Red Hat JBoss Data Grid 6

Administration and Configuration Guide

A guide for configuring and administrating Red Hat JBoss Data Grid 6.

Edition 1

Last Updated: 2017-11-21

Red Hat JBoss Data Grid 6 Administration and Configuration Guide

A guide for configuring and administrating Red Hat JBoss Data Grid 6.
Edition 1

Misha Husnain Ali
Red Hat Engineering Content Services
mhusnain@redhat.com

Darrin Mison

Gemma Sheldon
Red Hat Engineering Content Services
gsheldon@redhat.com

Legal Notice

Abstract

This guide presents information about the administration and configuration of Red Hat JBoss Data Grid.

Table of Contents

PREFACE

CHAPTER 1. JBOSS DATA GRID

1.1. ABOUT JBOSS DATA GRID

JBoss Data Grid is a distributed in-memory data grid, which provides the following capabilities:

- Schemaless key-value store – Red Hat JBoss Data Grid is a NoSQL database that provides the flexibility to store different objects without a fixed data model.
- Grid-based data storage – Red Hat JBoss Data Grid is designed to easily replicate data across multiple nodes.
- Elastic scaling – Adding and removing nodes is achieved simply and is non-disruptive.
- Multiple access protocols – It is easy to access to the data grid using REST, Memcached, Hot Rod, or simple map-like API.

Report a bug

1.2. JBOSS DATA GRID USAGE MODES

JBoss Data Grid offers two usage modes:

Remote Client-Server mode

Remote Client-Server mode provides a managed, distributed and clusterable data grid server. Applications can remotely access the data grid server using Hot Rod, Memcached or REST client APIs.

Library mode

Library mode provides all the binaries required to build and deploy a custom runtime environment. The library usage mode allows local access to a single node in a distributed cluster. This usage mode gives the application access to data grid functionality within a virtual machine in the container being used. Supported containers include Tomcat 7 and JBoss Enterprise Application Platform 6.

Report a bug

1.3. JBOSS DATA GRID BENEFITS

Benefits of JBoss Data Grid

Massive Heap and High Availability

In JBoss Data Grid, applications no longer need to delegate the majority of their data lookup processes to a large single server database for performance benefits. JBoss Data Grid completely removes the bottleneck that exists in the vast majority of current enterprise applications.

> **Example 1.1. Massive Heap and High Availability Example**
>
> In a sample grid with one hundred blade servers, each node has 2 GB storage space dedicated for a replicated cache. In this case, all the data in the grid is copies of the 2 GB data. In contrast, using a distributed grid (assuming the requirement of one copy per data item) the resulting

> memory backed virtual heap contains 100 GB data. This data can now be effectively accessed from anywhere in the grid. In case of a server failure, the grid promptly creates new copies of the lost data and places them on operational servers in the grid.

Scalability

Due to the even distribution of data in JBoss Data Grid, the only upper limit for the size of the grid is the group communication on the network. The network's group communication is minimal and restricted only to the discovery of new nodes. Nodes are permitted by all data access patterns to communicate directly via peer-to-peer connections, facilitating further improved scalability. JBoss Data Grid clusters can be scaled up or down in real time without requiring an infrastructure restart. The result of the real time application of changes in scaling policies results in an exceptionally flexible environment.

Data Distribution

JBoss Data Grid uses consistent hash algorithms to determine the locations for keys in clusters. Benefits associated with consistent hashing include:

- cost effectiveness.
- speed.
- deterministic location of keys with no requirements for further metadata or network traffic.

Data distribution ensures that sufficient copies exist within the cluster to provide durability and fault tolerance, while not an abundance of copies, which would reduce the environment's scalability.

Persistence

JBoss Data Grid exposes a `CacheStore` interface and several high-performance implementations, including the JDBC Cache stores and file system based cache stores. Cache stores can be used to seed the cache and to ensure that the relevant data remains safe from corruption. The cache store also overflows data to the disk when required if a process runs out of memory.

Language bindings

JBoss Data Grid supports both the popular Memcached protocol, with existing clients for a large number of popular programming languages, as well as an optimized JBoss Data Grid specific protocol called Hot Rod. As a result, instead of being restricted to Java, JBoss Data Grid can be used for any major website or application.

Management

In a grid environment of several hundred or more servers, management is an important feature. JBoss Operations Network, the enterprise network management software, is the best tool to manage multiple JBoss Data Grid instances. JBoss Operations Network's features allow easy and effective monitoring of the Cache Manager and cache instances.

Report a bug

1.4. JBOSS DATA GRID PREREQUISITES

JBoss Data Grid requires only a Java 6.0 and above compatible Java Virtual Machine (JVM) to run. An application server is not a requirement for JBoss Data Grid.

Report a bug

1.5. JBOSS DATA GRID VERSION INFORMATION

JBoss Data Grid is based on Infinispan, the open source community version of the data grid software. Infinispan uses code, designs and ideas from JBoss Cache, which has been tried, tested and proved in high stress environments. As a result, JBoss Data Grid's first release is version 6.0 as a result of its deployment history.

