



Red Hat JBoss A-MQ 6.3

Configuring Broker Persistence

Red Hat JBoss A-MQ's persistence layer can be tailored for speed and robustness

Red Hat JBoss A-MQ 6.3 Configuring Broker Persistence

Red Hat JBoss A-MQ's persistence layer can be tailored for speed and robustness

JBoss A-MQ Docs Team

Content Services

fuse-docs-support@redhat.com

Legal Notice

Copyright © 2016 Red Hat.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide discusses how to configure Red Hat JBoss A-MQ's persistence layer to best suite your application and your environment.

Table of Contents

CHAPTER 1. INTRODUCTION TO RED HAT JBOSS A-MQ PERSISTENCE	4
OVERVIEW	4
PERSISTENT MESSAGE STORES	4
MESSAGE CURSORS	4
ACTIVATING AND DEACTIVATING PERSISTENCE	4
CONFIGURING PERSISTENCE ADAPTER BEHAVIOR	5
CUSTOMIZING THE STORE'S LOCKER	5
CHAPTER 2. USING THE KAHADB MESSAGE STORE	6
2.1. UNDERSTANDING THE KAHADB MESSAGE STORE	6
2.2. CONFIGURING THE KAHADB MESSAGE STORE	8
2.3. CONCURRENT STORE AND DISPATCH	13
2.4. OPTIMIZING THE METADATA CACHE	17
2.5. CREATING AN OFFLINE BACKUP (OPENSIFT)	18
2.6. RECOVERY	19
CHAPTER 3. USING A MULTI KAHADB PERSISTENCE ADAPTER	22
OVERVIEW	22
CONFIGURATION	22
WILDCARD SYNTAX	23
EXAMPLE	23
AUTOMATIC PER-DESTINATION PERSISTENCE ADAPTER	24
TRANSACTIONS	24
CHAPTER 4. USING THE LEVELDB PERSISTENCE ADAPTER	25
OVERVIEW	25
PLATFORM SUPPORT	25
BASIC CONFIGURATION	25
CONFIGURATION ATTRIBUTES	26
CHAPTER 5. USING THE REPLICATED LEVELDB PERSISTENCE ADAPTER	29
OVERVIEW	29
DEPLOYMENT TIPS	29
BASIC CONFIGURATION	30
CONFIGURATION ATTRIBUTES	31
CHAPTER 6. USING JDBC TO CONNECT TO A DATABASE STORE	34
6.1. BASICS OF USING THE JDBC PERSISTENCE ADAPTER	34
6.2. USING THE PLAIN JDBC ADAPTER	37
6.3. USING JDBC WITH THE HIGH PERFORMANCE JOURNAL	38
6.4. CUSTOMIZING THE JDBC PERSISTENCE ADAPTER	41
6.5. TUTORIAL: JDBC PERSISTENCE	45
6.6. TUTORIAL: CONFIGURING ACTIVEMQ JDBC PERSISTENCE ON FABRIC CONTAINER WITH POSTGRESQL	48
CHAPTER 7. MESSAGE CURSORS	55
7.1. TYPES OF CURSORS	55
7.2. CONFIGURING THE TYPE OF CURSOR USED BY A DESTINATION	58
CHAPTER 8. MESSAGE STORE LOCKERS	62
8.1. LOCKER BASICS	62
8.2. USING THE PROVIDED LOCKERS	63
8.3. USING CUSTOM LOCKERS	66

INDEX 68

CHAPTER 1. INTRODUCTION TO RED HAT JBOSS A-MQ PERSISTENCE

Abstract

Message persistence allows for the recovery of undelivered messages in the event of a system failure. By default, Red Hat JBoss A-MQ's persistence features are activated. The default set-up is fast and scalable. It is easy to customize the broker configuration to use a JDBC compliant database.

OVERVIEW

Loss of messages is not acceptable in mission critical applications. Red Hat JBoss A-MQ reduces the risk of message loss by using a persistent message store by default. Persistent messages are written to the persistent store when they are sent. The messages persist in the store until their delivery is confirmed. This means that, in the case of a system failure, JBoss A-MQ can recover all of the undelivered messages at the time of the failure.

PERSISTENT MESSAGE STORES

The default message store is embeddable and transactional. It is both very fast and extremely reliable. JBoss A-MQ implements several different kinds of message store, including:

- KahaDB message store
- distributed KahaDB message store
- LevelDB message store
- JDBC adapter
- Journalized JDBC adapter (*deprecated*)

MESSAGE CURSORS

JBoss A-MQ caches message using *message cursors*. A message cursor represents a batch of messages cached in memory. When necessary, a message cursor can be used to retrieve the batch of persisted messages through the persistence adapter. See [Chapter 7, Message Cursors](#) for details.

ACTIVATING AND DEACTIVATING PERSISTENCE

By default, brokers are configured to use a persistence layer to ensure that persistent messages will survive a broker failure and meet the once-and-only-once requirement of the JMS specification. Having a broker's persistence layer configured comes with a cost in terms of resources used and speed, so for testing purposes or cases where persistence will never be required, it may make sense to disable a broker's persistence layer.

Deactivating a broker's persistence layer means that a broker will treat *all* messages as non-persistent. If a producer sets a message's `JMSDeliveryMode` property to **PERSISTENT** the broker will not respect the setting. The message will be delivered at-most-once instead of once-and-only-once. This means that persistent messages will *not* survive broker shutdown.

Persistence in JBoss A-MQ is controlled by a broker's XML configuration file. To change a broker's persistence behavior you modify the configuration's **broker** element's **persistent** attribute.

Table 1.1. Setting a Broker's Persistence

Value	Description
true	The broker will use a persistent message store and respect the value of a message's JMSDeliveryMode setting.
false	The broker will not use a persistent message store and will treat all messages as non-persistent regardless of the value of a message's JMSDeliveryMode setting.

[Example 1.1, "Turning Off a Broker's Persistence"](#) shows a configuration snippet for turning off a broker's message persistence.

Example 1.1. Turning Off a Broker's Persistence

```
<broker persistent="false" ... >
...
</broker>
```

CONFIGURING PERSISTENCE ADAPTER BEHAVIOR

JBoss A-MQ offers a number of different persistence mechanisms besides the default message store. To use one of the alternative message stores, or to modify the behavior of the default message store, you need to configure the persistence adapter. This is done by adding a **persistenceAdapter** element or a **persistenceFactory** element (depending on the kind of adapter you want to use) to the broker's configuration file.

CUSTOMIZING THE STORE'S LOCKER

For added flexibility in master/slave deployments JBoss A-MQ's message stores have configurable lockers. All of the message stores have a default locker implementation. The default implementation can be replaced by a custom implementation.

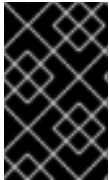
Regardless of the implementation, the locker has two configurable properties:

- if the broker should fail if the store is locked
- how long a broker waits before trying to reacquire a lock

CHAPTER 2. USING THE KAHADB MESSAGE STORE

Abstract

The KahaDB Message Store is the default message store used by Red Hat JBoss A-MQ. It is a light-weight transactional store that is fast and reliable. It uses a hybrid system that couples a transactional journal for message storage and a reference store for quick retrieval.



IMPORTANT

If you use antivirus software it can interfere with Red Hat JBoss A-MQ's ability to access the files in the KahaDB message store. You should configure your antivirus software to skip the KahaDB data folders when doing automatic scans.

2.1. UNDERSTANDING THE KAHADB MESSAGE STORE

Overview

The KahaDB message store is the default persistence store used by Red Hat JBoss A-MQ. It is a file-based persistence adapter that is optimized for maximum performance. The main features of KahaDB are:

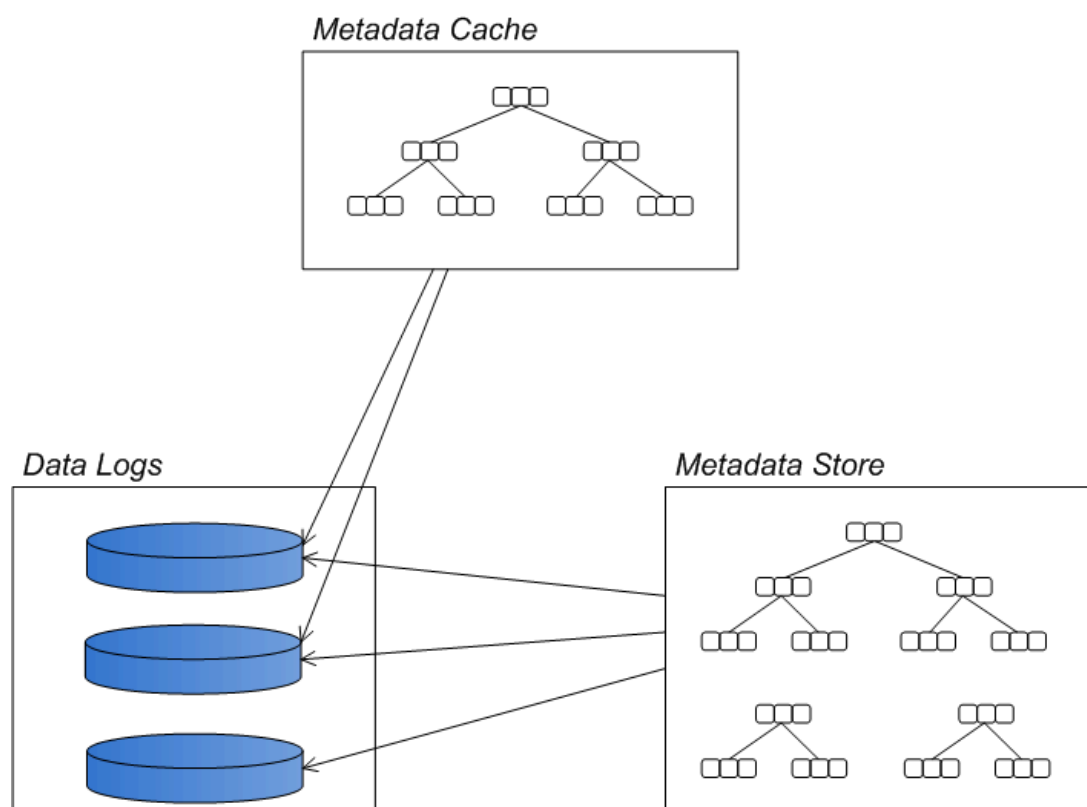
- journal-based storage so that messages can be rapidly written to disk
- allows for the broker to restart quickly
- storing message references in a B-tree index which can be rapidly updated at run time
- full support for JMS transactions
- various strategies to enable recovery after a disorderly shutdown of the broker

Architecture

The KahaDB message store is an embeddable, transactional message store that is fast and reliable. It is an evolution of the AMQ message store used by Apache ActiveMQ 5.0 to 5.3. It uses a transactional journal to store message data and a B-tree index to store message locations for quick retrieval.

[Figure 2.1, "Overview of the KahaDB Message Store"](#) shows a high-level view of the KahaDB message store.

Figure 2.1. Overview of the KahaDB Message Store



Messages are stored in file-based data logs. When all of the messages in a data log have been successfully consumed, the data log is marked as deletable. At a predetermined clean-up interval, logs marked as deletable are either removed from the system or moved to an archive.

An index of message locations is cached in memory to facilitate quick retrieval of message data. At configurable checkpoint intervals, the references are inserted into the metadata store.

Data logs

The data logs are used to store data in the form of *journals*, where events of all kinds—messages, acknowledgments, subscriptions, subscription cancellations, transaction boundaries, etc.—are stored in a rolling log. Because new events are always appended to the end of the log, a data log file can be updated extremely rapidly.

Implicitly, the data logs contain all of the message data and all of the information about destinations, subscriptions, transactions, etc.. This data, however, is stored in an arbitrary manner. In order to facilitate rapid access to the content of the logs, the message store constructs metadata to reference the data embedded in the logs.

Metadata cache

The metadata cache is an in-memory cache consisting mainly of destinations and message references. That is, for each JMS destination, the metadata cache holds a tree of message references, giving the location of every message in the data log files. Each message reference maps a message ID to a particular offset in one of the data log files. The tree of message references is maintained using a B-tree algorithm, which enables rapid searching, insertion, and deletion operations on an ordered list of messages.

The metadata cache is periodically written to the *metadata store* on the file system. This procedure is known as *check pointing* and the length of time between checkpoints is configurable using the **checkpointInterval** configuration attribute. For details on how to configure the metadata cache, see

[Section 2.4, "Optimizing the Metadata Cache"](#) .

Metadata store

The metadata store contains the complete broker metadata, consisting mainly of a B-tree index giving the message locations in the data logs. The metadata store is written to a file called **db.data**, which is periodically updated from the metadata cache.

The metadata store duplicates data that is already stored in the data logs (in a raw, unordered form). The presence of the metadata store, however, enables the broker instance to restart rapidly. If the metadata store got damaged or was accidentally deleted, the broker could recover by reading the data logs, but the restart would then take a considerable length of time.

2.2. CONFIGURING THE KAHADB MESSAGE STORE

Overview

Red Hat JBoss A-MQ's default configuration includes a persistence adapter that uses a KahaDB message store. The default configuration is suitable for many use cases, but you will likely want to update it for individual broker instances. You do this using the attributes of the **kahaDB** element.

The basic configuration tells the broker where to write the data files used by the store.

The KahaDB message store also has a number of advanced configuration attributes that customize its behavior.

Basic configuration

The KahaDB message store is configured by placing a **kahaDB** element in the **persistenceAdapter** element of your broker's configuration. The **kahaDB** element's attributes are used to configure the message store.

The attributes, listed in [Table 2.1, "Configuration Properties of the KahaDB Message Store"](#) , all have reasonable default values, so you are not required to specify values for them. However, you will want to explicitly specify the location of the message store's data files by providing a value for the **directory** attribute. This will ensure that the broker will not conflict with other brokers.

[Example 2.1, "Configuring the KahaDB Message Store"](#) shows a basic configuration of the KahaDB message store. The KahaDB files are stored under the **activemq-data** directory.

Example 2.1. Configuring the KahaDB Message Store

```
<broker brokerName="broker" persistent="true" ... >
...
<persistenceAdapter>
  <kahaDB directory="activemq-data" />
</persistenceAdapter>
...
</broker>
```

Configuration attributes

Table 2.1, “Configuration Properties of the KahaDB Message Store” describes the attributes that can be used to configure the KahaDB message store.

Table 2.1. Configuration Properties of the KahaDB Message Store

Attribute	Default Value	Description
archiveCorruptedIndex	false	Specifies if corrupted indexes are archived when the broker starts up.
archiveDataLogs	false	Specifies if the message store moves spent data logs to the archive directory.
checkForCorruptJournalFiles	false	Specifies whether the message store checks for corrupted journal files on startup and tries to recover them.
checkpointInterval	5000	Specifies the time interval, in milliseconds, between writing the metadata cache to disk.
checksumJournalFiles	true	Specifies whether the message store generates a checksum for the journal files. If you want to be able to check for corrupted journals, you must set this property to true .
cleanupInterval	30000	Specifies the time interval, in milliseconds, between cleaning up data logs that are no longer used.
compactAcksAfterNoGC	10	When the acknowledgement compaction feature is enabled, specifies how many store GC cycles must be completed without cleaning up other files before the compaction logic is triggered to possibly compact older acknowledgements spread across journal files into a new log file. The lower the value set the faster the compaction may occur which can impact performance if it runs often.

