Using the management console you can view a list of the following information related to a messaging server:

- Connections
- Consumers
- Producers
- Connectors
- Roles
- Transactions

To view information related to a messaging server:

1. Access the management console and navigate to **Server** using either of the following:
   - **Runtime→ Browse By → Hosts → Host → Server**
   - **Runtime→ Browse By → Server Groups → Server Group → Server**
2. Click **Messaging ActiveMQ → Server** and then click **View**.
3. Click the appropriate item on the navigation pane to view a list of the item on the right pane.

Closing Connections for a Messaging Server

You can close connections by providing an IP address, an ActiveMQ address match or a user name.

To close connections for a messaging server:

1. Access the management console and navigate to **Server** using either of the following:
   - **Runtime→ Browse By → Hosts → Host → Server**
   - **Runtime→ Browse By → Server Groups → Server Group → Server**
2. Click **Messaging ActiveMQ → Server** and then click **View**.
3. On the navigation pane, click **Connections**.
4. On the **Close** window, click the appropriate tab based on which connection you want to close.
5. Based on your selection, enter the IP address, ActiveMQ address match, or the user name, and then click **Close**.

Rolling Back Transactions for a Messaging Server

1. Access the management console and navigate to **Server** using either of the following:
   - **Runtime→ Browse By → Hosts → Host → Server**
   - **Runtime→ Browse By → Server Groups → Server Group → Server**
2. Click **Messaging ActiveMQ → Server** and then click **View**.
3. On the navigation pane, click **Transactions**.
4. Select the transaction you want to roll back and click **Rollback**.

Committing Transactions for a Messaging Server

1. Access the management console and navigate to **Server** using either of the following:
   - **Runtime→ Browse By → Hosts → Host → Server**
   - **Runtime→ Browse By → Server Groups → Server Group → Server**
2. Click **Messaging ActiveMQ → Server** and then click **View**.
3. On the navigation pane, click **Transactions**.
4. Select the transaction you want to commit and click **Commit**.

CHAPTER 34. TUNING JAKARTA MESSAGING

If you use the Jakarta Messaging API, review the following information for tips on how to improve performance.

- Disable the message ID.
  If you do not need message IDs, disable them by using the **setDisableMessageID()** method on the **MessageProducer** class. Setting the value to **true** eliminates the overhead of creating a unique ID and decreases the size of the message.

- Disable the message timestamp.
  If you do not need message timestamps, disable them by using the **setDisableMessageTimeStamp()** method on the **MessageProducer** class. Setting the value to **true** eliminates the overhead of creating the timestamp and decreases the size of the message.

- Avoid using **ObjectMessage**.
  ObjectMessage is used to send a message that contains a serialized object, meaning the body of the message, or payload, is sent over the wire as a stream of bytes. The Java serialized form of even small objects is quite large and takes up a lot of space on the wire. It is also slow when compared to custom marshalling techniques. Use **ObjectMessage** only if you cannot use one of the other message types, for example, if you do not know the type of the payload until runtime.

- Avoid **AUTO_ACKNOWLEDGE**.
  The choice of acknowledgement mode in a consumer impacts performance because of the additional overhead and traffic incurred by sending the acknowledgment message sent over the network. **AUTO_ACKNOWLEDGE** incurs this overhead because it requires an acknowledgement to be sent from the server for each message received on the client. If you can, use **DUPS_OK_ACKNOWLEDGE**, which acknowledges messages in a lazy manner, **CLIENT_ACKNOWLEDGE**, meaning the client code will call a method to acknowledge the message, or batch up many acknowledgements with one acknowledge or commit in a transacted session.

- Avoid durable messages.
  By default, Jakarta Messaging messages are durable. If you do not need durable messages, set them to be **non-durable**. Durable messages incur a lot of overhead because they are persisted to storage.

- Use **TRANSACTED_SESSION** mode to send and receive messages in a single transaction.
  By batching messages in a single transaction, the ActiveMQ Artemis server integrated in JBoss EAP requires only one network round trip on the commit, not on every send or receive.

CHAPTER 35. TUNING PERSISTENCE

- Put the message journal on its own physical volume.
  One of the advantages of an append-only journal is that disk head movement is minimized. This advantage is lost if the disk is shared. When multiple processes, such as a transaction coordinator, databases, and other journals, read and write from the same disk, performance is impacted because the disk head must skip around between different files. If you are using paging or large messages, make sure they are also put on separate volumes.