Report a bug

1.6. JBOSS DATA GRID CACHE ARCHITECTURE

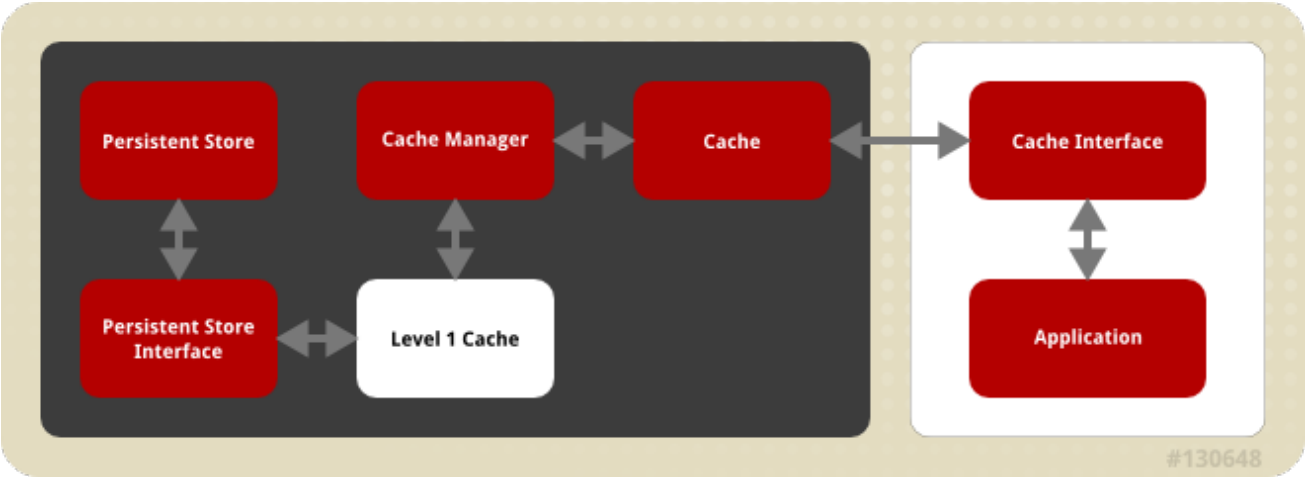


Figure 1.1. JBoss Data Grid Cache Architecture

JBoss Data Grid's cache infrastructure depicts the individual elements and their interaction with each other. For clarity, the cache architecture diagram is separated into two parts:

- Elements that a user cannot directly interact with (depicted within a dark box), which includes the Cache, Cache Manager, Level 1 Cache, Persistent Store Interfaces and the Persistent Store.
- Elements that a user can interact directly with (depicted within a white box), which includes Cache Interfaces and the Application.

Cache Architecture Elements

JBoss Data Grid's cache architecture includes the following elements:

1. The Persistent Store permanently houses cache instances and entries.
2. JBoss Data Grid offers two Persistent Store Interfaces to access the persistent store. Persistent store interfaces can be either:
   - A cache loader is a read only interface that provides a connection to a persistent data store. A cache loader can locate and retrieve data from cache instances and from the persistent store. For details, see Section 10.8, “Cache Loaders”.
   - A cache store extends the cache loader functionality to include write capabilities by exposing methods that allow the cache loader to load and store states. For details, see Chapter 10, *Cache Stores and Cache Loaders*.

3. The Level 1 Cache (or L1 Cache) stores remote cache entries after they are initially accessed, preventing unnecessary remote fetch operations for each subsequent use of the same entries. For details, see Chapter 14, *The L1 Cache*.

4. The Cache Manager is the primary mechanism used to retrieve a Cache instance in JBoss Data Grid, and can be used as a starting point for using the Cache. For details, see Chapter 11, *Cache Managers*.

5. The Cache houses cache instances retrieved by a Cache Manager.

6. Cache Interfaces use protocols such as Memcached and Hot Rod, or REST to interface with the cache. For details about the remote interfaces, refer to the *Developer Guide*.

    - Memcached is an in-memory caching system used to improve response and operation times for database-driver websites. The Memcached caching system defines a text based, client-server caching protocol called the Memcached protocol.

    - Hot Rod is a binary TCP client-server protocol used in JBoss Data Grid. It was created to overcome deficiencies in other client/server protocols, such as Memcached. Hot Rod enables clients to do smart routing of requests in partitioned or distributed JBoss Data Grid server clusters.

    - The REST protocol eliminates the need for tightly coupled client libraries and bindings. The REST API introduces an overhead, and requires a REST client or custom code to understand and create REST calls.

7. An application allows the user to interact with the cache via a cache interface. Browsers are a common example of such end-user applications.

Report a bug

1.7. JBOSS DATA GRID APIS

1.7.1. JBoss Data Grid APIs

JBoss Data Grid provides the following APIs:

- Cache
- Batching
- Grouping
- CacheStore
- Externalizable

In JBoss Data Grid's Remote Client-Server mode, only the following APIs can be used to interact with the data grid:

- The Asynchronous API (can only be used in conjunction with the Hot Rod Client in Remote Client-Server Mode)
- The REST Interface
- The Memcached Interface

- The Hot Rod Interface
  - The RemoteCache API

Report a bug

1.7.2. About the Asynchronous API

In addition to synchronous API methods, JBoss Data Grid also offers an asynchronous API that provides the same functionality in a non-blocking fashion.

The asynchronous method naming convention is similar to their synchronous counterparts, with **Async** appended to each method name. Asynchronous methods return a Future that contains the result of the operation.

For example, in a cache parameterized as ***`Cache(String, String)`***, ***`Cache.put(String key, String value)`*** returns a String, while ***`Cache.putAsync(String key, String value)`*** returns a **`Future(String)`**.

Report a bug

1.7.3. About the Batching API

The Batching API is used when the JBoss Data Grid cluster is the sole participant in a transaction. However, Java Transaction API (JTA) transactions (which use the Transaction Manager) should be used when multiple systems are participants in the transaction.

NOTE

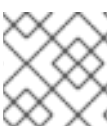
The Batching API cannot be used in JBoss Data Grid's Remote Client-Server mode.

Report a bug

1.7.4. About the Cache API

The Cache interface provides simple methods for the addition, retrieval and removal of entries, which includes atomic mechanisms exposed by the JDK's **`ConcurrentMap`** interface. How entries are stored depends on the cache mode in use. For example, an entry may be replicated to a remote node or an entry may be looked up in a cache store.

The Cache API is used in the same manner as the JDK Map API for basic tasks. This simplifies the process of migrating from Map-based, simple in-memory caches to JBoss Data Grid's cache.



NOTE

This API is not available in JBoss Data Grid's Remote Client-Server Mode

Report a bug

1.7.5. About the RemoteCache Interface

The RemoteCache Interface allows clients outside JBoss Data Grid to access the Hot Rod server module within JBoss Data Grid. The RemoteCache Interface offers optional features such as distribution and eviction.

Report a bug

1.8. TOOLS AND OPERATIONS

1.8.1. About Management Tools

Managing JBoss Data Grid instances requires exposing significant amounts of relevant statistical information. This information allows administrators to get a clear view of each JBoss Data Grid node's state. A single installation can comprise of tens or hundreds of JBoss Data Grid nodes and it is important to provide this information in a clear and concise manner. JBoss Operations Network is one example of a tool that provides runtime visibility. Other tools, such as **JConsole** can be used where JMX is enabled.

See Also:

- Chapter 21, *Management Tools in JBoss Data Grid*

Report a bug

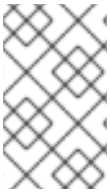
1.8.2. Accessing Data via URLs

Caches that have been configured with a REST interface have access to JBoss Data Grid using RESTful HTTP access.