Attribute	Default Value	Description
compactAcksIgnoresStoreGrowth	false	When the acknowledgement compaction feature is enabled, specifies whether compaction is run when the store is still growing or if it should only occur when the store has stopped growing (either due to idle or store limits reached). If enabled the compaction runs regardless of the store still having room or being active which can decrease overall performance but reclaim space faster.
concurrentStoreAndDispatchQueues	true	Specifies if the message store dispatches queue messages to clients concurrently with message storage. See Section 2.3, "Concurrent Store and Dispatch" .
concurrentStoreAndDispatchTopics	false	Specifies if the message store dispatches topic messages to interested clients concurrently with message storage. See Section 2.3, "Concurrent Store and Dispatch" . Enabling this property is not recommended. It is disabled by default because it can lead to stuck or duplicate messages and its real value is small when there are multiple durable consumers, it is unlikely that all of them will have acked before the store.
databaseLockedWaitDelay	10000	Specifies the time delay, in milliseconds, before trying to acquire the database lock in the context of a shared master/slave failover deployment. See section "Shared File System Master/Slave" in "Fault Tolerant Messaging" .
directory	activemq-data	Specifies the path to the top-level folder that holds the message store's data files.
directoryArchive	null	Specifies the location of the directory to archive data logs.

Attribute	Default Value	Description
enableAckCompaction	false	Specifies whether the store will perform periodic compaction of older journal log files that contain only Message acknowledgements. After compacting the older acknowledgements into new journal log files, the older files can be removed. This action frees the space and allows the message store to continue to operate without hitting store size limits.
enableIndexDiskSyncs	true	Ensures index updates are persisted to the disk on each write.
enableIndexPageCaching	true	Enables faster reads by caching the written pages in the memory.
enableIndexRecoveryFile	true	Enabled double write of index updates, allowing the index to recover from partial write failures.
enableIndexWriteAsync	false	Specifies if kahaDB will asynchronously write indexes.
enableJournalDiskSyncs	true	Specifies whether every non-transactional journal write is followed by a disk sync. If you want to satisfy the JMS durability requirement, you must also disable concurrent store and dispatch.
failoverProducersAuditDepth		Set the audit window depth for duplicate suppression (should exceed the max transaction batch).
forceRecoverIndex	false	This option is only used in specific upgrade scenarios. In normal scenarios, this can be ignored.
ignoreMissingJournalfiles	false	Specifies whether the message store ignores any missing journal files while it starts up. If false , the message store raises an exception when it discovers a missing journal file.

Attribute	Default Value	Description
indexCacheSize	10000	Specifies the number of B-tree index pages cached in memory.
indexDirectory		Set this attribute, if you need to store the index file (metadata store), db.data , in a <i>different</i> location from the other message store data files. By default, the index is stored in the same location as the other message store files.
indexWriteBatchSize	1000	Specifies the number of B-tree indexes written in a batch. Whenever the number of changed indexes exceeds this value, the metadata cache is written to disk.
journalMaxFileLength	32mb	Specifies the maximum size of the data log files.
journalMaxWriteBatchSize	4k	Specifies the amount of data to buffer between journal disk writes. Optimize to match the fastest write size for your disk.
lockKeepAlivePeriod		In the context of a master/slave broker cluster, instructs the master to check, at intervals of the specified milliseconds, whether it still holds the lock (lock is valid) and that the lock file still exists.
locker		
maxAsyncJobs	10000	Specifies the size of the task queue used to buffer the broker commands waiting to be written to the journal. The value should be greater than or equal to the number of concurrent message producers. See Section 2.3, "Concurrent Store and Dispatch" .
maxFailoverProducersToTrack		Set the maximum number of producers (LRU cache) to track for duplicate sends.

Attribute	Default Value	Description
preallocationStrategy	sparse_file	Specifies how the broker will try to preallocate the journal files when a new journal file is needed. The default allocation strategy sets the file length, but does not populate it with any data. The os_kernel_copy strategy delegates the preallocation to the Operating System. The zeros strategy configures ActiveMQ to do the preallocation by writing 0x00 to all of the positions in the journal file.
preallocationScope	entire_journal	Specifies how the broker will preallocate the journal data files. The default preallocates on first use in the appender thread. entire_journal_async will use preallocate ahead of time in a separate thread. none disables preallocation. On SSD, using entire_journal_async avoids delaying writes pending preallocation on first use. On HDD the additional thread contention for disk has a negative impact, hence it is advisable to use the default.
storeOpenWireVersion	11	Determines the version of OpenWire commands that are marshalled to the KahaDB journal.
useIndexLFRUEviction		
useLock	true	Specifies whether the adapter uses file locking.

2.3. CONCURRENT STORE AND DISPATCH

Abstract

Concurrent store and dispatch is a strategy that facilitates high rates of message throughput, provided the consumers are able to keep up with the flow of messages from the broker.

Overview

Concurrent store and dispatch is a strategy that facilitates high rates of message throughput, provided

the consumers are able to keep up with the flow of messages from the broker. By allowing the storing of messages to proceed concurrently with the dispatch of those messages to consumers, it can happen that the consumers return acknowledgments before the messages are ever written to disk. In this case, the message writes can be optimized away, because the dispatch has already completed.

Enabling concurrent store and dispatch

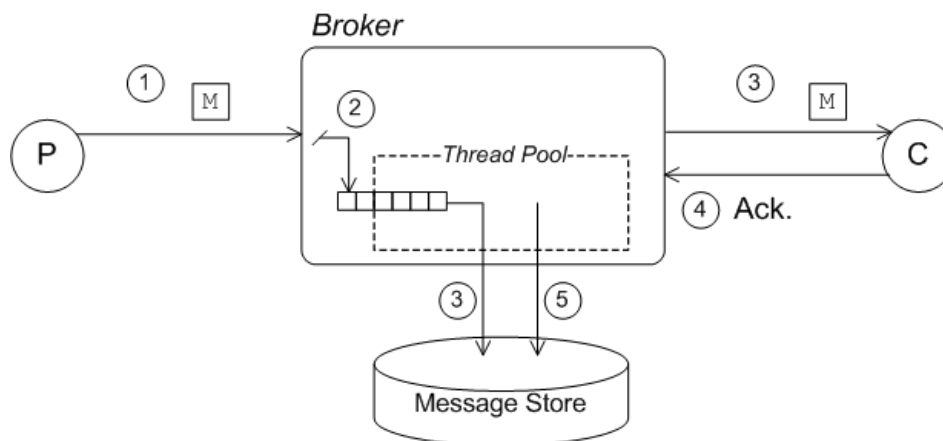
Concurrent store and dispatch is enabled by default for queues.

If you want to enable concurrent store and dispatch for topics, you must set the **kahaDB** element's **concurrentStoreAndDispatchTopics** attribute to **true**.

Concurrent with slow consumers

Figure 2.2, “Concurrent Store and Dispatch—Slow Consumers” shows an outline what happens in the broker when concurrent store and dispatch is enabled and the attached consumers are relatively *slow* to acknowledge messages.

Figure 2.2. Concurrent Store and Dispatch—Slow Consumers



In the *slow consumer* case, concurrent store and dispatch behaves as follows:

1. The producer sends a message, **M**, to a destination on the broker.
2. The broker sends the message, **M**, to the persistence layer. Because concurrency is enabled, the message is initially held in a task queue, which is serviced by a pool of threads that are responsible for writing to the journal.
3. Storing and dispatching are now performed concurrently. The message is dispatched either to one consumer (queue destination) or possibly to multiple destinations (topic consumer). In the meantime, because the attached consumers are slow, we can be sure that the thread pool has already pulled the message off the task queue and written it to the journal.
4. The consumer(s) acknowledge receipt of the message.
5. The broker asks the persistence layer to remove the message from persistent storage, because delivery is now complete.

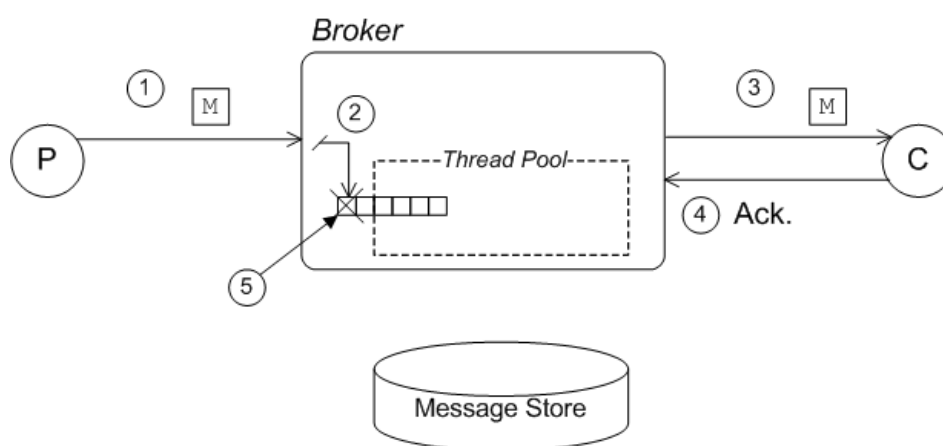
**NOTE**

In practice, because the KahaDB persistence layer is *not* able to remove the message from the rolling log files, KahaDB simply logs the fact that delivery of this message is complete. (At some point in the future, when all of the messages in the log file are marked as complete, the entire log file will be deleted.)

Concurrent with fast consumers

Figure 2.3, “Concurrent Store and Dispatch—Fast Consumers” shows an outline what happens in the broker when concurrent store and dispatch is enabled and the attached consumers are relatively *fast* at acknowledging messages.

Figure 2.3. Concurrent Store and Dispatch—Fast Consumers



In the *fast consumer* case, concurrent store and dispatch behaves as follows:

1. The producer sends a message, **M**, to a destination on the broker.
2. The broker sends the message, **M**, to the persistence layer. Because concurrency is enabled, the message is initially held in a queue, which is serviced by a pool of threads.
3. Storing and dispatching are now performed concurrently. The message is dispatched to one or more consumers.

In the meantime, assuming that the broker is fairly heavily loaded, it is probable that the message has not yet been written to the journal.

4. Because the consumers are fast, they rapidly acknowledge receipt of the message.
5. When all of the consumer acknowledgments are received, the broker asks the persistence layer to remove the message from persistent storage. But in the current example, the message is still pending and *has not been written to the journal*. The persistence layer can therefore remove the message just by deleting it from the in-memory task queue.

Disabling concurrent store and dispatch

If you want to configure the KahaDB message store to use serialized store and dispatch, you must explicitly disable concurrent store and dispatch for queues. [Example 2.2, “KahaDB Configured with Serialized Store and Dispatch”](#) explicitly disables the store and dispatch feature for queues and topics.

Example 2.2. KahaDB Configured with Serialized Store and Dispatch

```

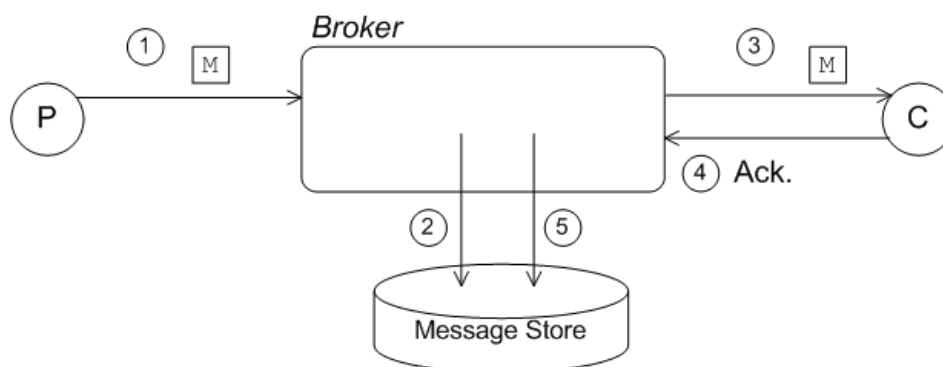
<broker brokerName="broker" persistent="true" useShutdownHook="false">
...
<persistenceAdapter>
  <kahaDB directory="activemq-data"
    journalMaxFileLength="32mb"
    concurrentStoreAndDispatchQueues="false"
    concurrentStoreAndDispatchTopics="false"
  />
</persistenceAdapter>
</broker>

```

Serialized store and dispatch

Figure 2.4, “Serialized Store and Dispatch” shows an outline what happens in the broker when concurrent store and dispatch is *disabled*, so that the store and dispatch steps are performed in sequence.

Figure 2.4. Serialized Store and Dispatch



In the serialized case, the store and dispatch steps occur as follows:

1. The producer sends a message, **M**, to a destination on the broker.
2. The broker sends the message, **M**, to the persistence layer. Because concurrency is disabled, the message is immediately written to the journal (assuming **enableJournalDiskSyncs** is **true**).
3. The message is dispatched to one or more consumers.
4. The consumers acknowledge receipt of the message.
5. When all of the consumer acknowledgments are received, the broker asks the persistence layer to remove the message from persistent storage (in the case of the KahaDB, this means that the persistence layer records in the journal that delivery of this message is now complete).

JMS durability requirements

In order to avoid losing messages, the JMS specification requires the broker to persist each message received from a producer, *before* sending an acknowledgment back to the producer. In the case of JMS transactions, the requirement is to persist the transaction data (including the messages in the transaction scope), before acknowledging a *commit* directive. Both of these conditions ensure that data is not lost.

Make sure that the message saves are synced to disk right away by setting the **kahaDB** element's **enableJournalDiskSyncs** attribute to **true**.

**NOTE**

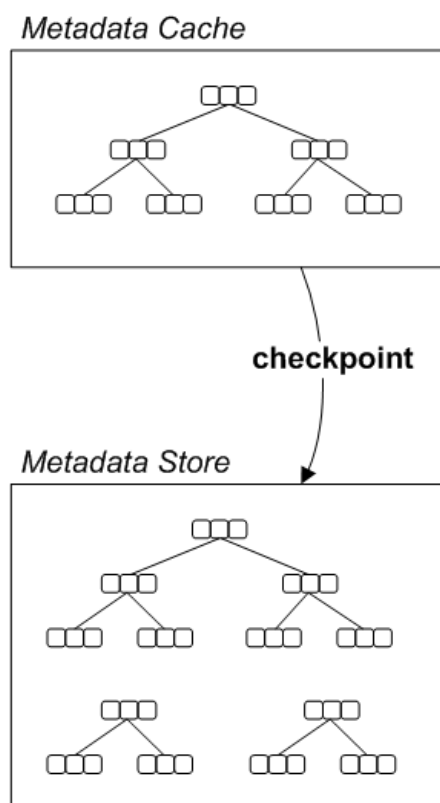
true is the default value for the **enableJournalDiskSyncs** attribute.

2.4. OPTIMIZING THE METADATA CACHE

Overview

Proper configuration of the metadata cache is one of the key factors affecting the performance of the KahaDB message store. In a production deployment, therefore, you should always take the time to tune the properties of the metadata cache for maximum performance. [Figure 2.5, “Overview of the Metadata Cache and Store”](#) shows an overview of the metadata cache and how it interacts with the metadata store. The most important part of the metadata is the B-tree index, which is shown as a tree of nodes in the figure. The data in the cache is periodically synchronized with the metadata store, when a checkpoint is performed.

Figure 2.5. Overview of the Metadata Cache and Store



Synchronizing with the metadata store

The metadata in the cache is constantly changing, in response to the events occurring in the broker. It is therefore necessary to write the metadata cache to disk, from time to time, in order to restore consistency between the metadata cache and the metadata store. There are two distinct mechanisms that can trigger a synchronization between the cache and the store, as follows:

- *Batch threshold*—as more and more of the B-tree indexes are changed, and thus inconsistent with the metadata store, you can define a threshold for the number of these *dirty indexes*. When the number of dirty indexes exceeds the threshold, KahaDB writes the cache to the store. The threshold value is set using the **indexWriteBatchSize** property.

- *Checkpoint interval*—irrespective of the current number of dirty indexes, the cache is synchronized with the store at regular time intervals, where the time interval is specified in milliseconds using the **checkpointInterval** property.

In addition, during a normal shutdown, the final state of the cache is synchronized with the store.

Setting the cache size

In the ideal case, the cache should be big enough to hold *all* of the KahaDB metadata in memory. Otherwise, the cache is forced to swap pages in and out of the persistent metadata store, which causes a considerable drag on performance.

You can specify the cache size using the **indexCacheSize** property, which specifies the size of the cache in units of pages (where one page is 4 KB by default). Generally, the cache should be as large as possible. You can check the size of your metadata store file, **db.data**, to get some idea of how big the cache needs to be.