- Tune the **journal-min-files** value.
  Set the **journal-min-files** parameter to the number of files that fits your average sustainable rate. If you frequently see new files being created on the journal data directory, meaning a lot data is being persisted, you need to increase the minimal number of files. This allows the journal to reuse, rather than create, new data files.

- Optimize the journal file size.
  The journal file size must be aligned to the capacity of a cylinder on the disk. The default value of **10MB** should be enough on most systems.

- Use the **AIO** journal type.
  For Linux operating systems, keep your journal type as **AIO**. **AIO** scales better than Java **NIO**.

- Tune the **journal-buffer-timeout** value.
  Increasing the **journal-buffer-timeout** value results in increased throughput at the expense of latency.

- Tune the **journal-max-io** value.
  If you are using **AIO**, you might be able improve performance by increasing the **journal-max-io** parameter value. Do not change this value if you are using **NIO**.

CHAPTER 36. OTHER TUNING OPTIONS

This section describes other places in JBoss EAP messaging that can be tuned.

- Use asynchronous send acknowledgements.
  If you need to send non-transactional, durable messages and do not need a guarantee that they have reached the server by the time the call to **send()** returns, do not set them to be sent blocking. Instead use asynchronous send acknowledgements to get your send acknowledgements returned in a separate stream. However, in the case of a server crash, some messages might be lost.

- Use **pre-acknowledge** mode.
  With **pre-acknowledge** mode, messages are acknowledged before they are sent to the client. This reduces the amount of acknowledgment traffic on the wire. However, if that client crashes, messages will not be redelivered if the client reconnects.

- Disable security.
  There is a small performance boost when you disable security by setting the **security-enabled** attribute to false.

- Disable persistence.
  You can turn off message persistence altogether by setting **persistence-enabled** to **false**.

- Sync transactions lazily.
  Setting **journal-sync-transactional** to **false** provides better transactional persistent performance at the expense of some possibility of loss of transactions on failure.

- Sync non-transactional lazily.
  Setting **journal-sync-non-transactional** to **false** provides better non-transactional persistent performance at the expense of some possibility of loss of durable messages on failure.

- Send messages non-blocking.
  To avoid waiting for a network round trip for every message sent, set **block-on-durable-send** and **block-on-non-durable-send** to **false** if you are using Jakarta Messaging and JNDI, or set it directly on the **ServerLocator** by calling the **setBlockOnDurableSend()** and **setBlockOnNonDurableSend()** methods.

- Optimize the **consumer-window-size**.
  If you have very fast consumers, you can increase the **consumer-window-size** to effectively disable consumer flow control.

- Use the core API instead of the Jakarta Messaging API.
  Jakarta Messaging operations must be translated into core operations before the server can handle them, resulting in lower performance than when you use the core API. When using the core API, try to use methods that take **SimpleString** as much as possible. **SimpleString**, unlike **java.lang.String**, does not require copying before it is written to the wire, so if you reuse **SimpleString** instances between calls, you can avoid some unnecessary copying. Note that the core API is not portable to other brokers.

CHAPTER 37. AVOIDING ANTI-PATTERNS

- Reuse connections, sessions, consumers, and producers where possible.
  The most common messaging anti-pattern is the creation of a new connection, session, and producer for every message sent or consumed. These objects take time to create and may involve several network round trips, so it is a poor use of resources. Always reuse them.

  NOTE

  Some popular libraries such as the Spring Messaging Template use these anti-patterns. If you are using the Spring Messaging Template, you may see poor performance. The Spring Messaging Template can only safely be used in an application server which caches Jakarta Messaging sessions, for example, using Jakarta Connectors, and only then for sending messages. It cannot safely be used for synchronously consuming messages, even in an application server.

- Avoid fat messages.
  Verbose formats such as XML take up a lot of space on the wire and performance suffers as result. Avoid XML in message bodies if you can.

- Do not create temporary queues for each request.
  This common anti-pattern involves the temporary queue request-response pattern. With the temporary queue request-response pattern, a message is sent to a target, and a reply-to header is set with the address of a local temporary queue. When the recipient receives the message, they process it, and then send back a response to the address specified in the reply-to header. A common mistake made with this pattern is to create a new temporary queue on each message sent, which drastically reduces performance. Instead, the temporary queue should be reused for many requests.