The RESTful service only requires a HTTP client library, eliminating the need for tightly coupled client libraries and bindings.

HTTP **`put()`** and **`post()`** methods place data in the cache, and the URL used determines the cache name and key(s) used. The data is the value placed into the cache, and is placed in the body of the request.

A Content-Type header must be set for these methods. **GET** and **HEAD** methods are used for data retrieval while other headers control cache settings and behavior.



NOTE

It is not possible to have conflicting server modules interact with the data grid. Caches must be configured with a compatible interface in order to have access to JBoss Data Grid.

Report a bug

1.8.3. Limitations of Map Methods

Specific **`Map`** methods, such as **`size()`**, **`values()`**, **`keySet()`** and **`entrySet()`**, can be used with certain limitations with JBoss Data Grid as they are unreliable. These methods do not acquire locks (global or local) and concurrent modification, additions and removals are excluded from consideration in these calls. Furthermore, the listed methods are only operational on the local data container and do not provide a global view of state.

If the listed methods acted globally, it would result in a significant impact on performance and would produce a scalability bottleneck. As a result, it is recommended that these methods are used for informational and debugging purposes only.

Report a bug

CHAPTER 2. LOGGING IN JBOSS DATA GRID

2.1. AN OVERVIEW OF LOGGING IN JBOSS DATA GRID

JBoss Data Grid 6 provides highly configurable logging facilities for both its own internal use and for use by deployed applications. The logging subsystem is based on JBoss LogManager and it supports several third party application logging frameworks in addition to JBoss Logging.

The logging subsystem is configured using a system of log categories and log handlers. Log categories define what messages to capture, and log handlers define how to deal with those messages (write to disk, send to console, etc).

Report a bug

2.2. SUPPORTED APPLICATION LOGGING FRAMEWORKS

2.2.1. Supported Application Logging Frameworks

JBoss LogManager supports the following logging frameworks:

- JBoss Logging, which is included with JBoss Data Grid 6.
- Apache Commons Logging
- Simple Logging Facade for Java (SLF4J)
- Apache log4j
- Java SE Logging (java.util.logging)

Report a bug

2.2.2. About JBoss Logging

JBoss Logging is the application logging framework that is included in JBoss Enterprise Application Platform 6. As a result of this inclusion, JBoss Data Grid 6 also uses JBoss Logging.

JBoss Logging provides an easy way to add logging to an application. You add code to your application that uses the framework to send log messages in a defined format. When the application is deployed to an application server, these messages can be captured by the server and displayed and/or written to file according to the server's configuration.

Report a bug

2.2.3. JBoss Logging Features

JBoss Logging includes the following features:

- Provides an innovative, easy to use *typed* logger.
- Full support for internationalization and localization. Translators work with message bundles in properties files while developers can work with interfaces and annotations.

- Build-time tooling to generate typed loggers for production, and runtime generation of typed loggers for development.

Report a bug

2.3. CONFIGURE LOGGING

2.3.1. About Boot Logging

The boot log is the record of events that occur while the server is starting up (or "booting"). JBoss Data Grid 6 also includes a server log, which includes log entries generated after the server concludes the boot process.

Report a bug

2.3.2. Configure Boot Logging

Edit the **`logging.properties`** file to configure the boot log. This file is a standard Java properties file and can be edited in a text editor. Each line in the file has the format of **`property=value`**.

In JBoss Data Grid, the **`logging.properties`** file is available in the **`$JDG_HOME/standalone/configuration`** folder.

Report a bug

2.3.3. Default Log File Locations

The following table provides a list of log files in JBoss Data Grid and their locations:

Table 2.1. Default Log File Locations

Log File	Location	Description
`boot.log`	**`$JDG_HOME/standalone/log/`**	The Server Boot Log. Contains log messages related to the start up of the server.
`server.log`	**`$JDG_HOME/standalone/log/`**	The Server Log. Contains all log messages once the server has launched.

Report a bug

2.4. LOGGING ATTRIBUTES

2.4.1. About Log Levels

Log levels are an ordered set of enumerated values that indicate the nature and severity of a log message. The level of a given log message is specified by the developer using the appropriate methods of their chosen logging framework to send the message.

JBoss Data Grid 6 supports all the log levels used by the supported application logging frameworks.

The six most commonly used log levels are (ordered by lowest to highest):

1. **TRACE**
2. **DEBUG**
3. **INFO**
4. **WARN**
5. **ERROR**
6. **FATAL**

Log levels are used by log categories and handlers to limit the messages they are responsible for. Each log level has an assigned numeric value which indicates its order relative to other log levels. Log categories and handlers are assigned a log level and they only process log messages of that level or higher. For example a log handler with the level of **WARN** will only record messages of the levels **WARN**, **ERROR** and **FATAL**.

Report a bug

2.4.2. Supported Log Levels

The following table lists log levels that are supported in JBoss Data Grid. Each entry includes the log level, its value and description. The log level values indicate each log level's relative value to other log levels. Additionally, log levels in different frameworks may be named differently, but have a log value consistent to the provided list.

Table 2.2. Supported Log Levels

Log Level	Value	Description
FINEST	300	-
FINER	400	-
TRACE	400	Used for messages that provide detailed information about the running state of an application. **TRACE** level log messages are captured when the server runs with the **TRACE** level enabled.
DEBUG	500	Used for messages that indicate the progress of individual requests or activities of an application. **DEBUG** level log messages are captured when the server runs with the **DEBUG** level enabled.
FINE	500	-

Log Level	Value	Description
CONFIG	700	-
INFO	800	Used for messages that indicate the overall progress of the application. Used for application start up, shut down and other major lifecycle events.
WARN	900	Used to indicate a situation that is not in error but is not considered ideal. Indicates circumstances that can lead to errors in the future.
WARNING	900	-
ERROR	1000	Used to indicate an error that has occurred that could prevent the current activity or request from completing but will not prevent the application from running.
FATAL	1100	Used to indicate events that could cause critical service failure and application shutdown and possibly cause JBoss Data Grid 6 to shut down.

Report a bug

2.4.3. About Log Categories

Log categories define a set of log messages to capture and one or more log handlers which will process the messages.