Setting the write batch size

The **indexWriteBatchSize** defines the threshold for the number of dirty indexes that are allowed to accumulate, before KahaDB writes the cache to the store. Normally, these batch writes occur between checkpoints.

If you want to maximize performance, however, you could suppress the batch writes by setting **indexWriteBatchSize** to a very large number. In this case, the store would be updated only during checkpoints. The tradeoff here is that there is a risk of losing a relatively large amount of metadata, in the event of a system failure (but the broker should be able to restore the lost metadata when it restarts, by reading the tail of the journal).

2.5. CREATING AN OFFLINE BACKUP (OPENSIFT)

Overview

When using Red Hat JBoss A-MQ on Red Hat OpenShift Container Platform 3.x, you can create an offline backup of your A-MQ container for the KahaDB message store. An offline backup is the safest way to backup your data because it avoids the risk of copying data that might be in the process of being updated.

Prerequisites

Before you begin, you should be familiar with the OpenShift Container Platform command line interface (CLI). See the OpenShift Container platform documentation (<https://docs.openshift.com/index.html>), specifically the *Developer's Guide* and the *CLI Reference*.

Procedure

To create an offline backup for the KahaDB message store, follow these steps:

Procedure 2.1. Creating an offline backup

1. Use the **oc scale** command to stop the A-MQ broker pod to make sure that no writes occur during the backup process:

```
oc scale <object_type> <object_name> --replicas=<#_of_replicas>
```

In the following example, deployment configuration (dc) is the object-type and broker-amq is the object-name, and the number of replicas is set to zero:

```
oc scale dc broker-amq --replicas=0
```

2. Create and run a no-op container to use for the backup process. In the following example, **temp** is the name of the no-op container and **registry.access.redhat.com/rhel7** is the image for the container to run. The **--tail -f** option specifies to send logs about the **run** command to a local **/dev/null** folder and keeps the container running by preventing the replication process from closing it prematurely.

```
oc run temp --image=registry.access.redhat.com/rhel7 --tail -f /dev/null
```

3. Mount the A-MQ broker's persistent volume claims (PVC) to the no-op container by using the **oc volume add** command as shown in the following example:

```
oc volume dc/temp --add -t pvc --name=broker-amq-claim --claim-name=broker-amq-claim --mount-path=/opt/amq/old-data
```

4. Copy the the A-MQ broker's data from the no-op container to a local directory (for example **/opt/amq/old-data**):

```
oc rsync temp-2-b5p6d:/opt/amq/old-data
```

5. Remove the no-op container:

```
oc delete dc temp
```

6. Use the **oc scale** command to start the A-MQ broker pod:

```
oc scale dc broker-amq --replicas=1
```

2.6. RECOVERY

Overview

KahaDB supports a variety of mechanisms that enable it to recover and restart after a disorderly shutdown (system failure). This includes features to detect missing data files and to restore corrupted metadata. These features on their own, however, are *not* sufficient to guard completely against loss of data in the event of a system failure. If your broker is expected to mediate critical data, it is recommended that you deploy a disaster recovery system, such as a [RAID](#) disk array, to protect your data.

Clean shutdown

When the broker shuts down normally, the KahaDB message store flushes its cached data (representing the final state of the broker) to the file system. Specifically, the following information is written to the file system:

- All of the outstanding journal entries.
- All of the cached metadata.

Because this data represents the final state of the broker, the metadata store and the journal's data logs are consistent with each other after shutdown is complete. That is, the stored metadata takes into account *all* the commands recorded in the journal.

Recovery from disorderly shutdown

Normally, the journal tends to run ahead of the metadata store, because the journal is constantly being updated, whereas the metadata store is written only periodically (for example, whenever there is a checkpoint). Consequently, whenever there is a disorderly shutdown (which prevents the final state of the broker from being saved), it is likely that the stored metadata will be inconsistent with the journal, with the journal containing additional events not reflected in the metadata store.

When the broker restarts after a disorderly shutdown, the KahaDB message store recovers by reading the stored metadata into the cache and then reading the additional journal events not yet taken into account in the stored metadata (KahaDB can easily locate the additional journal events, because the metadata store always holds a reference to the last consistent location in the journal). KahaDB replays the additional journal events in order to recreate the original metadata.



NOTE

The KahaDB message store also uses a redo log, **db.redo**, to reduce the risk of a system failure occurring while updating the metadata store. Before updating the metadata store, KahaDB always saves the redo log, which summarizes the changes that are about to be made to the store. Because the redo log is a small file, it can be written relatively rapidly and is thus less likely to be affected by a system failure. During recovery, KahaDB checks whether the changes recorded in the redo log need to be applied to the metadata.

Forcing recovery by deleting the metadata store

If the metadata store somehow becomes irretrievably corrupted, you can force recovery as follows (assuming the journal's data logs are clean):

1. While the broker is shut down, delete the metadata store, **db.data**.
2. Start the broker.
3. The broker now recovers by re-reading the *entire* journal and replaying all of the events in the journal in order to recreate the missing metadata.

While this is an effective means of recovering, you should bear in mind that it could take a considerable length of time if the journal is large.

Missing journal files

KahaDB has the ability to detect when journal files are missing. If one or more journal files are detected to be missing, the default behavior is for the broker to raise an exception and shut down. This gives an administrator the opportunity to investigate what happened to the missing journal files and to restore them manually, if necessary.

If you want the broker to ignore any missing journal files and continue processing regardless, you can set the **ignoreMissingJournalfiles** property to **true**.

Checking for corrupted journal files

KahaDB has a feature that checks for corrupted journal files, but this feature must be explicitly enabled. [Example 2.3, "Configuration for Journal Validation"](#) shows how to configure a KahaDB message store to detect corrupted journal files.

Example 2.3. Configuration for Journal Validation

```
<persistenceAdapter>
  <kahaDB directory="activemq-data"
    journalMaxFileLength="32mb"
    checksumJournalFiles="true"
    checkForCorruptJournalFiles="true"    />
</persistenceAdapter>
```

CHAPTER 3. USING A MULTI KAHADB PERSISTENCE ADAPTER

Abstract

When you have destinations with different performance profiles or different persistence requirements you can distribute them across multiple KahaDB message stores.

OVERVIEW

The stock KahaDB persistence adapter works well when all of the destinations being managed by the broker have similar performance and reliability profiles. When one destination has a radically different performance profile, for example its consumer is exceptionally slow compared to the consumers on other destinations, the message store's disk usage can grow rapidly. When one or more destinations don't require disc synchronization and the others do require it, all of the destinations must take the performance hit.

The *multi KahaDB persistence adapter* allows you to distribute a broker's destinations across multiple KahaDB message stores. Using multiple message stores allows you to tailor the message store more precisely to the needs of the destinations using it. Destinations and stores are matched using filters that take standard wild card syntax.

CONFIGURATION

The multi KahaDB persistence adapter configuration wraps more than one KahaDB message store configuration.

The multi KahaDB persistence adapter configuration is specified using the **mKahaDB** element. The **mKahaDB** element has a single attribute, **directory**, that specifies the location where the adapter writes its data stores. This setting is the default value for the **directory** attribute of the embedded KahaDB message store instances. The individual message stores can override this default setting.

The **mKahaDB** element has a single child **filteredPersistenceAdapters**. The **filteredPersistenceAdapters** element contains multiple **filteredKahaDB** elements that configure the KahaDB message stores that are used by the persistence adapter.

Each **filteredKahaDB** element configures one KahaDB message store (except in the case where the **perDestination** attribute is set to **true**). The destinations matched to the message store are specified using attributes on the **filteredKahaDB** element:

- **queue**—specifies the name of queues
- **topic**—specifies the name of topics

The destinations can be specified either using explicit destination names or using wildcards. For information on using wildcards see [the section called "Wildcard syntax"](#). If no destinations are specified the message store will match any destinations that are not matched by other filters.

The KahaDB message store configured inside a **filteredKahaDB** element is configured using the standard KahaDB persistence adapter configuration. It consists of a **kahaDB** element wrapped in a **persistenceAdapter** element. For details on configuring a KahaDB message store see [Section 2.2, "Configuring the KahaDB Message Store"](#).

WILDCARD SYNTAX

You can use wildcards to specify a group of destination names. This is useful for situations where your destinations are set up in federated hierarchies.

For example, imagine you are sending price messages from a stock exchange feed. You might name your destinations as follows:

- **PRICE.STOCK.NASDAQ.ORCL** to publish Oracle Corporation's price on NASDAQ
- **PRICE.STOCK.NYSE.IBM** to publish IBM's price on the New York Stock Exchange

You could use exact destination names to specify which message store will be used to persist message data, or you could use wildcards to define hierarchical pattern matches to the pair the destinations with a message store.

Red Hat JBoss A-MQ uses the following wild cards:

- `.` separates names in a path
- `*` matches any name in a path
- `>` recursively matches any destination starting from this name

For example using the names above, these filters are possible:

- **PRICE.>**—any price for any product on any exchange
- **PRICE.STOCK.>**—any price for a stock on any exchange
- **PRICE.STOCK.NASDAQ.***—any stock price on NASDAQ
- **PRICE.STOCK.*.IBM**—any IBM stock price on any exchange

EXAMPLE

[Example 3.1, "Multi KahaDB Persistence Adapter Configuration"](#) shows a multi KahaDB persistence adapter that distributes destinations across two KahaDB message stores. The first message store is used for all queues managed by the broker. The second message store is used for all other destinations (in this case, for all topics).

Example 3.1. Multi KahaDB Persistence Adapter Configuration

```
<persistenceAdapter>
  <mKahaDB directory="{activemq.base}/data/kahadb">
    <filteredPersistenceAdapters>
      <!-- match all queues -->
      <filteredKahaDB queue="">
        <persistenceAdapter>
          <kahaDB journalMaxFileLength="32mb"/>
        </persistenceAdapter>
      </filteredKahaDB>

      <!-- match all destinations -->
      <filteredKahaDB>
        <persistenceAdapter>
```

```

    <kahaDB enableJournalDiskSyncs="false"/>
  </persistenceAdapter>
</filteredKahaDB>
</filteredPersistenceAdapters>
</mKahaDB>
</persistenceAdapter>

```

AUTOMATIC PER-DESTINATION PERSISTENCE ADAPTER

When the **perDestination** attribute is set to **true** on the catch-all **filteredKahaDB** element (that is, the instance of **filteredKahaDB** that specifies neither a **queue** nor a **topic** attribute), every matching destination gets its own **kahaDB** instance. For example, the following sample configuration shows how to configure a per-destination persistence adapter:

```

<broker brokerName="broker" ... >
  <persistenceAdapter>
    <mKahaDB directory="{activemq.base}/data/kahadb">
      <filteredPersistenceAdapters>
        <!-- kahaDB per destinations -->
        <filteredKahaDB perDestination="true" >
          <persistenceAdapter>
            <kahaDB journalMaxFileLength="32mb" />
          </persistenceAdapter>
        </filteredKahaDB>
      </filteredPersistenceAdapters>
    </mKahaDB>
  </persistenceAdapter>
  ...
</broker>

```



NOTE

Combining the **perDestination** attribute with either the **queue** or **topic** attributes has not been verified to work and could cause runtime errors.

TRANSACTIONS

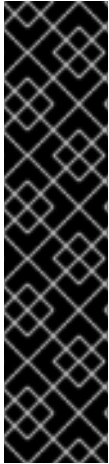
Transactions can span multiple journals if the destinations are distributed. This means that two phase completion is required. This does incur the performance penalty of the additional disk sync to record the commit outcome.

If only one journal is involved in the transaction, the additional disk sync is not used. The performance penalty is not incurred in this case.

CHAPTER 4. USING THE LEVELDB PERSISTENCE ADAPTER

Abstract

The LevelDB persistence adapter uses LevelDB as a high-performance message store. It allows for higher throughput speeds than the default message store.



IMPORTANT

The LevelDB store is supported only on the Red Hat Enterprise Linux (RHEL) platform (for the RHEL versions and configurations specified in *Supported Configurations*). If you want to store the LevelDB files on a distributed file system, the following configuration is also supported: the broker itself and the LevelDB adapter plugin run on a supported RHEL platform, while the LevelDB files are stored on an NFSv4 file system (the O/S of the server that hosts the NFSv4 file system is irrelevant). Note that the distributed file system *must* be NFSv4.

For all other JBoss A-MQ-supported platforms, the LevelDB store is provided for technical preview only. For details on what "technical preview" means, see <https://access.redhat.com/support/offerings/techpreview/>.

OVERVIEW

The LevelDB message store is a file based message store implemented using Google's LevelDB library to maintain indexes into log files holding the messages. The main advantages of the LevelDB store include:

- higher persistent throughput
- faster recovery times when a broker restarts
- supports concurrent read access
- no pausing during garbage collection cycles
- uses fewer read IO operations to load stored messages
- supports XA transactions
- checks for duplicate messages
- exposes status via JMX for monitoring
- supports replication

PLATFORM SUPPORT

LevelDB is implemented in C++, and Red Hat JBoss A-MQ accesses the libraries using a JNI driver.

JBoss A-MQ also provides an experimental pure Java driver for all other platforms.

BASIC CONFIGURATION

The LevelDB message store is configured by placing a **levelDB** element in the **persistenceAdapter** element of your broker's configuration. The **levelDB** element's attributes are used to configure the message store.

All attributes, listed in [Table 4.1, "Configuration Properties of the LevelDB Message Store—standard LevelDB attributes"](#), have reasonable default values, so you are not required to specify values for them. However, you will need to explicitly specify the location of the message store's data files by providing a value for the **directory** attribute. This will ensure that the broker will not conflict with other brokers.

[Example 4.1, "Configuring the LevelDB Message Store"](#) shows a basic configuration of the LevelDB message store. The LevelDB files are stored under the **activemq-data** directory.

Example 4.1. Configuring the LevelDB Message Store

```
<broker brokerName="broker" persistent="true" ... >
...
<persistenceAdapter>
  <levelDB directory="activemq-data" />
</persistenceAdapter>
...
</broker>
```

CONFIGURATION ATTRIBUTES

[Table 4.1, "Configuration Properties of the LevelDB Message Store—standard LevelDB attributes"](#) describes the attributes that can be used to configure the LevelDB message store.

Table 4.1. Configuration Properties of the LevelDB Message Store—standard LevelDB attributes

Attribute	Default Value	Description
directory	activemq-data	Specifies the path to the top-level folder that holds the message store's data files.
readThreads	10	Specifies the number of concurrent IO reads to allow.
sync	true	Specifies whether syncs log operations to disk.
logSize	104857600	Specifies the maximum size, in bytes, of each data log file before log file rotation occurs.
verifyChecksums	false	Specifies whether checksum verification is performed on all data that is read from the file system.

Attribute	Default Value	Description
paranoidChecks	false	Specifies whether the store errors out as soon as possible when it detects internal corruption.
indexFactory	org.fusesource.leveldbjni.JniDBFactory, org.iq80.leveldb.impl.Iq80DBFactory	<p>Specifies a comma separated list of leveldb API implementation factory classes that the broker will attempt to load. The broker will use the first one that loads successfully.</p> <ul style="list-style-type: none"> • org.fusesource.leveldbjni.JniDBFactory enables the JNI base implementation • org.iq80.leveldb.impl.Iq80DBFactory enables the pure Java implementation
indexMaxOpenFiles	1000	Specifies the number of open files that can be used by the index.
indexBlockRestartInterval	16	Specifies the number of keys between restart points for delta encoding of keys.
indexWriteBufferSize	4194304	Specifies the amount, in bytes, of index data to build up in memory before converting to a sorted on-disk file.
indexBlockSize	4096	Specifies the size, in bytes, of index data packed per block.
indexCacheSize	268435456	Specifies the maximum amount, in bytes, of memory to use to cache index blocks.
indexCompression	snappy	Specifies the type of compression to apply to the index blocks. Can be snappy or none .
logCompression	snappy	Specifies the type of compression to apply to the log records. Can be snappy or none .