- Do not use message driven beans unless it is necessary.
  Using MDBs to consume messages is slower than consuming messages using a simple Jakarta Messaging message consumer.

APPENDIX A. REFERENCE MATERIAL

A.1. ADDRESS SETTING ATTRIBUTES

Table A.1. Address Setting Attributes

Name	Description
address-full-policy	Determines what happens when an address where **max-size-bytes** is specified becomes full. Accepted values are **PAGE**, **DROP**, **FAIL** or **BLOCK**. If the value is **PAGE** then further messages will be paged to disk. If the value is **DROP** then further messages will be silently dropped. If the value is **FAIL** then the messages will be dropped and the client message producers will receive an exception. If the value is **BLOCK** then client message producers will block when they try and send further messages. **PAGE** is the default. See About Paging for details on paging.
auto-create-jms-queues	Determines whether JBoss EAP should automatically create a Jakarta Messaging queue corresponding to the address settings match when a Jakarta Messaging producer or a consumer tries to use such a queue. The default is **false**. *Deprecated: Use **auto-create-queues** instead.*
auto-create-jms-topics	Determines whether JBoss EAP should automatically create a Jakarta Messaging topic corresponding to the address settings match when a Jakarta Messaging producer or a consumer tries to use such a queue. The default is **false**. *Deprecated: Use **auto-create-addresses** instead.*
auto-create-addresses	Determines whether the broker should automatically create an address when a message is sent or a consumer tries to connect to a queue whose name fits the address **match**. Queues that are auto-created are durable, non-temporary, and non-transient. The default is **true**.
auto-create-queues	Determines whether the broker should automatically create a queue when a message is sent or a consumer tries to connect to a queue whose name fits the address **match**. Queues that are auto-created are durable, non-temporary, and non-transient. The default is **true**.
auto-delete-jms-queues	Determines whether JBoss EAP should automatically delete auto-created Jakarta Messaging queues when they have no consumers and no messages. The default is **false**. *Deprecated: Use **auto-delete-queues** instead.*
auto-delete-jms-topics	Determines whether JBoss EAP should automatically delete auto-created Jakarta Messaging topics when they have no consumers and no messages. The default is **false**. *Deprecated: Use **auto-delete-addresses** instead.*
auto-delete-addresses	Determines whether or not the broker should automatically delete auto-created addresses once the address no longer has any queues. The default is **true**.

Name	Description
auto-delete-queues	Determines whether or not the broker should automatically delete auto-created queues when they have both **0** consumers and **0** messages. The default is **true**.
dead-letter-address	The address to send dead messages to. See Configuring Dead Letter Addresses for more information.
expiry-address	The address that will receive expired messages. See Configuring Message Expiry for details.
expiry-delay	Defines the expiration time, in milliseconds, that will be used for messages using the default expiration time. Default is **-1**.
last-value-queue	Defines whether a queue only uses last values or not. See Last-value Queues for more information.
max-delivery-attempts	Defines how many time a canceled message can be redelivered before sending to the dead-letter-address. Default is **10**.
max-redelivery-delay	Maximum value for the redelivery-delay, in milliseconds. Default is **0**.
max-size-bytes	The maximum size for this address, in bytes. Default is **-1**.
message-counter-history-day-limit	Day limit for the message counter history. Default is **0**.
page-max-cache-size	The number of page files to keep in memory to optimize IO during paging navigation. Default is **5**.
page-size-bytes	The paging size, in bytes. Default is **10485760**.
redelivery-delay	Defines how long to wait before attempting redelivery of a canceled message, in milliseconds. Default is **0**. See Configuring Delayed Redelivery for more information.
redelivery-multiplier	Multiplier to apply to the redelivery-delay parameter. Default is **1.0**.
redistribution-delay	Defines how long to wait, in milliseconds, after the last consumer is closed on a queue before redistributing any messages. Default is **-1**.
send-to-dla-on-no-route	When set to **true**, a message will be sent to the configured dead letter address if it cannot be routed to any queues. Default is **false**.
slow-consumer-check-period	How often to check, in seconds, for slow consumers. Default is **5**.