The log messages to capture are defined by their Java package of origin and log level. Messages from classes in that package and of that log level or lower are captured by the log category and sent to the specified log handlers. As an example, the **DEBUG** log level results in log values of **300**, **400** and **500** are captured.

Log categories can optionally use the log handlers of the root logger instead of their own handlers.

Report a bug

2.4.4. About the Root Logger

The root logger captures all log messages sent to the server (of a specified level) that are not captured by a log category. These messages are then sent to one or more log handlers.

By default the root logger is configured to use a console and a periodic log handler. The periodic log handler is configured to write to the file `server.log`. This file is sometimes referred to as the server log.

Report a bug

2.4.5. About Log Handlers

Log handlers define how captured log messages are recorded by JBoss Data Grid. The six types of log handlers configurable in JBoss Data Grid are:

- `Console`
- `File`
- `Periodic`
- `Size`
- `Async`
- `Custom`

Report a bug

2.4.6. Log Handler Types

The following table lists the different types of log handlers available in JBoss Data Grid:

Table 2.3. Log Handler Types

Log Handler Type	Description
Console	Console log handlers write log messages to either the host operating system's standard out (`stdout`) or standard error (`stderr`) stream. These messages are displayed when JBoss Data Grid 6 is run from a command line prompt. The messages from a Console log handler are not saved unless the operating system is configured to capture the standard out or standard error stream.
File	File log handlers are the simplest log handlers. Their primary use is to write log messages to a specified file.
Periodic	Periodic file handlers write log messages to a named file until a specified period of time has elapsed. Once the time period has elapsed, the specified time stamp is appended to the file name. The handler then continues to write into the newly created log file with the original name.

Log Handler Type	Description
Size	Size log handlers write log messages to a named file until the file reaches a specified size. When the file reaches a specified size, it is renamed with a numeric prefix and the handler continues to write into a newly created log file with the original name. Each size log handler must specify the maximum number of files to be kept in this fashion.
Async	Async log handlers are wrapper log handlers that provide asynchronous behavior for one or more other log handlers. These are useful for log handlers that have high latency or other performance problems such as writing a log file to a network file system.
Custom	Custom log handlers enable to you to configure new types of log handlers that have been implemented. A custom handler must be implemented as a Java class that extends **`java.util.logging.Handler`** and be contained in a module.

Report a bug

2.4.7. About Log Formatters

A log formatter is the configuration property of a log handler. The log formatter defines the appearance of log messages that originate from the relevant log handler. The log formatter is a string that uses the same syntax as the **`java.util.Formatter`** class.

Refer to: http://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html

Report a bug

2.5. LOGGING CONFIGURATION PROPERTIES

2.5.1. Root Logger Properties

Table 2.4. Root Logger Properties

Property	Datatype	Description
level	string	The maximum level of log message that the root logger records.
handlers	list of strings	A list of log handlers that are used by the root logger.

Report a bug

2.5.2. Log Category Properties

Table 2.5. Log Category Properties

Property	Datatype	Description
level	string	The maximum level of log message that the log category records.
handlers	list of strings	A list of log handlers that are used by the root logger.
use-parent-handlers	boolean	If set to true, this category will use the log handlers of the root logger in addition to any other assigned handlers.
category	string	The log category from which log messages will be captured.

Report a bug

2.5.3. Console Log Handler Properties

Table 2.6. Console Log Handler Properties

Property	Datatype	Description
level	string	The maximum level of log message the log handler records.
encoding	string	The character encoding scheme to be used for the output.
formatter	string	The log formatter used by this log handler.
target	string	The system output stream where the output of the log handler goes. This can be System.err or System.out for the system error stream or standard out stream respectively.
autoflush	boolean	If set to true the log messages will be sent to the handlers target immediately upon receipt.
name	string	The unique identifier for this log handler.

Report a bug

2.5.4. File Log Handler Properties

Table 2.7. File Log Handler Properties

Property	Datatype	Description
level	string	The maximum level of log message the log handler records.
encoding	string	The character encoding scheme to be used for the output.

Property	Datatype	Description
formatter	string	The log formatter used by this log handler.
append	boolean	If set to true then all messages written by this handler will be appended to the file if it already exists. If set to false a new file will be created each time the application server launches. Changes to **append** require a server reboot to take effect.
autoflush	boolean	If set to true the log messages will be sent to the handlers assigned file immediately upon receipt. Changes to **autoflush** require a server reboot to take effect.
name	string	The unique identifier for this log handler.
file	object	The object that represents the file where the output of this log handler is written to. It has two configuration properties, **relative-to** and **path**.
relative-to	string	This is a property of the file object and is the directory where the log file is written to. JBoss Enterprise Application Platform 6 file path variables can be specified here. The **jboss.server.log.dir** variable points to the **log/** directory of the server.
path	string	This is a property of the file object and is the name of the file where the log messages will be written. It is a relative path name that is appended to the value of the **relative-to** property to determine the complete path.

Report a bug

2.5.5. Periodic Log Handler Properties

Table 2.8. Periodic Log Handler Properties

Property	Datatype	Description
append	boolean	If set to true then all messages written by this handler will be appended to the file if it already exists. If set to false a new file will be created each time the application server launches. Changes to append require a server reboot to take effect.
autoflush	boolean	If set to true the log messages will be sent to the handlers assigned file immediately upon receipt. Changes to autoflush require a server reboot to take effect.
encoding	string	The character encoding scheme to be used for the output.
formatter	string	The log formatter used by this log handler.
level	string	The maximum level of log message the log handler records.

Property	Datatype	Description
name	string	The unique identifier for this log handler.
file	object	Object that represents the file where the output of this log handler is written to. It has two configuration properties, ***relative-to*** and ***path***.
relative-to	string	This is a property of the file object and is the directory where the log file is written to. File path variables can be specified here. The ***jboss.server.log.dir*** variable points to the **log/>** directory of the server.
path	string	This is a property of the file object and is the name of the file where the log messages will be written. It is a relative path name that is appended to the value of the ***relative-to*** property to determine the complete path.
suffix	string	This is a string with is both appended to filename of the rotated logs and is used to determine the frequency of rotation. The format of the suffix is a dot (.) followed by a date string which is parsable by the **java.text.SimpleDateFormat** class. The log is rotated on the basis of the smallest time unit defined by the suffix. For example the suffix **.yyyy-MM-dd** will result in daily log rotation. Refer to http://docs.oracle.com/javase/6/docs/api/index.html?java/text/SimpleDateFormat.html

Report a bug

2.5.6. Size Log Handler Properties

Table 2.9. Size Log Handler Properties

Property	Datatype	Description
append	boolean	If set to true then all messages written by this handler will be appended to the file if it already exists. If set to false a new file will be created each time the application server launches. Changes to append require a server reboot to take effect.
autoflush	boolean	If set to true the log messages will be sent to the handlers assigned file immediately upon receipt. Changes to autoflush require a server reboot to take effect.
encoding	string	The character encoding scheme to be used for the output.
formatter	string	The log formatter used by this log handler.
level	string	The maximum level of log message the log handler records.
name	string	The unique identifier for this log handler.