Table 4.2. Configuration Properties of the LevelDB Message Store—pluggable storage locker attributes

Attribute	Default Value	Description
failIfLocked	false	Specifies whether the broker will fail on start up if the message store's data files are locked. If the broker does not fail, it blocks until the data files are unlocked.
useLock	true	Specifies whether the adapter uses file locking.

CHAPTER 5. USING THE REPLICATED LEVELDB PERSISTENCE ADAPTER

Abstract

The Replicated LevelDB persistence adapter uses Apache ZooKeeper to select a master from a cluster of broker nodes that are configured to replicate a levelDB store. The Replicated LevelDB persistence adapter then synchronizes all slave LevelDB stores with the master LevelDB store by replicating the master broker's updates to all slave brokers in the cluster.

Because the Replicated LevelDB store uses the same data files as the regular LevelDB store, you can switch a broker's LevelDB configuration between replicated and regular at any time.

OVERVIEW



IMPORTANT

The Replicated LevelDB persistence adapter is provided for technical preview only, and is not suitable for production environments. For details on what "technical preview" means, see <https://access.redhat.com/support/offerings/techpreview/>.

The Replicated LevelDB message store uses the same file-based store implemented using Google's LevelDB library. As such, it provides the same advantages and runs on the same platforms as the LevelDB persistence adapter (for details, see [Using the LevelDB Persistence Adapter](#)).

The Replicated LevelDB store uses Apache ZooKeeper to coordinate and select which broker node in the cluster becomes master. Only the master accepts and starts client connections. All other broker nodes enter slave mode and connect to the master, synchronizing their persistence state with it. The master node then replicates all persistent operations to the connected slaves.

When the master dies, Apache ZooKeeper selects a slave that has the latest updates to be the new master. Once the new master is activated, the old master can be brought back online, at which time it enters slave mode.

All messaging operations that require a sync-to-disk wait for the update to be replicated to a quorum of slave nodes before the operations complete. For example, a store configured with **replicas="3"** has a quorum size of $(3/2)+1=2$. In this case, the master stores the update locally, then waits for at least one slave to store the update before it reports success.

To select a new master, a quorum of nodes must be online for ZooKeeper to find a slave with the latest updates. Therefore, it's recommend that you run with at least three replica nodes, so you can take one down without suffering a service outage.

DEPLOYMENT TIPS

- Clients should use the Failover Transport to connect to the broker nodes in the replication cluster; for example, using a URL like this:

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

- To enable highly available ZooKeeper service, run at least three ZooKeeper server nodes.

**NOTE**

Avoid overcommitting ZooKeeper servers. An overworked ZooKeeper server might infer that a live broker replication node has gone offline due to delayed processing of keep-alive messages.

For details on setting up and running a distributed cluster of Apache ZooKeeper servers, see the ZooKeeper [Getting Started](#) document.

- To enable highly available ActiveMQ service, run at least three replicated broker nodes.

**NOTE**

Though [Example 5.1, "Configuring the Replicated LevelDB Message Store"](#) configures three replicated broker nodes and three ZooKeeper servers, having the same number of ZooKeeper nodes as replicated broker nodes is not required. Both the ZooKeeper service and the messaging service operate under the same outage probability formula, wherein running three nodes allows one node to fail without incurring a service outage, running five nodes allows two nodes to fail simultaneously without incurring a service outage, and so on. Applications that must meet stringent high availability requirements might configure more ZooKeeper nodes than replicated broker nodes, as the messaging service depends on the ZooKeeper service and is limited by its availability.

BASIC CONFIGURATION

To configure the Replicated LevelDB message store, place a **replicatedLevelDB** element in the **persistenceAdapter** element of your broker's configuration, and use the **replicatedLevelDB** element's attributes to configure the message store.

**IMPORTANT**

All broker nodes in the same replication cluster must use the same value in the **brokerName** attribute.

[Example 5.1, "Configuring the Replicated LevelDB Message Store"](#) shows a basic configuration of the Replicated LevelDB message store. The Replicated LevelDB files are stored under the **activemq-data** directory.

Example 5.1. Configuring the Replicated LevelDB Message Store

```
<broker brokerName="broker" persistent="true" ... >
...
<persistenceAdapter>
  <replicatedLevelDB
    directory="activemq-data" />
    replicas="3"
    bind="tcp://0.0.0.0:0"
    zkAddress="zoo1.example.org:2181,zoo2.example.org:2181,zoo3.example.org:2181"
    zkPassword="password"
    zkPath="/activemq/leveldb-stores"
  />
</persistenceAdapter>
...
</broker>
```

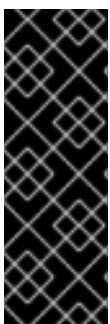
CONFIGURATION ATTRIBUTES

The attributes listed in Table 5.1, “Configuration Properties of the Replicated LevelDB Message Store—attributes which must be identical on all broker nodes in a replication cluster” must be configured with the same value on all broker nodes in a replication cluster.

Table 5.1. Configuration Properties of the Replicated LevelDB Message Store—attributes which must be identical on all broker nodes in a replication cluster

Attribute	Default Value	Description
replicas	3	Specifies the number of replicated stores the replication cluster will contain. At least (replicas/2)+1 nodes must be online to avoid messaging service outages.
securityToken		Specifies the security token to use, which must match on all replication nodes in the cluster for the nodes to accept each other's replication requests.
zkAddress	127.0.0.1:2181	A comma-separated list of addresses that specify the ZooKeeper servers managing the LevelDB stores in the cluster.
zkPassword		Specifies the password to use for connecting to the ZooKeeper servers.
zkPath	/default	Specifies the path to the ZooKeeper directory in which information about master/slave selection is exchanged.

Attribute	Default Value	Description
zkSessionTimeout	2s	<p>Specifies the time limit by which the broker will detect a network failure. Valid units are:</p> <ul style="list-style-type: none"> ● s = seconds ● m = minutes ● h = hours ● d = days ● w = weeks ● M = months ● y = years <p>Specifying a number without a suffix (s, m, h,...y) selects milliseconds.</p> <p>You can also combine units for more fine-grained scheduling; for example, 10m30s.</p>
sync	quorum_mem	<p>Controls where updates are stored before they are considered as having completed.</p> <p>The options are: local_mem, local_disk, remote_mem, remote_disk, quorum_mem, and quorum_disk</p> <p>If you specify multiple options—in a comma-separated list—the stronger guarantee is used.</p> <p>For example, specifying local_mem,local_disk is the same as specifying local_disk; specifying quorum_mem is the same as specifying local_mem,remote_mem; and quorum_disk is the same as specifying local_disk,remote_disk.</p>



IMPORTANT

The broker uses **zkSessionTimeout** to detect when it has been disconnected from the ZooKeeper server due to a network failure. When set to **2s**, the replicated broker nodes will detect a disconnect within two seconds of a network failure. Once the disconnect is detected, the master broker gives up the master role, and the slave brokers begin the election process. The lower the timeout value, the faster the process to select a new master. However, setting the timeout value too low can result in false positives, causing masters to switch when no disconnect has occurred.

The attributes listed in [Table 5.2, “Configuration Properties of the Replicated LevelDB Message Store—attributes which can be unique for each broker node in a replication cluster”](#) can be unique per broker node in a replication cluster.

Table 5.2. Configuration Properties of the Replicated LevelDB Message Store—attributes which can be unique for each broker node in a replication cluster

Attribute	Default Value	Description
bind	tcp://0.0.0.0:61619	Specifies the address and port to which the broker will bind to service the replication protocol, when it becomes master. To configure dynamic ports, use tcp://0.0.0.0:0 .
hostname		Specifies the hostname to use for advertising the replication service when the broker node becomes master. When left unset, the messaging service automatically determines the hostname. It's possible for the messaging service to incorrectly determine the hostname. For example, it might select localhost , which would prevent remote slave brokers from connecting to the master broker.



NOTE

Except for the Pluggable Storage Lockers, the Replicated LevelDB store supports all of the standard LevelDB store configuration attributes. For details, see [Table 4.1, “Configuration Properties of the LevelDB Message Store—standard LevelDB attributes”](#) in [Using the LevelDB Persistence Adapter](#).

CHAPTER 6. USING JDBC TO CONNECT TO A DATABASE STORE

Abstract

Red Hat JBoss A-MQ supports the use of relational databases as a message store through JDBC.

6.1. BASICS OF USING THE JDBC PERSISTENCE ADAPTER

Overview

For long term persistence you may want to use a relational database as your persistent message store. Red Hat JBoss A-MQ's default database when using the JDBC persistence adapter is Apache Derby. JBoss A-MQ also supports most major SQL databases. You can enable other databases by properly configuring the JDBC connection in the broker's configuration file.

Supported databases

JBoss A-MQ is known to work with the following databases:

- MySQL
- Oracle
- PostgreSQL
- Microsoft SQL Server

For full details of the compatible database versions, see [Red Hat JBoss A-MQ Supported Configurations](#).

Specifying the type of JDBC store to use

JBoss A-MQ support two types of JDBC store:

- *JDBC store:*

The regular JDBC store is specified using the **jdbcPersistenceAdapter** element inside the **persistenceAdapter** element. For more details see [Section 6.2, "Using the Plain JDBC Adapter"](#).

- *Journalled JDBC store:*

(Deprecated) The journalled JDBC store is specified using the **journalledJDBC** element inside the **persistenceFactory** element. For more details see [Section 6.3, "Using JDBC with the High Performance Journal"](#).

**WARNING**

The journaled JDBC store is incompatible with the JDBC master/slave failover pattern—see *Fault Tolerant Messaging*.

The journaled JDBC store features better performance than the plain JDBC store.

**WARNING**

The journaled JDBC store is incompatible with the JDBC master/slave failover pattern—see *Fault Tolerant Messaging*.

Prerequisites

Before you can use one of the JDBC persistence stores you need to ensure that the following are installed in the broker's container:

- The **org.apache.servicemix.bundles.commons-dbc** bundle. You can install this bundle into a standalone container using the following console command:

```
osgi:install mvn:org.apache.servicemix.bundles/org.apache.servicemix.bundles.commons-dbc/1.4_3
```

- The JDBC driver for the database being used.

**NOTE**

Depending on the database being used, you may need to wrap the driver in an OSGi bundle by using the wrap: URI prefix when adding it to the container.

Configuring your JDBC driver

JBoss A-MQ autodetects the JDBC driver that is in use at start-up. For the supported databases, the JDBC adapter automatically adjusts the SQL statements and JDBC driver methods to work with the driver. If you wish to customize the names of the database tables or work with an unsupported database, you can modify both the SQL statements and the JDBC driver methods. See [the section called "Customizing the SQL statements used by the adapter"](#) for information about modifying the SQL statements. See [the section called "Using generic JDBC providers"](#) for information about changing the JDBC methods.

JDBC configuration for Apache Derby

[Example 6.1, "Configuration for the Apache Derby Database"](#) shows the configuration for using the default Apache Derby JDBC driver.

Example 6.1. Configuration for the Apache Derby Database

```

<beans ...>
  <broker xmlns="http://activemq.apache.org/schema/core"
    brokerName="localhost">
    ...
    <persistenceAdapter>
      <jdbcPersistenceAdapter
        dataDirectory="{activemq.base}/data"
        dataSource="#derby-ds"/>
    </persistenceAdapter>
    ...
  </broker>

  <!-- Embedded Derby DataSource Sample Setup -->
  <bean id="derby-ds" class="org.apache.derby.jdbc.EmbeddedDataSource">
    <property name="databaseName" value="derbydb"/>
    <property name="createDatabase" value="create"/>
  </bean>

</beans>

```

JDBC configuration for Oracle

[Example 6.2, "Configuration for the Oracle JDBC Driver"](#) shows the configuration for using the Oracle JDBC driver. The persistence adapter configuration refers to the Spring **bean** element that configures the JDBC driver.

Example 6.2. Configuration for the Oracle JDBC Driver

```

<beans ... >
  <broker xmlns="http://activemq.apache.org/schema/core"
    brokerName="localhost">
    ...
    <persistenceAdapter>
      <jdbcPersistenceAdapter
        dataDirectory="{activemq.base}/data"
        dataSource="#oracle-ds"/>
    </persistenceAdapter>
    ...
  </broker>

  <!-- Oracle DataSource Sample Setup -->
  <bean id="oracle-ds"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:AMQDB"/>
    <property name="username" value="scott"/>
    <property name="password" value="tiger"/>
    <property name="maxActive" value="200"/>
    <property name="poolPreparedStatements" value="true"/>
  </bean>

```



```

</bean>
</beans>

```

The JDBC drivers are configured using a Spring **bean** element. The **id** attribute specifies the name by which you will refer to the driver when configuring the JDBC persistence adapter. The **class** attribute specifies the class that implements the data source used to interface with the JDBC driver. The **destroy-method** attribute specifies the name of the method to call when the JDBC driver is shutdown.

In addition to the **bean** element, the JDBC driver configuration includes a number of **property** elements. Each **property** element specifies a property required by the JDBC driver. For information about the configurable properties refer to your JDBC driver's documentation.

6.2. USING THE PLAIN JDBC ADAPTER

Overview

This section describes how to use the plain JDBC adapter, which is the recommended adapter to use for most applications.

Example

Example 6.3, “Configuring Red Hat JBoss A-MQ to use the Plain JDBC Persistence Adapter” shows a configuration fragment that configures the plain JDBC adapter to use the Apache Derby database.

Example 6.3. Configuring Red Hat JBoss A-MQ to use the Plain JDBC Persistence Adapter

```

<beans ... >
  <broker ...>
  ...
  1 <persistenceAdapter>
    2 <jdbcPersistenceAdapter dataSource="#derby-ds" />
    </persistenceAdapter>
  ...
  </broker>
  ...
  3 <bean id="derby-ds" class="org.apache.derby.jdbc.EmbeddedDataSource">
    <property name="databaseName" value="derbydb"/>
    <property name="createDatabase" value="create"/>
  </bean>

```

The configuration in Example 6.3, “Configuring Red Hat JBoss A-MQ to use the Plain JDBC Persistence Adapter” has three noteworthy elements:

- 1 The **persistenceAdapter** element wraps the configuration for the JDBC persistence adapter.
- 2 The **jdbcPersistenceAdapter** element specifies that the broker will use the plain JDBC persistence adapter and that the JDBC driver's configuration is specified in a **bean** element with the ID, **derby-ds**.
- 3 The **bean** element specified the configuration for the Derby JDBC driver.

Configuration

Table 6.1, “Attributes for Configuring the Plain JDBC Persistence Adapter” describes the attributes used to configure the plain JDBC persistence adapter.

Table 6.1. Attributes for Configuring the Plain JDBC Persistence Adapter

Attribute	Default Value	Description
adapter		Specifies the strategy to use when accessing a non-supported database. For more information see the section called “Using generic JDBC providers” .
cleanupPeriod	300000	Specifies, in milliseconds, the interval at which acknowledged messages are deleted.
createTablesOnStartup	true	Specifies whether or not new database tables are created when the broker starts. If the database tables already exist, the existing tables are reused.
dataDirectory	activemq-data	Specifies the directory into which the default Derby database writes its files.
dataSource	#derby	Specifies the id of the Spring bean storing the JDBC driver's configuration. For more information see the section called “Configuring your JDBC driver” .
transactionIsolation	Connection.TRANSACTION_READ_UNCOMMITTED	Specifies the required transaction isolation level. For allowed values, see java.sql.Connection .
useLock	true	Specifies in the adapter uses file locking.
lockKeepAlivePeriod	30000	Specifies the time period, in milliseconds, at which the current time is saved in the locker table to ensure that the lock does not timeout. 0 specifies unlimited time.

6.3. USING JDBC WITH THE HIGH PERFORMANCE JOURNAL

Overview

The journaled JDBC store is *deprecated* in this release. The journaled JDBC store was designed to optimize performance where there is a slow connection to the remote database. With modern high-speed networks, however, the advantage of this optimization is negligible.



WARNING

The journaled JDBC store is deprecated from JBoss A-MQ 6.2 onwards and may be removed in a future release.