Name	Description
slow-consumer-policy	Determines what happens when a slow consumer is identified. Valid options are **KILL** or **NOTIFY**. **KILL** will kill the consumer's connection, which will impact any client threads using that same connection. **NOTIFY** will send a **CONSUMER_SLOW** management notification to the client. Default is **NOTIFY**.
slow-consumer-threshold	The minimum rate of message consumption allowed before a consumer is considered slow. Default is **-1**.

A.2. CONNECTION FACTORY ATTRIBUTES

Table A.2. Connection Factory Attributes

Attribute	Description
auto-group	Whether message grouping is automatically used.
block-on-acknowledge	Whether to block on acknowledge.
block-on-durable-send	Whether to block on durable send.
block-on-non-durable-send	Whether to block on non durable send.
cache-large-message-client	Whether to cache large messages.
call-failover-timeout	The timeout, in milliseconds, to use when failover is in process.
call-timeout	The call timeout, in milliseconds.
client-failure-check-period	The client failure check period, in milliseconds.
client-id	The client ID.
compress-large-messages	Whether large messages should be compressed.
confirmation-window-size	The confirmation window size, in bytes.
connection-load-balancing-policy-class-name	Name of a class implementing a client-side load balancing policy that a client can use to load balance sessions across different nodes in a cluster.
connection-ttl	The connection time to live, in milliseconds.
connectors	Defines the connectors, which are stored in a map by connector name (with an undefined value). It is possible to pass a list of connector names when writing this attribute.

Attribute	Description
consumer-max-rate	The consumer maximum rate, per second.
consumer-window-size	The consumer window size, in bytes.
deserialization-black-list	Defines the list of classes or packages that are not allowed to be deserialized.
deserialization-white-list	Defines the list of classes or packages that are allowed to be deserialized.
discovery-group	The discovery group name.
dups-ok-batch-size	The dups ok batch size.
enable-amq1-prefix	Specifies whether to include a prefix on the Jakarta Messaging destination name. If the value is false, the prefix is not included.
entries	The JNDI names the connection factory should be bound to.
factory-type	The type of connection factory. Valid values are: • **GENERIC** • **TOPIC** • **QUEUE** • **XA_GENERIC** • **XA_QUEUE** • **XA_TOPIC** Use **GENERIC** for a general connection to a broker, while**TOPIC** and **QUEUE** should be used for connections to their respective Jakarta Messaging types. The XA counterparts should be used for transactional messaging.
failover-on-initial-connection	Whether to failover on initial connection.
group-id	The group ID.
ha	Whether the connection factory supports high availability.
max-retry-interval	The maximum retry interval, in milliseconds.
min-large-message-size	The minimum large message size, in bytes.
pre-acknowledge	Whether to pre-acknowledge.

Attribute	Description
producer-max-rate	The producer maximum rate, per second.
producer-window-size	The producer window size, in bytes.
protocol-manager-factory	The protocol manager factory used by this connection factory.
reconnect-attempts	The reconnect attempts.
retry-interval	The retry interval, in milliseconds.
retry-interval-multiplier	The retry interval multiplier.
scheduled-thread-pool-max-size	The scheduled thread pool maximum size.
thread-pool-max-size	The thread pool maximum size.
transaction-batch-size	The transaction batch size.
use-global-pools	Whether to use global pools.
use-topology-for-load-balancing	Specifies whether the messaging client will use the cluster topology to connect to a cluster, or will use the initial connector for all connections.

A.3. POOLED CONNECTION FACTORY ATTRIBUTES

Table A.3. Pooled Connection Factory Attributes

Attribute	Description
allow-local-transactions	Whether to allow local transactions for outbound Jakarta Messaging Sessions. When set to **true**, this enables a Jakarta Messaging producer to send messages in the following situations: ● From a servlet in a transacted session. ● From an MDB in a transacted session with the transaction type attribute set to **NOT_SUPPORTED**. This attribute does not apply to the **JMSContext**, which explicitly disallows it.
auto-group	Whether message grouping is automatically used.
block-on-acknowledge	Whether to block on acknowledge.