Property	Datatype	Description
file	object	Object that represents the file where the output of this log handler is written to. It has two configuration properties, ***relative-to*** and ***path***.
relative-to	string	This is a property of the file object and is the directory where the log file is written to. File path variables can be specified here. The **jboss.server.log.dir** variable points to the **log/** directory of the server.
path	string	This is a property of the file object and is the name of the file where the log messages will be written. It is a relative path name that is appended to the value of the **relative-to** property to determine the complete path.
rotate-size	integer	The maximum size that the log file can reach before it is rotated. A single character appended to the number indicates the size units: **b** for bytes, **k** for kilobytes, **m** for megabytes, **g** for gigabytes. Eg. **50m** for 50 megabytes.
max-backup-index	integer	The maximum number of rotated logs that are kept. When this number is reached, the oldest log is reused.

Report a bug

2.5.7. Async Log Handler Properties

Table 2.10. Async Log Handler Properties

Property	Datatype	Description
level	string	The maximum level of log message the log handler records.
name	string	The unique identifier for this log handler.
Queue-length	integer	Maximum number of log messages that will be held by this handler while waiting for sub-handlers to respond.
overflow-action	string	How this handler responds when its queue length is exceeded. This can be set to **BLOCK** or **DISCARD**. **BLOCK** makes the logging application wait until there is available space in the queue. This is the same behavior as an non-async log handler. **DISCARD** allows the logging application to continue but the log message is deleted.
subhandlers	list of strings	This is the list of log handlers to which this async handler passes its log messages.

Report a bug

2.6. LOGGING SAMPLE CONFIGURATIONS

2.6.1. Sample XML Configuration for the Root Logger

```
<subsystem xmlns="urn:jboss:domain:logging:1.1">

   <root-logger>
      <level name="INFO"/>
      <handlers>
         <handler name="CONSOLE"/>
         <handler name="FILE"/>
      </handlers>
   </root-logger>

</subsystem>
```

Report a bug

2.6.2. Sample XML Configuration for a Log Category

```
<subsystem xmlns="urn:jboss:domain:logging:1.1">

   <logger category="com.company.accounts.rec">
      <handlers>
         <handler name="accounts-rec"/>
      </handlers>
   </logger>

</subsystem>
```

Report a bug

2.6.3. Sample XML Configuration for a Console Log Handler

```
<subsystem xmlns="urn:jboss:domain:logging:1.1">

   <console-handler name="CONSOLE">
      <level name="INFO"/>
      <formatter>
         <pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t)
%s%E%n"/>
      </formatter>
   </console-handler>

</subsystem>
```

Report a bug

2.6.4. Sample XML Configuration for a File Log Handler

```
<file-handler name="accounts-rec-trail" autoflush="true">
    <level name="INFO"/>
    <file relative-to="jboss.server.log.dir" path="accounts-rec-
```

```
trail.log"/>
    <append value="true"/>
</file-handler>
```

Report a bug

2.6.5. Sample XML Configuration for a Periodic Log Handler

```
<periodic-rotating-file-handler name="FILE">
   <formatter>
      <pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t)
%s%E%n"/>
   </formatter>
   <file relative-to="jboss.server.log.dir" path="server.log"/>
   <suffix value=".yyyy-MM-dd"/>
   <append value="true"/>
</periodic-rotating-file-handler>
```

Report a bug

2.6.6. Sample XML Configuration for a Size Log Handler

```
<size-rotating-file-handler name="accounts_debug" autoflush="false">
   <level name="DEBUG"/>
   <file relative-to="jboss.server.log.dir" path="accounts-debug.log"/>
   <rotate-size value="500k"/>
   <max-backup-index value="5"/>
   <append value="true"/>
</size-rotating-file-handler>
```

Report a bug

2.6.7. Sample XML Configuration for a Async Log Handler

```
<async-handler name="Async_NFS_handlers">
   <level name="INFO"/>
   <queue-length value="512"/>
   <overflow-action value="block"/>
   <subhandlers>
      <handler name="FILE"/>
      <handler name="accounts-record"/>
   </subhandlers>
</async-handler>
```

Report a bug

CHAPTER 3. HIGH AVAILABILITY USING SERVER HINTING

3.1. ABOUT SERVER HINTING

In JBoss Data Grid, the term Server Hinting ensures that backup copies of data are not stored on the same physical server, rack or data center as the original. Server Hinting does not apply to total replication because total replication mandates complete replicas on every server, rack and data center.

Server Hinting is particularly important when ensuring the high availability of your JBoss Data Grid implementation.

Report a bug

3.2. ESTABLISHING SERVER HINTING WITH JGROUPS

When setting up a clustered environment in JBoss Data Grid, Server Hinting is configured when establishing JGroups configuration.

JBoss Data Grid ships with several JGroups files pre-configured for clustered mode, and using Server Hinting. These files can be used as a starting point when configuring clustered modes in JBoss Data Grid.