WARNING

The journaled JDBC store is incompatible with the JDBC master/slave failover pattern—see *Fault Tolerant Messaging*.

Prerequisites

Before you can use the journaled JDBC persistence store you need to ensure that the **activeio-core-3.1.4.jar** bundle is installed in the container.

The bundle is available in the archived ActiveMQ installation included in the **InstallDir/extras** folder or can be downloaded from Maven at <http://mvnrepository.com/artifact/org.apache.activemq/activeio-core/3.1.4>.

Example

Example 6.4, “Configuring Red Hat JBoss A-MQ to use the Journaled JDBC Persistence Adapter” shows a configuration fragment that configures the journaled JDBC adapter to use a MySQL database.

Example 6.4. Configuring Red Hat JBoss A-MQ to use the Journaled JDBC Persistence Adapter

```

<beans ... >
  <broker ...>
  ...
  1 <persistenceFactory>
    2 <journalPersistenceAdapterFactory journalLogFiles="5" dataDirectory="${data}/kahadb"
      dataSource="#mysql-ds" useDatabaseLock="true" useDedicatedTaskRunner="false />
    </persistenceFactory>
  ...
  <broker>
  ...
  3 <bean id="mysql-ds"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close">

```

```

<property name="driverClassName" value="com.mysql.jdbc.Driver"/>
<property name="url" value="jdbc:mysql://localhost/activemq?relaxAutoCommit=true"/>
<property name="username" value="activemq"/>
<property name="password" value="activemq"/>
<property name="poolPreparedStatements" value="true"/>
</bean>

```

The configuration in [Example 6.4, “Configuring Red Hat JBoss A-MQ to use the Journalized JDBC Persistence Adapter”](#) has three noteworthy elements:

- 1 The **persistenceFactory** element wraps the configuration for the JDBC persistence adapter.
- 2 The **journalizedJDBC** element specifies that the broker will use the JDBC persistence adapter with the high performance journal. The element's attributes configure the following properties:
 - The journal will span five log files.
 - The configuration for the JDBC driver is specified in a **bean** element with the ID, **mysql-ds**.
 - The data for the journal will be stored in **`\${data}/kahadb**.
- 3 The **bean** element specified the configuration for the MySQL JDBC driver.

Configuration

[Table 6.2, “Attributes for Configuring the Journalized JDBC Persistence Adapter”](#) describes the attributes used to configure the journalized JDBC persistence adapter.

Table 6.2. Attributes for Configuring the Journalized JDBC Persistence Adapter

Attribute	Default Value	Description
adapter		Specifies the strategy to use when accessing a non-supported database. For more information see the section called “Using generic JDBC providers” .
createTablesOnStartup	true	Specifies whether or not new database tables are created when the broker starts. If the database tables already exist, the existing tables are reused.
dataDirectory	activemq-data	Specifies the directory into which the default Derby database writes its files.

Attribute	Default Value	Description
dataSource	#derby	Specifies the id of the Spring bean storing the JDBC driver's configuration. For more information see the section called "Configuring your JDBC driver" .
journalArchiveDirectory		Specifies the directory used to store archived journal log files.
journalLogFiles	2	Specifies the number of log files to use for storing the journal.
journalLogFileSize	20MB	Specifies the size for a journal's log file.
journalThreadPriority	10	Specifies the thread priority of the thread used for journaling.
useJournal	true	Specifies whether or not to use the journal.
useLock	true	Specifies in the adapter uses file locking.
lockKeepAlivePeriod	30000	Specifies the time period, in milliseconds, at which the current time is saved in the locker table to ensure that the lock does not timeout. 0 specifies unlimited time.
checkpointInterval	1000 * 60 * 5	Specifies the time period, in milliseconds, between writing metadata cache to disk. .

6.4. CUSTOMIZING THE JDBC PERSISTENCE ADAPTER

Overview

Red Hat JBoss A-MQ provides options to customize the interaction between the JDBC persistence adapter and the underlying database. In some cases you might be able to use these customization options to integrate the JDBC persistence adapter with an unsupported database.

Customizing the SQL statements used by the adapter

You can customize the SQL statements that the JDBC persistence adapter uses to access the database. This is done by adding a **statements** element to the JDBC persistence adapter configuration. [Example 6.5, "Fine Tuning the Database Schema"](#) shows a configuration snippet that specifies that long

strings are going to be stored as VARCHAR(128).

Example 6.5. Fine Tuning the Database Schema

```
<persistenceAdapter>
  <jdbcPersistenceAdapter ... >
    <statements>
      <statements stringIdDataType = "VARCHAR(128)"/>
    </statements>
  </jdbcPersistenceAdapter>
</persistenceAdapter>
```

The first **statements** element is a wrapper for one or more nested **statements** elements. Each nested **statements** element specifies a single configuration statement. [Table 6.3, "Statements for Configuring the SQL Statements Used by the JDBC Persistence Adapter"](#) describes the configurable properties.

Table 6.3. Statements for Configuring the SQL Statements Used by the JDBC Persistence Adapter

Attribute	Default	Description
tablePrefix		Specifies a prefix that is added to every table name. The prefix should be unique per broker if multiple brokers will be sharing the same database.
messageTableName	ACTIVEMQ_MSGS	Specifies the name of the table in which persistent messages are stored.
durableSubAcksTableName	ACTIVEMQ_ACKS	Specifies the name of the database table used to store acknowledgment messages from durable subscribers.
lockTableName	ACTIVEMQ_LOCK	Specifies the name of the lock table used to determine the master in a master/slave scenario.
binaryDataType	BLOB	Specifies the data type used to store the messages.
containerNameDataType	VARCHAR(250)	Specifies the data type used to store the destination name.
msgIdDataType	VARCHAR(250)	Specifies the data type used to store a message id.
sequenceDataType	INTEGER	Specifies the datatype used to store the sequence id of a message.

Attribute	Default	Description
longDataType	BIGINT	Specifies the data type used to store a Java long.
stringIdDataType	VARCHAR(250)	Specifies the data type used to store long strings like client ids, selectors, and broker names.

The properties listed in [Table 6.3, "Statements for Configuring the SQL Statements Used by the JDBC Persistence Adapter"](#) configure the default SQL statements used by the JDBC adapter and work with all of the supported databases.

Customizing SQL statements for unsupported databases

If you need to override the default statements to work with an unsupported database, there are a number of other properties that can be set. These include:

- **addMessageStatement**
- **updateMessageStatement**
- **removeMessageStatement**
- **findMessageSequenceIdStatement**
- **findMessageStatement**
- **findAllMessagesStatement**
- **findLastSequenceIdInMsgsStatement**
- **findLastSequenceIdInAcksStatement**
- **createDurableSubStatement**
- **findDurableSubStatement**
- **findAllDurableSubsStatement**
- **updateLastAckOfDurableSubStatement**
- **deleteSubscriptionStatement**
- **findAllDurableSubMessagesStatement**
- **findDurableSubMessagesStatement**
- **findAllDestinationsStatement**

- `removeAllMessagesStatement`
- `removeAllSubscriptionsStatement`
- `deleteOldMessagesStatement`
- `lockCreateStatement`
- `lockUpdateStatement`
- `nextDurableSubscriberMessageStatement`
- `durableSubscriberMessageCountStatement`
- `lastAckedDurableSubscriberMessageStatement`
- `destinationMessageCountStatement`
- `findNextMessageStatement`
- `createSchemaStatements`
- `dropSchemaStatements`

Using generic JDBC providers

To use a JDBC provider not natively supported by Red Hat JBoss A-MQ, you can configure the JDBC persistence adapter, by setting the persistence adapter's **adapter** attribute to reference the bean ID of the relevant adapter. The following adapter types are supported:

- `org.activemq.store.jdbc.adapter.BlobJDBCAdapter`
- `org.activemq.store.jdbc.adapter.BytesJDBCAdapter`
- `org.activemq.store.jdbc.adapter.DefaultJDBCAdapter`
- `org.activemq.store.jdbc.adapter.ImageJDBCAdapter`

Various settings are provided to customize how the JDBC adapter stores and accesses BLOB fields in the database. To determine the proper settings, consult the documentation for your JDBC driver and your database.

[Example 6.6, "Configuring a Generic JDBC Provider"](#) shows a configuration snippet configuring the journaled JDBC persistence adapter to use the blob JDBC adapter.

Example 6.6. Configuring a Generic JDBC Provider

```
<broker persistent="true" ... >
...
<persistenceAdapter>
  <jdbcPersistenceAdapter adapter="#blobAdapter" ... />
</persistenceAdapter>

<bean id="blobAdapter"
```



```

class="org.activemq.store.jdbc.adapter.BlobJDBCAdapter"/>
...
</broker>

```

6.5. TUTORIAL: JDBC PERSISTENCE

Overview

This tutorial provides complete instructions for installing a JDBC persistence layer into the JBoss A-MQ broker, using the MySQL database to store the broker's data. This example uses a plain JDBC persistence adapter and uses the default database schema.

This tutorial assumes you are using a standalone JBoss A-MQ container, which is the condition of the container immediately after the product is installed. It does not cover the case of a Fabric container.

Prerequisites

Before following the instructions for this tutorial, make sure that your system satisfies the following prerequisites:

- You have already installed a MySQL database server (following the instructions in the [MySQL Installation Guide](#), including the post installation set-up and testing).
- The MySQL database server is already running.
- You have root access to the MySQL database server (that is, you have access to the **root** user account in MySQL, which you can use to administer the database).
- You have access to the Internet (so that you can install the MySQL JDBC driver bundle and the Apache Commons data source bundle, both of which must be downloaded from the Maven Central repository).

Steps to configure JDBC persistence with MySQL

To configure a standalone JBoss A-MQ broker to use JDBC persistence with MySQL, perform the following steps:

1. Log into the MySQL database using the **mysql** client shell. Enter the following command to log on as the **root** user:

```
mysql -u root -p
```

You will be prompted to enter the **root** user password (alternatively, if the root user has no associated password, you can omit the **-p** option).

2. Add the new user account, **amq**, to MySQL with password, **amqPass**, by entering the following command at the **mysql** shell prompt:

```
mysql> CREATE USER 'amq'@'localhost' IDENTIFIED BY 'amqPass';
```

If you would rather create the **amq** user without any password, you can omit the **IDENTIFIED BY** clause, as follows:

```
mysql> CREATE USER 'amq'@'localhost';
```



NOTE

This example assumes you are invoking the **mysql** shell from the same host where the MySQL database server is running. If you are logging on to the MySQL database remotely, however, you should replace **localhost** in the preceding command (and subsequent commands) by the name of the host where you are invoking the **mysql** shell.

- Grant privileges to the **amq** user, enabling it to access the **activemq** database instance (which has yet to be created). Enter the following **GRANT** command at the **mysql** shell prompt:

```
mysql> GRANT ALL PRIVILEGES ON activemq.* TO 'amq'@'localhost' WITH GRANT OPTION;
```

- Create the **activemq** database instance, by entering the following command:

```
mysql> CREATE DATABASE activemq;
```

There is no need to create any database tables at this point. The broker's JDBC persistence will automatically create the necessary tables when it starts up for the first time.

- Start the JBoss A-MQ standalone container, with its default (unchanged) configuration:

```
cd InstallDir/bin
./amq
```

- Install the MySQL JDBC driver into the container, as follows:

```
JBossA-MQ:karaf@root> osgi:install mvn:mysql/mysql-connector-java/5.1.27
```

- Install the Apache Commons data source bundle, as follows:

```
JBossA-MQ:karaf@root> osgi:install
mvn:org.apache.servicemix.bundles/org.apache.servicemix.bundles.commons-dbc/1.4_3
```

- Stop the JBoss A-MQ container (for example, by entering the **shutdown** command at the console). Now configure the broker to use JDBC persistence by editing the **InstallDir/etc/activemq.xml** file. Modify the **broker/persistenceAdapter** element and add a new **bean** element (for the MySQL data source) as follows:

```
<beans ...>
...
<bean id="mysql-ds"
  class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://localhost/activemq?
relaxAutoCommit=true"/>
  <property name="username" value="amq"/>
  <property name="password" value="amqPass"/> <property
```

```

name="poolPreparedStatements" value="true"/> </bean>

  <broker ...>
    ...
    <persistenceAdapter>
<jdbcPersistenceAdapter dataSource="#mysql-ds"/>
    </persistenceAdapter>
    ...
  </broker>

</beans>

```

Where the bean with the ID, **mysql-ds**, creates a data source instance for connecting to the MySQL database through the JDBC protocol. Note particularly the following property settings for this bean:

url

Is used to specify a JDBC URL in the following format:

```
jdbc:mysql://Hostame/DBName[?Property=Value]
```

Where **Hostname** is the host where the MySQL database server is running; **DBName** is the name of the database instance used to store the broker data (which is **activemq**, in this example); and you can optionally set property values, **Property=Value**, after the **?** character.

password

If you specified a password for the **amq** user when you created it, specify the password here. Otherwise, if no password was defined, specify a blank string, "".

- Restart the JBoss A-MQ container, as follows:

```
./amq
```

As the broker initializes, it automatically creates new tables in the **activemq** database instance to hold the broker data (this is the default behavior).

- To verify that the requisite tables have been created in the **activemq** database instance, enter the following commands at the **mysql** client shell:

```

mysql> USE activemq;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> SHOW TABLES;
+-----+
| Tables_in_activemq |
+-----+
| ACTIVEMQ_ACKS      |
| ACTIVEMQ_LOCK      |
| ACTIVEMQ_MSGS      |
+-----+
3 rows in set (0.00 sec)

```

```
mysql> describe ACTIVEMQ_LOCK;
+-----+-----+-----+-----+-----+-----+
| Field      | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| ID         | bigint(20) | NO   | PRI | NULL    |      |
| TIME      | bigint(20) | YES  |     | NULL    |      |
| BROKER_NAME | varchar(250) | YES  |     | NULL    |      |
+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

mysql> describe ACTIVEMQ_MSGS
-> ;
+-----+-----+-----+-----+-----+-----+
| Field      | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| ID         | bigint(20) | NO   | PRI | NULL    |      |
| CONTAINER | varchar(250) | YES  | MUL | NULL    |      |
| MSGID_PROD | varchar(250) | YES  | MUL | NULL    |      |
| MSGID_SEQ  | bigint(20) | YES  |     | NULL    |      |
| EXPIRATION | bigint(20) | YES  | MUL | NULL    |      |
| MSG       | longblob  | YES  |     | NULL    |      |
| PRIORITY  | bigint(20) | YES  | MUL | NULL    |      |
| XID       | varchar(250) | YES  | MUL | NULL    |      |
+-----+-----+-----+-----+-----+-----+
8 rows in set (0.00 sec)
```

6.6. TUTORIAL: CONFIGURING ACTIVEMQ JDBC PERSISTENCE ON FABRIC CONTAINER WITH POSTGRESQL

Overview

This tutorial provides instructions for configuring ActiveMQ JDBC Persistence on Fabric Container using PostgreSQL database to store the broker's data.

This tutorial assumes that the PostgreSQL is provided as a JAR file on the local filesystem of each broker and referenced using a file:URI. If a filesystem location is used, then the location needs to be the same on each broker instance, or the configuration becomes uncoordinated.

The brokers must have fixed, known port numbers. In this example, a single broker is used which has a fixed port number. This port number is provided in a child profile. In case of multiple brokers, the brokers in the group could have the same configuration but different port numbers. You can create additional child profiles with different port numbers.

Prerequisites

Before following the instructions for this tutorial, make sure that your system satisfies the following prerequisites:

- You have already installed a PostgreSQL database server.
- The PostgreSQL database server is already running.

- You have root access to the PostgreSQL database server (that is, you have access to the root user account in PostgreSQL, which you can use to administer the database).
- You have access to the internet so that you can install the PostgreSQL JDBC driver bundle and the Apache Commons data source bundle, both of which must be downloaded from the Maven Central repository.