Attribute	Description
block-on-durable-send	Whether to block on durable send.
block-on-non-durable-send	Whether to block on non durable send.
cache-large-message-client	Whether to cache large messages.
call-failover-timeout	The timeout, in milliseconds, to use when failover is in process.
call-timeout	The call timeout, in milliseconds.
client-failure-check-period	The client failure check period, in milliseconds.
client-id	The client id.
compress-large-messages	Whether large messages should be compressed.
confirmation-window-size	The confirmation window size, in bytes.
connection-load-balancing-policy-class-name	Name of a class implementing a client-side load balancing policy that a client can use to load balance sessions across different nodes in a cluster.
connection-ttl	The connection time to live, in milliseconds.
connectors	Defines the connectors, which are stored in a map by connector name (with an undefined value). It is possible to pass a list of connector names when writing this attribute.
consumer-max-rate	The consumer max rate, per second.
consumer-window-size	The consumer window size, in bytes.
credential-reference	Credential, from a credential store, to authenticate the pooled connection factory.
deserialization-black-list	Defines the list of classes or packages that are not allowed to be deserialized.
deserialization-white-list	Defines the list of classes or packages that are allowed to be deserialized.
discovery-group	The discovery group name.
dups-ok-batch-size	The dups ok batch size.

Attribute	Description
enable-amq1-prefix	Specifies whether to include a prefix on the Jakarta Messaging destination name. If the value is false, the prefix is not included.
enlistment-trace	Enables IronJacamar to record enlistment traces for this pooled connection factory. This attribute is undefined by default and the behavior is driven by the presence of the **ironjacamar.disable_enlistment_trace** system property.
entries	The JNDI names that the connection factory should be bound to.
failover-on-initial-connection	Whether to failover on initial connection.
group-id	The group id.
ha	Whether the connection factory supports high availability.
initial-connect-attempts	The number of attempts to connect initially with this factory.
initial-message-packet-size	The initial size of messages created through this factory.
jndi-params	The JNDI params to use for locating the destination for incoming connections.
managed-connection-pool	The class name of the managed connection pool used by this pooled connection factory.
max-pool-size	The maximum size for the pool.
max-retry-interval	The max retry interval, in milliseconds.
min-large-message-size	The min large message size, in bytes.
min-pool-size	The minimum size for the pool.
password	The default password to use with this connection factory. This is only needed when pointing the connection factory to a remote host.
pre-acknowledge	Whether to pre-acknowledge.
producer-max-rate	The producer max rate, per second.
producer-window-size	The producer window size, in bytes.
protocol-manager-factory	The protocol manager factory used by this pooled connection factory.

Attribute	Description
reconnect-attempts	The reconnect attempts. By default, a pooled connection factory will try to reconnect infinitely to the messaging servers.
retry-interval	The retry interval, in milliseconds.
retry-interval-multiplier	The retry interval multiplier.
scheduled-thread-pool-max-size	The scheduled thread pool max size.
setup-attempts	The number of times to set up an MDB endpoint.
setup-interval	The interval, in milliseconds, between attempts at setting up an MDB endpoint.
statistics-enabled	Whether runtime statistics are enabled.
thread-pool-max-size	The thread pool max size.
transaction	The transaction mode.
transaction-batch-size	The transaction batch size.
use-auto-recovery	Whether to use auto recovery.
use-global-pools	Whether to use global pools.
use-jndi	Use JNDI to locate the destination for incoming connections.
use-local-tx	Use a local transaction for incoming sessions.
use-topology-for-load-balancing	Specifies whether the messaging client will use the cluster topology to connect to a cluster, or will use the initial connector for all connections.
user	The default username to use with this connection factory. This is only needed when pointing the connection factory to a remote host.