See Also:

- Section 20.4.2, “Pre-Configured JGroups Files”

Report a bug

3.3. CONFIGURE SERVER HINTING IN REMOTE CLIENT-SERVER MODE

In JBoss Data Grid's Remote Client-Server mode, Server Hinting is configured using the subsystem tags, as follows:

```
<subsystem xmlns="urn:jboss:domain:jgroups:1.1"
     default-stack="${jboss.default.jgroups.stack:udp}"  >
     <stack name="udp">
       <transport type="UDP"
                           socket-binding="jgroups-udp"
                           site="${jboss.jgroups.transport.site:s1}"
         rack="${jboss.jgroups.transport.rack:r1}"

machine="${jboss.jgroups.transport.machine:m1}">
                      ...
       </transport>
        </stack>
</subsystem>
```

Report a bug

3.4. CONFIGURE SERVER HINTING IN LIBRARY MODE

In JBoss Data Grid's Library mode, Server Hinting is configured at the transport level. The following is a Server Hinting sample configuration:

```
<transport clusterName = "MyCluster"
           machineId = "LinuxServer01"
           rackId = "Rack01"
           siteId = "US-WestCoast" />
```

As illustrated in the sample configuration, the following configuration attributes are used to configure Server Hinting in JBoss Data Grid:

- The ***clusterName*** attribute specifies the name assigned to the cluster.
- The ***machineId*** attribute specifies the JVM instance that houses the original data. This is particularly useful for nodes with multiple JVMs and physical hosts with multiple virtual hosts.
- The ***rackId*** parameter specifies the rack that contains the original data, so that other racks are used for backups.
- The ***siteId*** parameter differentiates between nodes in different data centers replicating to each other.

The listed parameters are optional in a JBoss Data Grid configuration. However, if Server Hinting is not configured, JBoss Data Grid's distribution algorithms are allowed to store replications in the same node/data center/rack as the original data.

Report a bug

CHAPTER 4. CACHE MODES

4.1. ABOUT CACHE MODES

JBoss Data Grid provides two modes:

- Local (Standalone) mode.
- Clustered mode.

Report a bug

4.2. LOCAL MODE

4.2.1. About Local Mode

Local mode is the only non-clustered cache mode offered in JBoss Data Grid. In local mode, JBoss Data Grid operates as a simple in-memory data cache, similar to JBoss Cache and EHCache .

Report a bug

4.2.2. Local Mode Operations

Using JBoss Data Grid's local mode instead of a map provides a number of benefits.

Caches offer features that are unmatched by simple maps, such as:

- Write-through and write-behind caching to persist data.
- Entry eviction to prevent the Java Virtual Machine (JVM) running out of memory.
- Support for entries that expire after a defined period.

JBoss Data Grid is built around a high performance, read-biased data container that uses techniques such as optimistic and pessimistic locking to manage lock acquisitions.

JBoss Data Grid also uses compare-and-swap and other lock-free algorithms, resulting in high throughput multi-CPU or multi-core environments. Additionally, JBoss Data Grid's Cache API extends the JDK's **`ConcurrentMap`**, resulting in a simple migration process from a map to JBoss Data Grid.

Report a bug

4.2.3. Configure Local Mode

A local cache can be added to any cache container. For example:

```
<cache-container name="local"
   default-cache="default"
    listener-executor="infinispan-listener">
 <local-cache name="default"
      start="EAGER">
  <locking isolation="NONE"
    acquire-timeout="30000"
    concurrency-level="1000"
```

```
    striping="false" />
  <transaction mode="NONE" />
 </local-cache>
</cache-container>
```

Alternatively, create a **DefaultCacheManager** with the "no-argument" constructor. Both of these methods create a local default cache.

Local and clustered caches are able to coexist in the same cache container, however where the container is without a **<transport/>** it can only contain local caches. The container used in the example can only contain local caches as it does not have a **<transport/>**.

The cache interface extends the **ConcurrentMap** and is compatible with multiple cache systems.

Report a bug

4.3. CLUSTERED MODES

4.3.1. About Clustered Modes

JBoss Data Grid provides a clustered mode to replicate state changes to a small subset of nodes. The subset size is sufficient for fault tolerance purposes but not large enough to hinder scalability. Before attempting to use clustered mode, it is important to first configure JGroups for a clustered configuration.

See Also:

- Section 20.4, “JGroups for Clustered Modes”

Report a bug

4.3.2. Clustered Mode Operations

JBoss Data Grid offers the following clustered modes:

- Replication Mode replicates any entry that is added across all cache instances in the cluster.
- Invalidation Mode does not share any data, but but signals remote caches to initiate the removal of invalid entries.
- Distribution Mode stores each entry on a subset of nodes instead of on all nodes in the cluster.

The clustered modes can be further configured to use synchronous or asynchronous transport for network communications.

Report a bug

4.3.3. Asynchronous and Synchronous Operations

When a clustered mode (such as invalidation, replication or distribution) is used, data is propagated to other nodes in either a synchronous or asynchronous manner.

If synchronous mode is used, the sender waits for responses from receivers before allowing the thread to continue, whereas asynchronous mode transmits data but does not wait for responses from other nodes in the cluster to continue operations.

Asynchronous mode prioritizes speed over consistency, which is ideal for use cases such as HTTP session replications with sticky sessions enabled. Such a session (or data for other use cases) is always accessed on the same cluster node, unless this node fails.

Report a bug

4.3.4. Cache Mode Troubleshooting

4.3.4.1. Invalid Data in ReadExternal

If invalid data is passed to **`readExternal`**, it can be because when using **`Cache.putAsync()`**, starting serialization can cause your object to be modified, causing the datastream passed to **`readExternal`** to be corrupted. This can be resolved if access to the object is synchronized.

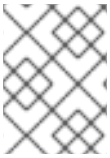
Report a bug

4.3.4.2. About Asynchronous Communications

In JBoss Data Grid, the local, distributed and replicated modes are represented by the **`local-cache`**, **`distributed-cache`** and **`replicated-cache`** elements respectively. Each of these elements contains a ***mode*** property, the value of which can be set to **`SYNC`** for synchronous or **`ASYNC`** for asynchronous communications.

An example of this configuration is as follows:

```
<replicated-cache name="default"
                  start="EAGER"
                  mode="SYNC"
                  batching="false" >
                 ...
</replicated-cache>
```



NOTE

This configuration is valid for both JBoss Data Grid's usage modes (Library mode and Remote Client-Server mode).

Report a bug

4.3.4.3. Cluster Physical Address Retrieval

How can the physical addresses of the cluster be retrieved?

The physical address can be retrieved using an instance method call. For example: **`AdvancedCache.getRpcManager().getTransport().getPhysicalAddresses()`**.

Report a bug

CHAPTER 5. DISTRIBUTION MODE

5.1. ABOUT DISTRIBUTION MODE

When enabled, JBoss Data Grid's distribution mode stores each entry on a subset of the nodes in the grid instead of replicating each entry on every node. Typically, each entry is stored on more than one node for redundancy and fault tolerance.