Steps to configure JDBC persistence with PostgreSQL

To configure Red Hat JBoss A-MQ broker to use JDBC persistence with PostgreSQL on a fabric container, perform following steps:

Procedure 6.1. Configure PostgreSQL database

1. Log into the PostgreSQL database. Enter the following command to log on as the root user:

```
su - postgres
psql
```

2. Add the new user account **amq** to PostgreSQL with password **amqPass**, by entering the following command:

```
postgres=# create ROLE amq LOGIN PASSWORD 'amqPass' SUPERUSER;
CREATE ROLE
```

3. Create the **activemq** database instance, by entering the following command:

```
postgres=# CREATE DATABASE activemq WITH OWNER = amq;
CREATE DATABASE
```

4. Grant privileges to the **amq** user, enabling it to access the activemq database instance (which has yet to be created). Enter the following **GRANT** command at the postgresql shell prompt:

```
postgres=# GRANT CONNECT ON DATABASE activemq TO amq;
GRANT
```

There is no need to create any database tables at this point. The broker's JDBC persistence will automatically create the necessary tables when it starts up for the first time.

5. Stop and restart the PostgreSQL server.

```
# service postgresql stop

Stopping postgresql service: [ OK ]

# service postgresql start

Starting postgresql service:
```

Procedure 6.2. Create A-MQ Broker Group

1. In JBoss A-MQ, start the local container as follows:

■

```
cd InstallDir/bin
./amq
```

2. Create a broker group in a fabric container as follows:

```
JBossA-MQ:karaf@root> fabric:create --clean --wait-for-provisioning
JBossA-MQ:karaf@root> fabric:mq-create --kind MasterSlave --group BrokerGroup
AMQBroker
```

3. Install the **PostgreSQL JDBC** driver into the container, as follows:

```
JBossA-MQ:karaf@root> profile-edit --bundle wrap:file:/// $DRIVER_PATH/postgresql-9.4-
1206-jdbc4.jar mq-broker-BrokerGroup.AMQBroker
```

4. Install the **Apache Commons** data source bundle and **osgi.compendium** bundle as follows:

```
JBossA-MQ:karaf@root> profile-edit --bundle
mvn:org.apache.servicemix.bundles/org.apache.servicemix.bundles.commons-dbc4/1.4_3
mq-broker-BrokerGroup.AMQBroker
JBossA-MQ:karaf@root> profile-edit --bundle mvn:org.osgi/org.osgi.compendium/4.3.1 mq-
broker-BrokerGroup.AMQBroker
```

5. Create a child container:

```
JBossA-MQ:karaf@root> fabric:container-create-child --profile mq-broker-
BrokerGroup.AMQBroker root broker-container1
```

6. Enter following command to view the newly created container.

```
JBossA-MQ:karaf@root> container-list
```

7. Stop the JBoss A-MQ container by entering following command at the console:

```
JBossA-MQ:karaf@root> container-stop broker-container1
```

Procedure 6.3. Create a custom JDBC broker configuration

When you use **mq-create** command, it creates a fabric profile which can be assigned to a container to instantiate an A-MQ message broker in that container. However, that profile does not have its own broker configuration file (for example, **activemq.xml/broker.xml**), but inherits it from the parent profile. Since this is the default configuration file, it assumes **KahaDB** persistence and also defaults to other settings. Hence it is necessary to create and configure a customized broker, without affecting the fabric defaults. For this, you need to add a customized configuration XML file to this profile and instruct fabric to use it. You can achieve this using **Hawtio Web console**.

1. Log in to Hawtio console and navigate to the mq-base profile in the console. Open the **broker.xml** and copy its contents to a text editor.
2. Navigate to newly created profile **mq-broker-BrokerGroup.AMQBroker** and click the **+ Create** button in the top right hand corner.

3. Select XML document option. Enter the name of the file as **jdbc-broker.xml** and click Create. An empty xml document opens. Paste the content copied in the step 1. An example code is given below.

```

<beans
  xmlns="http://www.springframework.org/schema/beans"
  xmlns:amq="http://activemq.apache.org/schema/core"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://activemq.apache.org/schema/core http://activemq.apache.org/schema/core/activemq-
core.xsd">

  <!-- Allows us to use system properties and fabric as variables in this configuration file -->
  <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="properties">
      <bean class="io.fabric8.mq.fabric.ConfigurationProperties"/>
    </property>
  </bean>

  <bean id="postgres-ds" class="org.postgresql.ds.PGSimpleDataSource">
    <property name="url" value="jdbc:postgresql://localhost/activemq"/>
    <property name="user" value="amq"/>
    <property name="password" value="amqPass"/>
    <property name="initialConnections" value="1"/>
    <property name="maxConnections" value="10"/>
  </bean>

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

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

    <managementContext>
      <managementContext createConnector="false"/>
    </managementContext>

    <persistenceAdapter>
      <jdbcPersistenceAdapter dataSource="#postgres-ds" lockKeepAlivePeriod="5000">
        <locker>
          <lease-database-locker lockAcquireSleepInterval="10000"/>
        </locker>
      </jdbcPersistenceAdapter>
    </persistenceAdapter>

```

```

<plugins>
  <jaasAuthenticationPlugin configuration="karaf" />
  <authorizationPlugin>
    <map>
      <authorizationMap
groupClass="org.apache.karaf.jaas.boot.principal.RolePrincipal">
        <!--
manager,viewer,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin,User -
-->
          <authorizationEntries>
            <authorizationEntry queue="">
read="manager,viewer,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin"
write="manager,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin"
admin="manager,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin"/>
            <authorizationEntry topic="">
read="manager,viewer,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin"
write="manager,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin"
admin="manager,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin"/>
            <authorizationEntry topic="ActiveMQ.Advisory.>"
read="manager,viewer,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin,
User"
write="manager,viewer,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin,
User"
admin="manager,viewer,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin
,User" />
          </authorizationEntries>
            <tempDestinationAuthorizationEntry>
              <tempDestinationAuthorizationEntry
read="manager,viewer,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin"
write="manager,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin"
admin="manager,Operator,Maintainer,Deployer,Auditor,Administrator,SuperUser,admin"/>
              </tempDestinationAuthorizationEntry>
            </authorizationMap>
          </map>
        </authorizationPlugin>
      </plugins>

<systemUsage>
  <systemUsage>
    <memoryUsage>
      <memoryUsage percentOfJvmHeap="70"/>
    </memoryUsage>
    <storeUsage>
      <storeUsage limit="100 gb"/>
    </storeUsage>
    <tempUsage>
      <tempUsage limit="50 gb"/>
    </tempUsage>
  </systemUsage>
</systemUsage>

<transportConnectors>
  <transportConnector name="openwire" uri="tcp://${bindAddress}:${bindPort}"/>
</transportConnectors>

```



```

</broker>

<beans>

```

4. Edit the broker/persistenceAdapter element in the **jdbc-broker.xml** as follows:

```

<persistenceAdapter>
  <jdbcPersistenceAdapter dataSource="#postgres-ds" lockKeepAlivePeriod="5000">
    <locker>
      <lease-database-locker lockAcquireSleepInterval="10000"/>
    </locker>
  </jdbcPersistenceAdapter>
</persistenceAdapter>

```

5. Add a new bean element for the PostgreSQL data source as follows:

```

<beans...>

  <bean id="postgres-ds" class="org.postgresql.ds.PGSimpleDataSource">
    <property name="url" value="jdbc:postgresql://localhost/activemq"/>
    <property name="user" value="amq"/>
    <property name="password" value="amqPass"/>
    <property name="initialConnections" value="1"/>
    <property name="maxConnections" value="10"/>
  </bean>

</beans>

```

6. Click Save to save the newly created **jdbc-broker.xml** document.
7. Navigate to **mq-broker-BrokerGroup.AMQBroker** broker profile. Edit the configuration properties file **io.fabric8.mq.fabric.server-broker.properties** to assign newly created **jdbc-broker.xml** file as shown below.

```

config = profile:jdbc-broker.xml

```

8. Click Save.

Procedure 6.4. Verify Broker configuration and creation of data tables

1. Restart the JBoss A-MQ container, as follows:

```

container-start broker-container1

```

2. To verify that the requisite tables have been created in the activemq database instance, enter the following commands at the postgres client shell:

```

$ su - postgres

Password:

-bash-4.1$ psql

```

```
# connect to activemq schema

postgres=# \c activemq

# check tables

activemq=# \dt

List of relations

Schema | Name | Type | Owner
-----+-----+-----+-----
public | activemq_acks | table | activemq
public | activemq_lock | table | activemq
public | activemq_msgs | table | activemq
(3 rows)

# run select query.

activemq=# select * from activemq_lock;

id | time | broker_name
---+-----+-----
1 | 1506947282760 | broker-container1
```

CHAPTER 7. MESSAGE CURSORS

Abstract

Red Hat JBoss A-MQ uses message cursors to improve the scalability of the persistent message store. By default, a hybrid approach that uses an in memory dispatch queue for fast consumers and message cursors for slower consumers is used. JBoss A-MQ also supports two alternative cursor implementations. The type of cursor can be configured on a per-destination basis.

Message data is cached in the broker using *message cursors*, where a cursor instance is associated with each destination. A message cursor represents a batch of messages cached in memory. When necessary, a message cursor will retrieve persisted messages through the persistence adapter. But the key point you need to understand about message cursors is that the cursors are essentially *independent* of the persistence layer.

Message cursors provide a means for optimizing a persistent message store. They allow the persistent store to maintain a pointer to the next batch of messages to pull from the persistent message store. Red Hat JBoss A-MQ has three types of cursors that can be used depending on the needs of your application:

- **Store-based** cursors are used by default to handle persistent messages.
- **VM** cursors are very fast, but cannot handle slow message consumers.
- **File-based** cursors are used by default to handle non-persistent messages. They are useful when the message store is slow and message consumers are relatively fast.

7.1. TYPES OF CURSORS

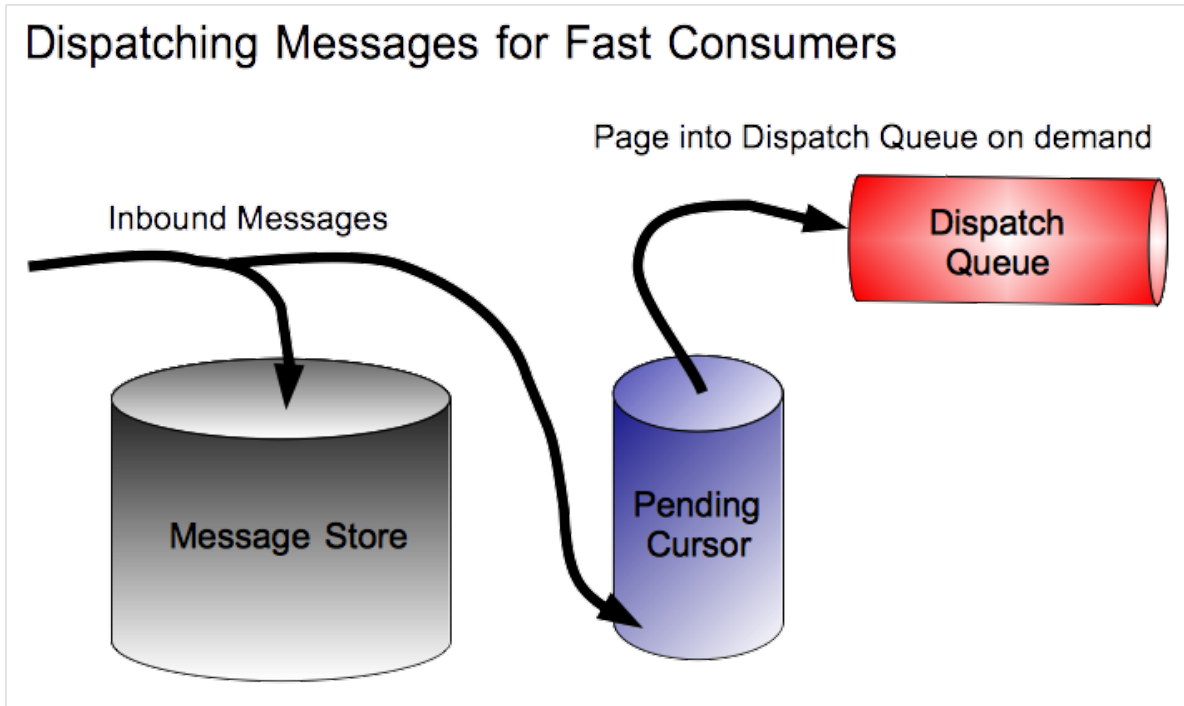
Store-based cursors

Store-based cursors are used, by default, for processing persistent messages. Store-based cursors are a hybrid implementation that offers the robustness of typical cursor implementations and the speed of in-memory message reference implementations.

Typically messaging systems will pull persistent messages from long-term storage in a batch when a client is ready to consume them. A cursor will be used to maintain the position for the next batch of messages. While this approach scales well and provides excellent robustness, it does not perform well when message consumers keep pace with message producers.

As shown in [Figure 7.1, "Store-based Cursors for a Fast Consumer"](#), the store-based cursor addresses the fast consumer case just like the VM cursor. Messages are written to the persistent store and are also directly stored in the pending cursor, which is held completely in memory. The pending cursor then feeds the messages into the dispatch queue. However, since the store cursor can hold only a limited number of messages in memory, it is a mapping of only a fraction of the persistent message store.

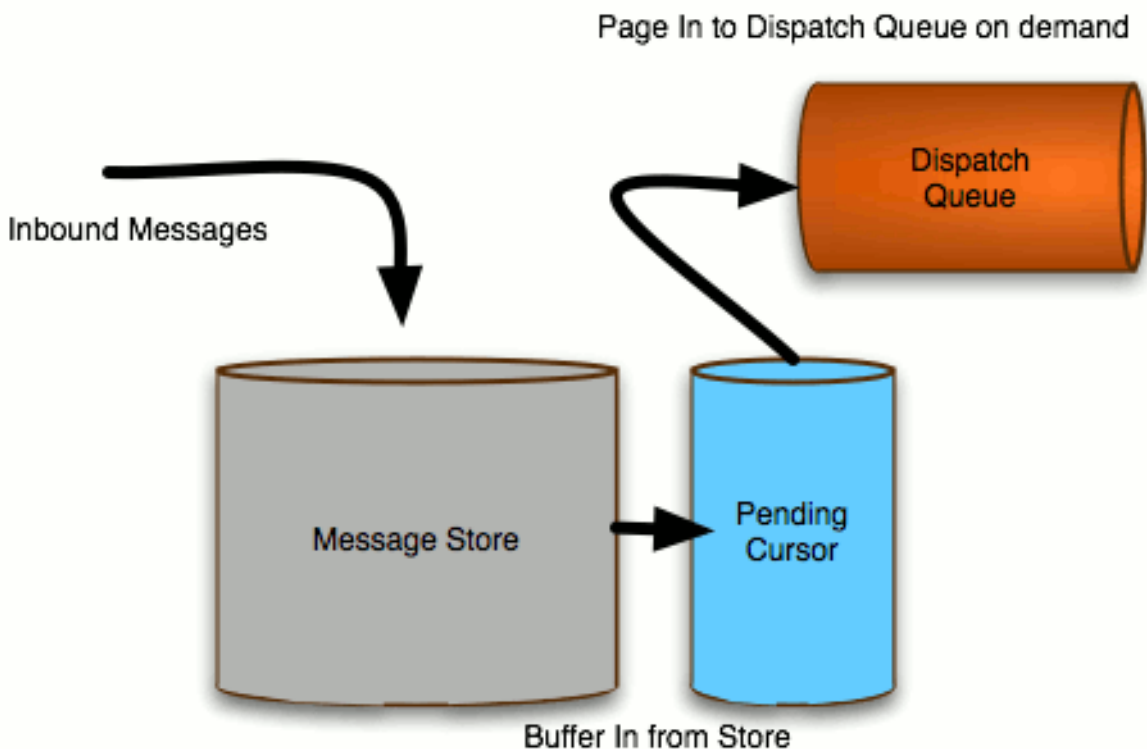
Figure 7.1. Store-based Cursors for a Fast Consumer



When a consumer starts with a back log of messages or falls behind its message producers, JBoss A-MQ changes the strategy used to dispatch messages. As shown in [Figure 7.2, "Store-based Cursors for a Slow Consumer"](#), messages are held in the message store and fed into the consumer's dispatch queue using the pending cursor.