A.4. CORE BRIDGE ATTRIBUTES

Table A.4. Core Bridge Attributes

Attribute	Description

Attribute	Description
call-timeout	The period of time, in milliseconds, to wait for a response for blocking calls that are made by the core bridge. The core bridge uses blocking calls when duplicate detection is not configured. The default value is **30000**, or 30 seconds.
check-period	The period, in milliseconds, between client failure checks.
confirmation-window-size	The size to use for the connection used to forward messages to the target node.
connection-ttl	The maximum time, in milliseconds, for which the connections used by the bridges are considered alive, in the absence of heartbeat.
credential-reference	Credential, from a credential store, to authenticate the bridge.
discovery-group	The name of the discovery group used by this bridge. This attribute may not be set if **static-connectors** is defined.
filter	An optional filter string. If specified, then only messages that match the filter expression specified will be forwarded.
forwarding-address	The address on the target server that the message will be forwarded to. If a forwarding address is not specified, then the original destination of the message will be retained.
ha	Whether or not this bridge should support high availability. If **true**, then it will connect to any available server in a cluster and support failover. The default value is **false**.
initial-connect-attempts	The number of attempts to connect initially with this bridge.
max-retry-interval	The maximum interval of time used to retry connections.
min-large-message-size	The minimum size, in bytes, for a message before it is considered as a large message.
password	The password to use when creating the bridge connection to the remote server. If it is not specified, then the default cluster password specified by the **cluster-password** attribute in the **messaging-activemq** subsystem resource will be used.
cluster-credential-reference	Credential store reference used to authenticate to the cluster. This may be used instead of **password**.
queue-name	The unique name of the local queue that the bridge consumes from. The queue must already exist by the time the bridge is instantiated at startup.

Attribute	Description
reconnect-attempts	The total number of reconnect attempts that the bridge will make before giving up and shutting down. The default value is **-1**, which signifies an unlimited number of attempts.
reconnect-attempts-on-same-node	The total number of reconnect attempts on the same node that the bridge will make before giving up and shutting down. A value of **-1** signifies an unlimited number of attempts. The default is **10**.
retry-interval	The period, in milliseconds, between subsequent reconnection attempts, if the connection to the target server has failed.
retry-interval-multiplier	A multiplier to apply to the time since the last retry to compute the time to the next retry. This allows you to implement an exponential backoff between retry attempts.
static-connectors	A list of statically defined connectors used by this bridge. This attribute may not be set if **discovery-group** is defined.
transformer-class-name	The name of a user-defined class that implements the **org.apache.activemq.artemis.core.server.cluster.Transformer** interface.
use-duplicate-detection	Whether the bridge will automatically insert a duplicate ID property into each message that it forwards.
user	The user name to use when creating the bridge connection to the remote server. If not specified, the default cluster user specified by the **cluster-user** attribute in the **messaging-activemq** subsystem resource will be used.

A.5. JAKARTA MESSAGING BRIDGE ATTRIBUTES

Table A.5. Jakarta Messaging bridge Attributes

Attribute	Description
add-messageID-in-header	If this is set to **true**, then the original message's message ID will be appended in the message sent to the destination in the header **AMQ_BRIDGE_MSG_ID_LIST**. If the message is bridged more than once, each message ID will be appended.
client-id	The Jakarta Messaging client ID to use when creating and looking up the subscription if it is durable and the source destination is a topic.
failure-retry-interval	The amount of time, in milliseconds, to wait between trying to recreate connections to the source or target servers when the bridge has detected they have failed.

Attribute	Description
max-batch-size	The maximum number of messages to consume from the source destination before sending them in a batch to the target destination. The value must greater than or equal to **1**.
max-batch-time	The maximum number of milliseconds to wait before sending a batch of messages to a target, even if the number of messages consumed has not reached max-batch-size. A value of **-1** means to wait forever.
max-retries	The number of times to attempt to recreate connections to the source or target servers when the bridge has detected they have failed. The bridge will give up after trying this number of times. A value of **-1** means to try forever.
module	The name of the JBoss EAP module containing the resources required to look up source and target Jakarta Messaging resources.
paused	A read-only property that reports whether the Jakarta Messaging bridge is paused.
quality-of-service	The desired quality of service mode. Possible values are **AT_MOST_ONCE**, **DUPLICATES_OK**, or **ONCE_AND_ONLY_ONCE**. See Quality of Service for details on the different modes.
selector	A Jakarta Messaging selector expression used for consuming messages from the source destination. Only messages that match the selector expression will be bridged from the source to the target destination.
subscription-name	The name of the subscription if it is durable and the source destination is a topic.
source-connection-factory	The name of the source connection factory to look up on the source messaging server.
source-context	The properties used to configure the source JNDI initial context.
source-credential-reference	Credential store reference used to authenticate the source connection. This may be used instead of **source-password**.
source-destination	The name of the source destination to look up on the source messaging server.
source-password	The password for creating the source connection.
source-user	The name of the user for creating the source connection.

Attribute	Description
target-connection-factory	The name of the target connection factory to look up on the target messaging server.
target-context	The properties used to configure the target JNDI initial context.
target-credential-reference	Credential store reference used to authenticate the target connection. This may be used instead of **target-password**.
target-destination	The name of the target destination to look up on the target messaging server.
target-password	The password for creating the target connection.
target-user	The name of the user for creating the target connection.