As a result of storing entries on selected nodes across the cluster, distribution mode provides improved scalability compared to other clustered modes.

A cache using distribution mode can transparently locate keys across a cluster using the consistent hash algorithm.

Report a bug

5.2. DISTRIBUTION MODE'S CONSISTENT HASH ALGORITHM

Distribution mode uses a consistent hash algorithm to select a node from the cluster to store entries upon. The consistent hash algorithm is configured with the number of copies of each cache entry to be maintained within the cluster.

The number of copies set for each data item requires balancing performance and fault tolerance. Creating too many copies of the entry can impair performance and too few copies can result in data loss in case of node failure.

Report a bug

5.3. LOCATING ENTRIES IN DISTRIBUTION MODE

The consistent hash algorithm used in JBoss Data Grid's distribution mode can locate entries deterministically, without multicasting a request or maintaining expensive metadata.

A **PUT** operation can result in as many remote calls as specified by the ***num_copies*** parameter, while a **GET** operation executed on any node in the cluster results in a single remote call. In the background, the **GET** operation results in the same number of remote calls as a **PUT** operation (specifically the value of the ***num_copies*** parameter), but these occur in parallel and the returned entry is passed to the caller as soon as one returns.

Report a bug

5.4. RETURN VALUES IN DISTRIBUTION MODE

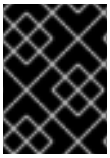
In JBoss Data Grid's distribution mode, a synchronous request is used to retrieve the previous return value if it cannot be found locally. A synchronous request is used for this task irrespective of whether distribution mode is using asynchronous or synchronous processes.

Report a bug

5.5. CONFIGURE DISTRIBUTION MODE

Distribution mode is a clustered mode in JBoss Data Grid. Distribution mode can be added to any cache container using the following:

```
<cache-container name="local"
   default-cache="default"
    listener-executor="infinispan-listener">
 <distributed-cache name="default"
       start="EAGER">
  <locking isolation="NONE"
    acquire-timeout="30000"
    concurrency-level="1000"
    striping="false" />
  <transaction mode="NONE" />
 </distributed-cache>
</cache-container>
```



IMPORTANT

JGroups must be appropriately configured for clustered mode before attempting to load this configuration.

See Also:

- Section 20.4, "JGroups for Clustered Modes"

Report a bug

5.6. SYNCHRONOUS AND ASYNCHRONOUS DISTRIBUTION

5.6.1. About Synchronous and Asynchronous Distribution

Distribution mode only supports synchronous communication. To elicit meaningful return values from certain public API methods, it is essential to use synchronized communication when using distribution mode.

Example 5.1. Communication Mode example

For example, with three caches in a cluster, cache **A**, **B** and **C**, and a key **K** that maps cache **A** to **B**. Perform an operation on cluster **C** that requires a return value, for example `Cache.remove(K)`. To execute successfully, the operation must first synchronously forward the call to both cache **A** and **B**, and then wait for a result returned from either cache **A** or **B**. If asynchronous communication was used, the usefulness of the returned values cannot be guaranteed, despite the operation behaving as expected.

Report a bug

5.7. GET AND PUT USAGE IN DISTRIBUTION MODE

5.7.1. About GET and PUT Operations in Distribution Mode

In distribution mode, the cache performs a remote **GET** command before a write command. This occurs because certain methods (for example, `Cache.put()`) return the previous value associated with the specified key according to the `java.util.Map` contract. When this is performed on an instance that

does not own the key and the entry is not found in the L1 cache, the only reliable way to elicit this return value is to perform a remote **GET** before the **PUT**.

The **GET** operation that occurs before the **PUT** operation is always synchronous, whether the cache is synchronous or asynchronous, because JBoss Data Grid must wait for the return value.

Report a bug

5.7.2. Distributed GET and PUT Operation Resource Usage

In distribution mode, the cache may execute a **GET** operation before executing the desired **PUT** operation.

This operation is very expensive in terms of resources. Despite operating in an synchronous manner, a remote **GET** operation does not wait for all responses, which would result in wasted resources. The **GET** process accepts the first valid response received, which allows its performance to be unrelated to cluster size.

Disable the ***`Flag.SKIP_REMOTE_LOOKUP`*** flag for a per-invocation setting if return values are not required for your implementation.

Such actions do not impair cache operations and the accurate functioning of all public methods, but do break the **`java.util.Map`** interface contract. The contract breaks because unreliable and inaccurate return values are provided to certain methods. As a result, ensure that these return values are not used for any important purpose on your configuration.

Report a bug

CHAPTER 6. REPLICATION MODE

6.1. ABOUT REPLICATION MODE

JBoss Data Grid's replication mode is a simple clustered mode. Cache instances automatically discover neighboring instances on other Java Virtual Machines (JVM) on the same network and subsequently form a cluster with the discovered instances. Any entry added to a cache instance is replicated across all cache instances in the cluster and can be retrieved locally from any cluster cache instance.

Report a bug

6.2. OPTIMIZED REPLICATION MODE USAGE

Replication mode is used for state sharing across a cluster. However, the cluster performance is optimal only when the target cluster contains less than ten servers.

In larger clusters, the fact that a large number of replication messages must be transmitted results in reduced performance.

JBoss Data Grid can be configured to use UDP multicast, which improves performance to a limited degree for larger clusters.

Report a bug

6.3. RETURN VALUES IN REPLICATION MODE

In JBoss Data Grid's replication mode, return values are locally available before the replication occurs.

Report a bug

6.4. CONFIGURE REPLICATION MODE

Replication mode is a clustered cache mode in JBoss Data Grid. Replication mode can be added to any cache container using the following:

```
<cache-container name="local"
   default-cache="default"
    listener-executor="infinispan-listener">
 <replicated-cache name="default"
       start="EAGER">
  <locking isolation="NONE"
    acquire-timeout="30000"
    concurrency-level="1000"
    striping="false" />
  <transaction mode="NONE" />
 </replicated-cache>
</cache-container>
```

IMPORTANT

JGroups must be appropriately configured for clustered mode before attempting to load this configuration.

See Also:

- Section 20.4, "JGroups for Clustered Modes"

Report a bug

6.5. SYNCHRONOUS AND ASYNCHRONOUS REPLICATION

6.5.1. Synchronous and Asynchronous Replication

Replication mode can be synchronous or asynchronous depending on the problem being addressed.