Figure 7.2. Store-based Cursors for a Slow Consumer

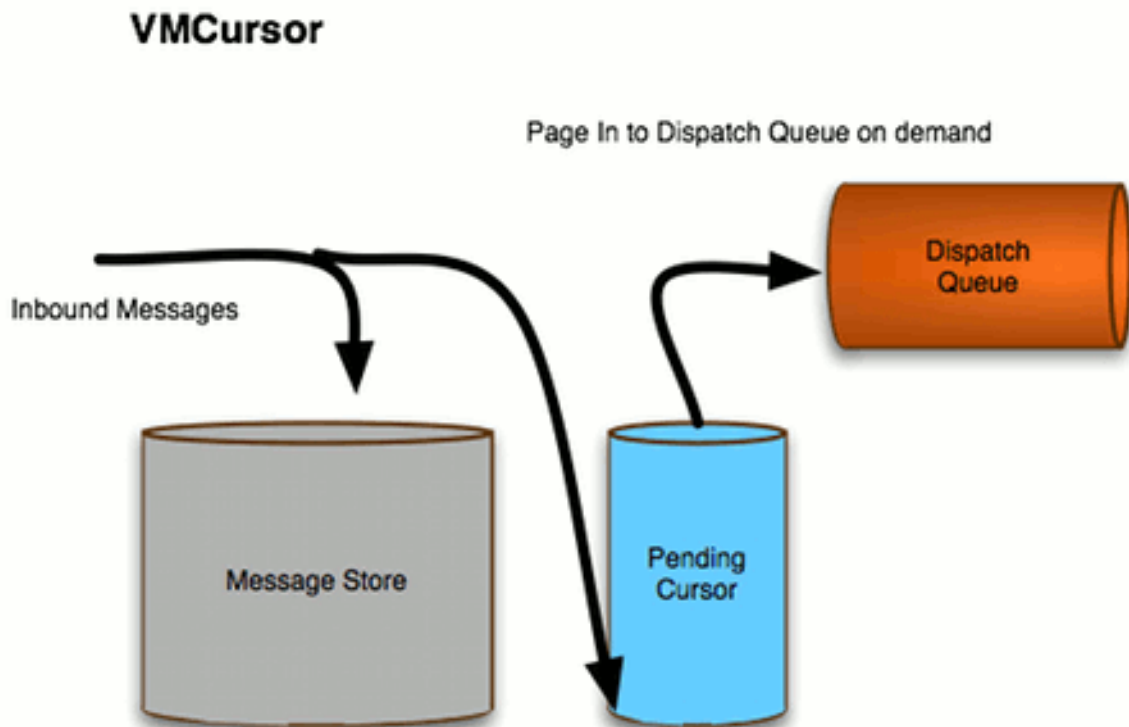
Dispatching Messages if Dispatch Queue is Full



VM cursors

When speed is the top priority and the consumers can definitely keep pace with the message producers, VM cursors could be the best approach. In this approach, shown in [Figure 7.3, “VM Cursors”](#), messages are written to the persistent store and are also stored in the pending cursor, which is held completely in memory. The messages are fed into the dispatch queue from the pending cursor. Since it needs to hold all messages in memory, the VM cursor is a snapshot of the entire persistent message store.

Figure 7.3. VM Cursors

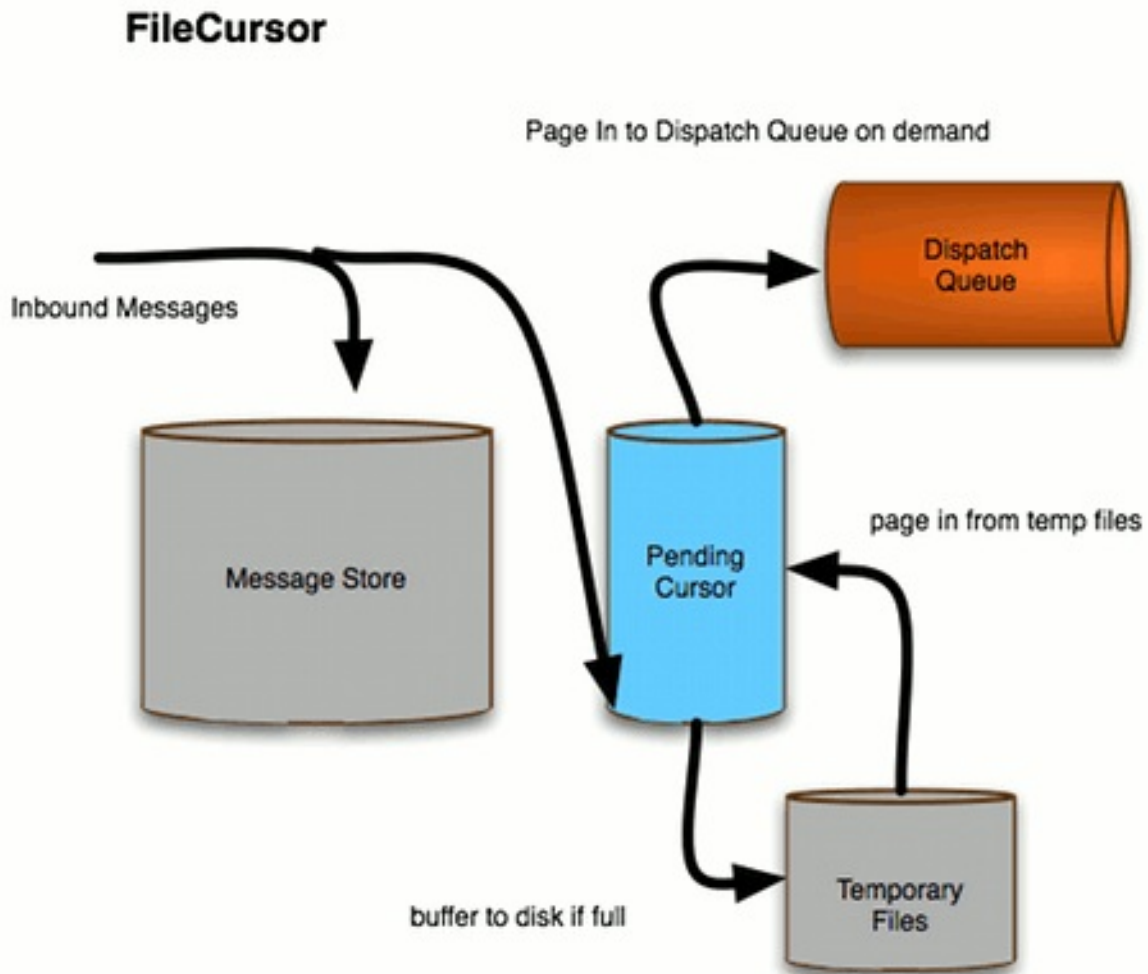


Because the messages are dispatched from active memory when using VM cursors, this method is exceptionally fast. However, if the number of unconsumed messages grows large, the producers are throttled to avoid exceeding available memory.

File-based cursors

File-based cursors are a variation of VM cursors that provides a buffer against running out of memory when a consumer falls behind. As shown in [Figure 7.4, “File-based Cursors”](#), the broker pages messages out to a temporary file when the broker’s memory limit is reached.

Figure 7.4. File-based Cursors



Using a temporary file cushions the broker against situations where a consumer occasionally falls behind or messages are produced in a burst. The broker uses the temporary file instead of resorting to using slower persistent storage.

File-based cursors do not scale well when consumers are frequently behind by a large margin. It is also not ideal when a fast long term message store is available.

File-based cursors are used, by default, to process non-persistent messages.

7.2. CONFIGURING THE TYPE OF CURSOR USED BY A DESTINATION

Overview

By default, JBoss A-MQ uses store-based cursors for persistent messages and file-based cursors for non-persistent messages. You can, however, configure your destinations to use a specified cursor implementation by adding the appropriate policy entries into the destination's policy map.

You configure a destination's policy set using a **destinationPolicy** element. The **destinationPolicy** element is a wrapper for a **policyMap** element. The **policyMap** element is a wrapper for a **policyEntries** element. The **policyEntries** element is a wrapper for one or more **policyEntry** elements.

The cursor policies are entered as children to a **policyEntry** element. The configuration elements used to specify the type of destination you are configuring. Topics use cursors for both durable subscribers and transient subscribers, so it uses two sets of configuration elements. Queues only a single cursor and

only require a single set of configuration elements.

Configuring topics

Topics maintain a dispatch queue and a pending cursor for every consumer subscribed to the topic regardless of whether the subscription is durable or transient. You can configure the cursor implementation used by durable subscribers separately from the cursor implementation used by transient subscribers.

You configure the cursor implementation used by durable subscribers by adding **PendingDurableSubscriberMessageStoragePolicy** child element to the topic's **policyEntry** element. [Table 7.1, "Elements for Configuring the Type of Cursor to Use for Durable Subscribers"](#) describes the possible children of **PendingDurableSubscriberMessageStoragePolicy**.

Table 7.1. Elements for Configuring the Type of Cursor to Use for Durable Subscribers

Element	Description
storeDurableSubscriberCursor	Specifies that store-based cursors will be used. See the section called "Store-based cursors" for more information.
vmDurableCursor	Specifies that VM cursors will be used. See the section called "VM cursors" for more information.
fileDurableSubscriberCursor	Specifies that file-based cursors will be used—only suitable for non-persistent messages. See the section called "File-based cursors" for more information.

You configure the cursor implementation used by transient subscribers by adding **pendingSubscriberPolicy** child element to the topic's **policyEntry** element. [Table 7.2, "Elements for Configuring the Type of Cursor to Use for Transient Subscribers"](#) describes the possible children of **pendingSubscriberPolicy**.

Table 7.2. Elements for Configuring the Type of Cursor to Use for Transient Subscribers

Element	Description
<i>Unspecified</i>	Default policy is to use store-based cursors. See the section called "Store-based cursors" for more information.
vmCursor	Specifies the VM cursors will be used. See the section called "VM cursors" for more information.
fileCursor	Specifies that file-based cursors will be used. See the section called "File-based cursors" for more information.

[Example 7.1, "Configuring a Topic's Cursor Usage"](#) shows a configuration snippet that configures a topic to use VM cursors for its transient subscribers and file-based cursors for its durable subscribers.

Example 7.1. Configuring a Topic's Cursor Usage

```

<beans ... >
  <broker ... >
    ...
    <destinationPolicy>
      <policyMap>
        <policyEntries>
          <policyEntry topic="com.fusesource.>" >
            ...
            <pendingSubscriberPolicy>
              <vmCursor />
            </pendingSubscriberPolicy>
            <pendingDurableSubscriberPolicy>
              <storeDurableSubscriberCursor />
            </pendingDurableSubscriberPolicy>
            ...
          </policyEntry>
          ...
        </policyEntries>
      </policyMap>
    </destinationPolicy>
    ...
  </broker>
  ...
</beans>

```

Configuring queues

Queues use a single pending cursor and dispatch queue. You configure the type of cursor to use by adding a **pendingQueuePolicy** element to the queue's **policyEntry** element. [Table 7.3, "Elements for Configuring the Type of Cursor to Use for a Queue"](#) describes the possible children elements of the **pendingQueuePolicy** element.

Table 7.3. Elements for Configuring the Type of Cursor to Use for a Queue

Element	Description
storeCursor	Specifies that store-based cursors will be used. See the section called "Store-based cursors" for more information.
vmQueueCursor	Specifies the VM cursors will be used. See the section called "VM cursors" for more information.
fileQueueCursor	Specifies that file-based cursors will be used. See the section called "File-based cursors" for more information.

[Example 7.2, "Configuring a Queue's Cursor Usage"](#) shows a configuration snippet that configures a queue to use VM cursors.

-

Example 7.2. Configuring a Queue's Cursor Usage

```
<beans ... >
  <broker ... >
    ...
    <destinationPolicy>
      <policyMap>
        <policyEntries>
          <policyEntry queue="com.fusesource.>" >
            ...
            <pendingQueuePolicy>
              <vmQueueCursor />
            </pendingQueuePolicy>
            ...
          </policyEntry>
          ...
        </policyEntries>
      </policyMap>
    </destinationPolicy>
    ...
  </broker>
  ...
</beans>
```

CHAPTER 8. MESSAGE STORE LOCKERS

Abstract

Message store locks are used to elect the master broker in master/slave groups. They are also useful for ensuring that multiple brokers are not attempting to share the same message store. Red Hat JBoss A-MQ's lockers are configurable to allow for tuning.

8.1. LOCKER BASICS

Overview

Red Hat JBoss A-MQ provides two default lockers that are used based on the type of message store being used:

- shared file locker—used by KahaDB and LevelDB stores
- database locker—used by the JDBC store



NOTE

JBoss A-MQ also provides a leased database locker that can be in cases where the brokers may periodically lose their connection to the message store.

These default lockers are configurable to optimize their performance.

For further optimization, you can implement your own locker and plug it into the message store. Doing so involves implementing a simple Java interface and adding some configuration to the persistence adapter.

Message store locks are primarily leveraged by the broker for electing masters in master/slave configurations. For more information on master/slave groups see [chapter "Master/Slave" in "Fault Tolerant Messaging"](#).

Configuring a persistence adapter's locker

To configure the locker used by a persistence adapter you add a **locker** element as a child to the adapter's configuration element as shown in [Example 8.1, "Configuring a Message Store Locker"](#).

Example 8.1. Configuring a Message Store Locker

```
<persistenceAdapter>
  <kahaDB directory = "target/activemq-data">
    <locker>
      ...
    </locker>
  </kahaDB>
</persistenceAdapter>
```

Standard locker configuration properties

All locker implementations are required to have the two common configuration properties described in Table 8.1, “Common Locker Properties”.

Table 8.1. Common Locker Properties

Property	Default Value	Description
lockAcquireSleepInterval	1000	Specifies the delay interval, in milliseconds, between attempts to acquire a lock.
failIfLocked	false	Specifies in the broker should immediately fail if a lock cannot be obtained.

The properties are specified as attributes to the locker's XML configuration element.

8.2. USING THE PROVIDED LOCKERS

Red Hat JBoss A-MQ includes three standard locker implementations:

- [shared file locker](#)—used by file-based message stores like KahaDB and LevelDB
- [database locker](#)—used as the default for JDBC message stores
- [lease database locker](#)—used as an alternative locker for JDBC message stores in scenarios where brokers have inconsistent connections to the message store

8.2.1. Shared File Locker

Overview

The shared file locker is used by file-based message stores to ensure that only one broker can modify the files used by the message store.

Configuration

As shown in [Example 8.2, “Configuring a Shared File Locker”](#), the shared file locker is configured using the **shared-file-locker** element.

Example 8.2. Configuring a Shared File Locker

```
<persistenceAdapter>
  <kahaDB directory = "target/activemq-data" lockKeepAlivePeriod="2000">
    <locker>
      <shared-file-locker lockAcquireSleepInterval="10000"/>
    </locker>
  </kahaDB>
</persistenceAdapter>
```

The shared file locker supports the common configuration properties described in [Table 8.1, "Common Locker Properties"](#).

8.2.2. Database Locker

Overview

The database locker is the default locker for all JDBC persistence adapters. It locks a database table in a transaction to ensure that only one broker can modify the message store.

The database locker does not perform well in two scenarios:

- intermittent database connectivity
- database failover

Configuration

As shown in [Example 8.3, "Configuring a Database Locker"](#), it is configured using the **database-locker** element.

Example 8.3. Configuring a Database Locker

```
<persistenceAdapter>
  <jdbcPersistenceAdapter dataDirectory="${activemq.data}" dataSource="#mysql-ds"
    lockKeepAlivePeriod="2000">
    <locker>
      <database-locker lockAcquireSleepInterval="10000"/>
    </locker>
  </jdbcPersistenceAdapter>
</persistenceAdapter>
```

The database locker supports the common configuration properties described in [Table 8.1, "Common Locker Properties"](#).