A.6. CLUSTER CONNECTION ATTRIBUTES

Table A.6. Cluster Connection Attributes

Attribute	Description
allow-direct-connections-only	If set to **true**, this node will **not** create a connection to another node in the cluster if it resides more than 1 hop away. Used only when the attribute **static-connectors** is defined. The default is**false**.
call-failover-timeout	The timeout, in milliseconds, to use when failover is in process for remote calls made by the cluster connection. The default is **-1**, which is unbounded.
call-timeout	The timeout, in milliseconds, for remote calls made by the cluster connection. The default is **30000**, or 30 seconds.
check-period	The period, in milliseconds, between client failure check. The default is **30000**, or 30 seconds.
cluster-connection-address	Each cluster connection only applies to messages sent to an address that starts with this value.
confirmation-window-size	The window size, in bytes, for the connection used to forward messages to a target node. The default is **1048576**.
connection-ttl	The maximum time, in milliseconds, for which the connections used by the cluster connections are considered alive in the absence of heartbeat. The default is **60000**, or 60 seconds.
connector-name	The name of the connector to use for the cluster connection.

Attribute	Description
discovery-group	The discovery group used to obtain the list of other servers in the cluster to which this cluster connection will make connections. Must be undefined (null) if **static-connectors** is defined.
initial-connect-attempts	The number of attempts to connect initially with this cluster connection. The default is **-1**, which is unbounded.
max-hops	The maximum number of times a message can be forwarded. JBoss EAP can be configured to also load balance messages to nodes that might be connected to it only indirectly with other ActiveMQ Artemis messaging servers as intermediates in a chain. The default is **1**.
max-retry-interval	The maximum interval of time, in milliseconds, used to retry connections. The default is **2000**, or two seconds.
message-load-balancing-type	This parameter determines how messages will be distributed between other nodes in the cluster. Replaces the deprecated **forward-when-no-consumers**. Valid values are**OFF**, **STRICT**, or **ON_DEMAND**. **OFF** Messages will never be forwarded to another node in the cluster. **STRICT** Messages will be distributed in a round robin fashion even though the same queues on the other nodes of the cluster may have no consumers at all, or they may have consumers that have nonmatching message filters or selectors. Note that JBoss EAP will not forward messages to other nodes if there are no queues of the same name on the other nodes, even if this parameter is set to **STRICT**. Using **STRICT** is like setting the legacy **forward-when-no-consumers** parameter to **true**. **ON_DEMAND** Messages are forwarded to other nodes of the cluster if the forwarding address has queues that have consumers. If those consumers have message filters or selectors, at least one of those selectors must match the message. Using **ON_DEMAND** is like setting the legacy **forward-when-no-consumers** parameter to **false**. The default is **ON_DEMAND**.
min-large-message-size	The minimum size, in bytes, for a message before it is considered as a large message. The default is **102400**.
node-id	The node ID used by this cluster connection. This attribute is read only.
notification-attempts	How many times the cluster connection will broadcast itself. The default is **2**.
notification-interval	The interval, in milliseconds, between notifications. The default is **10000**, or 10 seconds.

Attribute	Description
reconnect-attempts	The total number of reconnect attempts the bridge will make before giving up and shutting down. The default is **-1**, which signifies an unlimited number of attempts.
retry-interval	The period, in milliseconds, between subsequent attempts to reconnect to a target server, if the connection to the target server has failed. The default is **500**.
retry-interval-multiplier	A multiplier to apply to the time since the last retry to compute the time to the next retry. This allows you to implement an exponential backoff between retry attempts. The default is **1.0**.
static-connectors	The statically defined list of connectors to which this cluster connection will make connections. Must be undefined if **discovery-group-name** is defined.
topology	The topology of the nodes that this cluster connection is aware of. This attribute is read only.
use-duplicate-detection	Whether the bridge will automatically insert a duplicate ID property into each message that it forwards. The default is **true**.

A.7. MESSAGING STATISTICS

Queue Statistics

Table A.7. Queue Statistics

Statistic	Description
Consumer Count	The number of consumers consuming messages from this queue.
Message Count	The number of messages currently in this queue.
Messages Added Count	The number of messages added to this queue since it was created.
Scheduled Count	The number of scheduled messages in this queue.