Intermittent database connectivity

When the master broker loses its connection to the database, or crashes unexpectedly, the information about the lock remains in the database until the database responds to the half-closed socket connection via a TCP timeout. This can prevent the slave from starting for a period of time.

Database failover

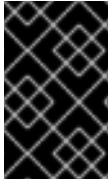
When the database used for the message store supports failover issues can arise. When the database connection is dropped in the event of a replica failover, the brokers see this as a database failure and all of the brokers in the master/slave group will begin competing for the lock. This restarts the master election process and can cause the group to failover to a new master. For more information see [section "Shared JDBC Master/Slave" in "Fault Tolerant Messaging"](#).

8.2.3. Lease Database Locker

Overview

The lease database locker is designed to overcome the shortcomings of the default database locker by forcing the lock holder to periodically renew the lock. When the lock is first acquired the broker holds it for the period specified in the persistence adapter's **lockKeepAlivePeriod** attribute. After the initial period, the lock is renewed for the period specified by the locker's **lockAcquireSleepInterval** attribute.

When all of broker's system clocks are properly synchronized, the master broker will always renew the lease before any of the slaves in the group can steal it. In the event of a master's failure, the lock will automatically expire within the configured amount of time and one of the slave's in the group will be able to acquire it.



IMPORTANT

For this to work correctly, give the **leaseHolderId** attribute, found on the **lease-database-locker** tag, a unique value. This unique value is used to create a lease lock definition.

Configuration

As shown in [Example 8.4, "Configuring a Lease Database Locker"](#), it is configured using the **lease-database-locker** element.

Example 8.4. Configuring a Lease Database Locker

```
<ioExceptionHandler>
  <jDBCIOExceptionHandler/>
</ioExceptionHandler>

<persistenceAdapter>
  <jdbcPersistenceAdapter dataDirectory="{activemq.data}" dataSource="#mysql-ds"
  lockKeepAlivePeriod="2000">
    <locker>
      <lease-database-locker lockAcquireSleepInterval="10000" leaseHolderId="broker1"/>
    </locker>
  </jdbcPersistenceAdapter>
</persistenceAdapter>
```

The lease database locker supports the common configuration properties described in [Table 8.1, "Common Locker Properties"](#).

Default IOException Handler

In order to cope with reconnection to the JDBC database, it is essential to enable a **jDBCIOExceptionHandler** exception handler to the configuration. The **jDBCIOExceptionHandler** will pause and resume the transport connectors on any I/O exception related to database access.

In ActiveMQ 5.11 **jDBCIOExceptionHandler** is deprecated, however in Red Hat JBoss A-MQ 6.3 it is still recommended and is not deprecated. The **LeaseLockerIOExceptionHandler** exception handler is supported as well and it works with any persistence adapter that supports pluggable storage lockers whether or not a locker is in use.

Dealing with unsynchronized system clocks

The lease database locker relies on each broker's system clock to ensure the proper timing of lease

expiration and lock requests. When all of the system clocks are synchronized, the timing works. Once the system clocks start drifting apart, the timing can be thrown off and a slave broker could possibly steal the lock from the group's master.

To avoid this problem the locker can make adjustments based on the database server's current time setting. This feature is controlled by setting the locker's **maxAllowableDiffFromDBTime** to specify the number of milliseconds by which a broker's system clock can differ from the database's before the locker automatically adds an adjustment. The default setting is zero which deactivates the adjustments.

[Example 8.5, "Configuring a Lease Database Locker to Adjust for Non-synchronized System Clocks"](#) shows configuration for making adjustments when a broker's clock differs from the database by one second.

Example 8.5. Configuring a Lease Database Locker to Adjust for Non-synchronized System Clocks

```
<ioExceptionHandler>
  <jdbcIOExceptionHandler/>
</ioExceptionHandler>

<persistenceAdapter>
  <jdbcPersistenceAdapter ... >
    <locker>
      <lease-database-locker maxAllowableDiffFromDBTime="1000"/>
    </locker>
  </jdbcPersistenceAdapter>
</persistenceAdapter>
```

8.3. USING CUSTOM LOCKERS

Overview

If one of the provided lockers are not sufficient for your needs, you can implement a custom locker. All lockers are implementations of the Red Hat JBoss A-MQ **Locker** interface. They are attached to the persistence adapter as a spring bean in the **locker** element.

Interface

All lockers are implementations of the **org.apache.activemq.broker.Locker** interface. Implementing the **Locker** interface involves implementing seven methods:

- **boolean keepAlive()**
throws IOException;
Used by the lock's timer to ensure that the lock is still active. If this returns false, the broker is shutdown.
- **void setLockAcquireSleepInterval(long lockAcquireSleepInterval);**
Sets the delay, in milliseconds, between attempts to acquire the lock.
lockAcquireSleepInterval is typically supplied through the locker's XML configuration.
- **public void setName(String name);**
Sets the name of the lock.

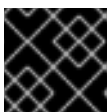
- **public void setFailIfLocked(boolean failIfLocked);**
Sets the property that determines if the broker should fail if it cannot acquire the lock at start-up. **failIfLocked** is typically supplied through the locker's XML configuration.
- **public void configure(PersistenceAdapter persistenceAdapter) throws IOException;**
Allows the locker to access the persistence adapter's configuration. This can be used to obtain the location of the message store.
- **void start();**
Executed when the locker is initialized by the broker. This is where the bulk of the locker's implementation logic should be placed.
- **void stop();**
Executed when the broker is shutting down. This method is useful for cleaning up any resources and ensuring that all of the locks are released before the broker is completely shutdown.

Using AbstractLocker

To simplify the implementation of lockers, Red Hat JBoss A-MQ includes a default locker implementation, **org.apache.activemq.broker.AbstractLocker**, that serves as the base for all of the provided lockers. It is recommended that all custom locker implementations also extend the **AbstractLocker** class instead of implementing the plain **Locker** interface.

AbstractLocker provides default implementations for the following methods:

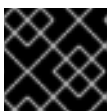
- **keepAlive()**—returns **true**
- **setLockAcquireSleepInterval()**—sets the parameter to the value of the locker beans' *lockAcquireSleepInterval* if provided or to **10000** if the parameter is not provided
- **setName()**
- **setFailIfLocked()**—sets the parameter to the value of the locker beans' *failIfLocked* if provided or to **false** if the parameter is not provided
- **start()**—starts the locker after calling two additional methods



IMPORTANT

This method should not be overridden.

- **stop()**—stops the locker and adds a method that is called before the locker is shutdown and one that is called after the locker is shutdown



IMPORTANT

This method should not be overridden.

AbstractLocker adds two methods that must be implemented:

- **void doStart() throws Exception;**
Executed as the locker is started. This is where most of the locking logic is implemented.

- **void doStop(ServiceStopper stopper)**
throws Exception;

Executed as the locker is stopped. This is where locks are released and resources are cleaned up.

In addition, **AbstractLocker** adds two methods that can be implemented to provide additional set up and clean up:

- **void preStart()**
throws Exception;

Executed before the locker is started. This method can be used to initialize resources needed by the lock. It can also be used to perform any other actions that need to be performed before the locks are created.

- **void doStop(ServiceStopper stopper)**
throws Exception;

Executed after the locker is stopped. This method can be used to clean up any resources that are left over after the locker is stopped.

Configuration

Custom lockers are added to a persistence adapter by adding the bean configuration to the persistence adapter's **locker** element as shown in [Example 8.6, "Adding a Custom Locker to a Persistence Adapter"](#).

Example 8.6. Adding a Custom Locker to a Persistence Adapter

```
<persistenceAdapter>
  <kahaDB directory = "target/activemq-data">
    <locker>
      <bean class="my.custom.LockerImpl">
        <property name="lockAcquireSleepInterval" value="5000" />
        ...
      </bean>
    </locker>
  </kahaDB>
</persistenceAdapter>
```

INDEX

A

AbstractLocker, [Using AbstractLocker](#)

B

broker element, [Activating and deactivating persistence](#)

persistent attribute, [Activating and deactivating persistence](#)

C

configuration

turning persistence on/off, [Activating and deactivating persistence](#)

cursors

file-based, [File-based cursors](#)

store-based, [Store-based cursors](#)

VM, [VM cursors](#)

D

database-locker, [Configuration](#)

destinationPolicy, [Overview](#)

durable subscribers

configuring cursors, [Configuring topics](#)

using file-based cursors, [Configuring topics](#)

using VM cursors, [Configuring topics](#)

F

failIfLocked, [Standard locker configuration properties](#)

fileCursor, [Configuring topics](#)

fileDurableSubscriberCursor, [Configuring topics](#)

fileQueueCursor, [Configuring queues](#)

filteredKahaDB, [Configuration](#)

filteredPersistenceAdapters, [Configuration](#)

J

JDBC

using generic providers, [Using generic JDBC providers](#)

JDBC message store

default locker, [Database Locker](#)

jdbcPersistenceAdapter, [Configuration](#)

adapter attribute, [Configuration](#), [Using generic JDBC providers](#)

cleanupPeriod attribute, [Configuration](#)

createTablesOnStartup attribute, [Configuration](#)

dataDirectory attribute, [Configuration](#)

dataSource attribute, [Configuration](#)

journalled JDBC message store

default locker, [Database Locker](#)

journaledJDBC, [Configuration](#)

adapter attribute, [Configuration](#), [Using generic JDBC providers](#)

createTablesOnStartup attribute, [Configuration](#)

dataDirectory attribute, [Configuration](#)

dataSource attribute, [Configuration](#)

journalArchiveDirectory attribute, [Configuration](#)

journalLogFiles attribute, [Configuration](#)

journalLogFileSize attribute, [Configuration](#)

journalThreadPriority attribute, [Configuration](#)

useJournal attribute, [Configuration](#)

K**kahaDB element, [Basic configuration](#)**

archiveCorruptedIndex attribute, [Configuration attributes](#)

archiveDataLogs attribute, [Configuration attributes](#)

checkForCorruptJournalFiles attribute, [Configuration attributes](#)

checkpointInterval attribute, [Configuration attributes](#)

checksumJournalFiles attribute, [Configuration attributes](#)

cleanupInterval attribute, [Configuration attributes](#)

compactAcksAfterNoGC attribute, [Configuration attributes](#)

compactAcksIgnoresStoreGrowth attribute, [Configuration attributes](#)

concurrentStoreAndDispatchQueues attribute, [Configuration attributes](#)

concurrentStoreAndDispatchTopics attribute, [Configuration attributes](#)

databaseLockedWaitDelay attribute, [Configuration attributes](#)

directory attribute, [Configuration attributes](#)

directoryArchive attribute, [Configuration attributes](#)

enableAckCompaction attribute, [Configuration attributes](#)

enableIndexWriteAsync attribute, [Configuration attributes](#)

enableJournalDiskSyncs attribute, [Configuration attributes](#)

ignoreMissingJournalfiles attribute, [Configuration attributes](#)

indexCacheSize attribute, [Configuration attributes](#)

indexDirectory, [Configuration attributes](#)

indexWriteBatchSize, [Configuration attributes](#)

journalMaxFileLength attribute, [Configuration attributes](#)
maxAsyncJobs attribute, [Configuration attributes](#)
preallocationScope attribute, [Configuration attributes](#)
preallocationStrategy attribute, [Configuration attributes](#)
storeOpenWireVersion attribute, [Configuration attributes](#)

KahaDB message store

architecture, [Architecture](#)
basic configuration, [Basic configuration](#)
configuration attributes, [Configuration attributes](#)
data logs, [Data logs](#)
default locker, [Shared File Locker](#)
metadata cache, [Metadata cache](#)
metadata store, [Metadata store](#)
multi, [Using a Multi KahaDB Persistence Adapter](#)

L

lease-database-locker, [Configuration](#)
levelDB element, [Basic configuration](#)
directory attribute, [Basic configuration](#), [Configuration attributes](#)
failIfLocked attribute, [Configuration attributes](#)
indexBlockRestartInterval attribute, [Configuration attributes](#)
indexBlockSize attribute, [Configuration attributes](#)
indexCacheSize attribute, [Configuration attributes](#)
indexCompression attribute, [Configuration attributes](#)
indexFactory attribute, [Configuration attributes](#)
indexMaxOpenFiles attribute, [Configuration attributes](#)
indexWriteBufferSize attribute, [Configuration attributes](#)
logCompression attribute, [Configuration attributes](#)
logSize attribute, [Configuration attributes](#)
paranoidChecks attribute, [Configuration attributes](#)
readThreads attribute, [Configuration attributes](#)
sync attribute, [Configuration attributes](#)
useLock attribute, [Configuration attributes](#)

verifyChecksums attribute, [Configuration attributes](#)

LevelDB message store

basic configuration, [Basic configuration](#)

configuration attributes, [Configuration attributes](#)

default locker, [Shared File Locker](#)

platform support, [Platform support](#)

lockAcquireSleepInterval, [Standard locker configuration properties](#)

locker, [Configuring a persistence adapter's locker](#), [Configuration](#)

Locker, [Interface](#)

locker configuration, [Standard locker configuration properties](#)

M

maxAllowableDiffFromDBTime, [Dealing with unsynchronized system clocks](#)

message store

locker configuration, [Configuring a persistence adapter's locker](#)

mKahaDB, [Configuration](#)

multi Kahadb persistence adapter

transactions, [Transactions](#)

P

PendingDurableSubscriberMessageStoragePolicy, [Configuring topics](#)

pendingQueuePolicy, [Configuring queues](#)

pendingSubscriberPolicy, [Configuring topics](#)

persistenceAdapter, [Configuring persistence adapter behavior](#)

persistenceFactory, [Configuring persistence adapter behavior](#)

policyEntries, [Overview](#)

policyEntry, [Overview](#)

policyMap, [Overview](#)

R

Replicated LevelDB message store

configuration attributes, [Configuration attributes](#)

replicatedLevelDB element, [Basic configuration](#)

- bind attribute, [Configuration attributes](#)
- directory attribute, [Basic configuration](#)
- hostname attribute, [Configuration attributes](#)
- replicas attribute, [Configuration attributes](#)
- securityToken attribute, [Configuration attributes](#)
- sync attribute, [Configuration attributes](#)
- zkAddress attribute, [Configuration attributes](#)
- zkPassword attribute, [Configuration attributes](#)
- zkPath attribute, [Configuration attributes](#)
- zkSessionTimeout attribute, [Configuration attributes](#)

- replicatedLevelDB message store
 - replicatedLevelDB basic configuration, [Basic configuration](#)

S

- shared-file-locker, [Configuration](#)
- SQL data types, [Customizing the SQL statements used by the adapter](#)
- statements, [Customizing the SQL statements used by the adapter](#)
 - binaryDataType attribute, [Customizing the SQL statements used by the adapter](#)
 - containerNameDataType attribute, [Customizing the SQL statements used by the adapter](#)
 - durableSubAcksTableName attribute, [Customizing the SQL statements used by the adapter](#)
 - lockTableName attribute, [Customizing the SQL statements used by the adapter](#)
 - longDataType attribute, [Customizing the SQL statements used by the adapter](#)
 - messageTableName attribute, [Customizing the SQL statements used by the adapter](#)
 - msgIdDataType attribute, [Customizing the SQL statements used by the adapter](#)
 - sequenceDataType attribute, [Customizing the SQL statements used by the adapter](#)
 - stringIdDataType attribute, [Customizing the SQL statements used by the adapter](#)
 - tablePrefix attribute, [Customizing the SQL statements used by the adapter](#)

T

- transactions
 - multi destination, [Transactions](#)
 - multi Kahadb persistence adapter, [Transactions](#)
 - multiple journals, [Transactions](#)

transient subscribers

configuring cursors, [Configuring topics](#)

using file-based cursors, [Configuring topics](#)

using VM cursors, [Configuring topics](#)

V

vmCursor, [Configuring topics](#)

vmDurableCursor, [Configuring topics](#)

vmQueueCursor, [Configuring queues](#)