Topic Statistics

Table A.8. Topic Statistics

Statistic	Description
Delivering Count	The number of messages that this topic is currently delivering to its consumers.

Statistic	Description
Durable Message Count	The number of messages for all durable subscribers for this topic.
Durable Subscription Count	The number of durable subscribers for this topic.
Message Count	The number of messages currently in this topic.
Messages Added	The number of messages added to this topic since it was created.
Subscription Count	The number of durable and non-durable subscribers for this topic.

Pooled Connection Factory Statistics

NOTE

Statistics collection for pooled connection factories are enabled separately from the other statistics collected for a messaging server. Use the following management CLI command to enable statistics collection for a pooled connection factory.

```
/subsystem=messaging-activemq/server=default/pooled-connection-factory=activemq-ra:write-attribute(name=statistics-enabled,value=true)
```

Table A.9. Pooled Connection Factory Statistics

Statistic	Description
ActiveCount	The active count.
AvailableCount	The available count.
AverageBlockingTime	Average blocking time for pool.
AverageCreationTime	The average time spent creating a physical connection.
AverageGetTime	The average time spent obtaining a physical connection.
AveragePoolTime	The average time spent by physical connections in the pool.
AverageUsageTime	The average time spent using a physical connection.
BlockingFailureCount	The number of failures trying to obtain a physical connection.
CreatedCount	The created count.
DestroyedCount	The destroyed count.

Statistic	Description
IdleCount	The number of physical connections currently idle.
InUseCount	The number of physical connections currently in use.
MaxCreationTime	The maximum time for creating a physical connection.
MaxGetTime	The maximum time for obtaining a physical connection.
MaxPoolTime	The maximum time for a physical connection in the pool.
MaxUsageTime	The maximum time using a physical connection.
MaxUsedCount	The maximum number of connections used.
MaxWaitCount	The maximum number of threads waiting for a connection.
MaxWaitTime	The maximum wait time for a connection.
TimedOut	The timed out count.
TotalBlockingTime	The total blocking time.
TotalCreationTime	The total time spent creating physical connections.
TotalGetTime	The total time spent obtaining physical connections.
TotalPoolTime	The total time spent by physical connections in the pool.
TotalUsageTime	The total time spent using physical connections.
WaitCount	The number of requests that had to wait to obtain a physical connection.
XACommitAverageTime	The average time for an XAResource commit invocation.
XACommitCount	The number of XAResource commit invocations.
XACommitMaxTime	The maximum time for an XAResource commit invocation.
XACommitTotalTime	The total time for all XAResource commit invocations.
XAEndAverageTime	The average time for an XAResource end invocation.
XAEndCount	The number of XAResource end invocations.

Statistic	Description
XAEndMaxTime	The maximum time for an XAResource end invocation.
XAEndTotalTime	The total time for all XAResource end invocations.
XAForgetAverageTime	The average time for an XAResource forget invocation.
XAForgetCount	The number of XAResource forget invocations.
XAForgetMaxTime	The maximum time for an XAResource forget invocation.
XAForgetTotalTime	The total time for all XAResource forget invocations.
XAPrepareAverageTime	The average time for an XAResource prepare invocation.
XAPrepareCount	The number of XAResource prepare invocations.
XAPrepareMaxTime	The maximum time for an XAResource prepare invocation.
XAPrepareTotalTime	The total time for all XAResource prepare invocations.
XARecoverAverageTime	The average time for an XAResource recover invocation.
XARecoverCount	The number of XAResource recover invocations.
XARecoverMaxTime	The maximum time for an XAResource recover invocation.
XARecoverTotalTime	The total time for all XAResource recover invocations.
XARollbackAverageTime	The average time for an XAResource rollback invocation.
XARollbackCount	The number of XAResource rollback invocations.
XARollbackMaxTime	The maximum time for an XAResource rollback invocation.
XARollbackTotalTime	The total time for all XAResource rollback invocations.
XAStartAverageTime	The average time for an XAResource start invocation.
XAStartCount	The number of XAResource start invocations.
XAStartMaxTime	The maximum time for an XAResource start invocation.
XAStartTotalTime	The total time for all XAResource start invocations.

Revised on 2024-01-17 05:24:52 UTC