- Synchronous replication blocks a thread or caller (for example on a `put()` operation) until the modifications are replicated across all nodes in the cluster.By waiting for acknowledgments, synchronous replication ensures that all replications are successfully applied before the operation is concluded.

- Asynchronous replication operates significantly faster than synchronous replication because it does not need to wait for responses from nodes.Asynchronous replication performs the replication in the background and the call returns immediately. Errors that occur during asynchronous replication are written to a log. As a result, a transaction can be successfully completed despite the fact that replication of the transaction may not have succeeded on all the cache instances in the cluster.

Report a bug

6.5.2. Troubleshooting Asynchronous Replication Behavior

In some instances, a cache configured for asynchronous replication or distribution may wait for responses, which is synchronous behavior. This occurs because caches behave synchronously when both state transfers and asynchronous modes are configured. This synchronous behavior is a prerequisite for state transfer to operate as expected.

Use one of the following to remedy this problem:

- Disable state transfer and use a `ClusteredCacheLoader` to lazily look up remote state as and when needed.

- Enable state transfer and ***REPL_SYNC***. Use the Asynchronous API (for example, the `cache.putAsync(k, v)`) to activate 'fire-and-forget' capabilities.

- Enable state transfer and ***REPL_ASYNC***. All RPCs end up becoming synchronous, but client threads will not be held up if you enable a replication queue (which is recommended for asynchronous mode).

Report a bug

6.6. THE REPLICATION QUEUE

6.6.1. Replication Queue

In replication mode, JBoss Data Grid uses a replication queue to replicate changes across nodes based on the following:

- Previously set intervals.
- The queue size exceeding the number of elements.
- A combination of previously set intervals and the queue size exceeding the number of elements.

The replication queue ensures that during replication, cache operations are transmitted in batches instead of individually. As a result, a lower number of replication messages are transmitted and fewer envelopes are used, resulting in improved JBoss Data Grid performance.

A replication queue is used in conjunction with asynchronous mode.

Report a bug

6.6.2. Replication Queue Operations

When using the replication queue, the primary disadvantage is that the queue is periodically flushed based on the time or the queue size. Such flushing operations delay the realization of replication, distribution or invalidation operations across cluster nodes. With the replication queue disabled, however, the data is directly transmitted and therefore arrives at the cluster nodes faster.

Report a bug

6.6.3. Replication Queue Usage

When using the replication queue, do one of the following:

- Disable asynchronous marshalling; or
- Set the ***max-threads*** count value to **1** for the **transport executor**. The **transport executor** is defined in **standalone.xml** as follows:

```
<transport executor="infinispan-transport"/>
```

To implement either of these solutions, the replication queue must be in use in asynchronous mode. Asynchronous mode can be set, along with the queue timeout (***queue-flush-interval***, value is in milliseconds) and queue size (***queue-size***) as follows:

```
<replicated-cache name="asyncCache"
                  start="EAGER"
                  mode="ASYNC"
                  batching="false"
                  indexing="NONE"
                  queue-size="1000"
                  queue-flush-interval="500">
               ...
</replicated-cache>
```

The replication queue allows requests to return to the client faster, therefore using the replication queue together with asynchronous marshalling does not present any significant advantages.

Report a bug

6.7. FREQUENTLY ASKED QUESTIONS

6.7.1. About Replication Guarantees

In a clustered cache, the user can receive synchronous replication guarantees as well as the parallelism associated with asynchronous replication. JBoss Data Grid provides an asynchronous API for this purpose.

The asynchronous methods used in the API return Futures, which can be queried. The queries block the thread until a confirmation is received about the success of any network calls used.

Report a bug

6.7.2. Replication Traffic on Internal Networks

Some cloud providers charge less for traffic over internal **IP** addresses than for traffic over public **IP** addresses, or do not charge at all for internal network traffic (for example, GoGrid). To take advantage of lower rates, you can configure JBoss Data Grid to transfer replication traffic using the internal network. With such a configuration, it is difficult to know the internal **IP** address you are assigned. JBoss Data Grid uses JGroups interfaces to solve this problem.

Report a bug

CHAPTER 7. INVALIDATION MODE

7.1. ABOUT INVALIDATION MODE

Invalidation is a clustered mode that does not share any data, but instead removes potentially obsolete data from remote caches. Using this cache mode requires another, more permanent store for the data such as a database.

Report a bug

7.2. USING INVALIDATION MODE

Invalidation mode requires a second permanent store for data, such as a database. JBoss Data Grid, in such a situation, is used as an optimization for a read-heavy system and prevents database usage each time a state is needed.

When invalidation mode is in use, data changes in a cache prompts other caches in the cluster to evict their outdated data from memory.

Report a bug

7.3. CONFIGURE INVALIDATION MODE

Invalidation mode is a clustered mode in JBoss Data Grid. Replication mode can be added to any cache container using the following:

```
<cache-container name="local"
   default-cache="default"
    listener-executor="infinispan-listener">
 <invalidated-cache name="default"
       start="EAGER">
  <locking isolation="NONE"
    acquire-timeout="30000"
    concurrency-level="1000"
    striping="false" />
  <transaction mode="NONE" />
 </invalidated-cache>
</cache-container>
```

IMPORTANT

JGroups must be appropriately configured for clustered mode before attempting to load this configuration.

See Also:

- Section 20.4, “JGroups for Clustered Modes”

Report a bug

7.4. SYNCHRONOUS/ASYNCHRONOUS INVALIDATION

In JBoss Data Grid's Library mode, invalidation operates either asynchronously or synchronously.

- Synchronous invalidation blocks the thread until all caches in the cluster have received invalidation messages and evicted the obsolete data.
- Asynchronous invalidation operates in a fire-and-forget mode that allows invalidation messages to be broadcast without blocking a thread to wait for responses.

Report a bug

7.5. THE L1 CACHE INVALIDATION

7.5.1. The L1 Cache and Invalidation

An invalidation message is generated each time a key is updated. This message is multicast to each node that contains data that corresponds to current L1 cache entries. The invalidation message ensures that each of these nodes marks the relevant entry as invalidated.

Report a bug

CHAPTER 8. CACHE WRITING MODES

8.1. WRITE-THROUGH AND WRITE-BEHIND CACHING

JBoss Data Grid presents configuration options with a single or multiple cache stores. This allows it to store data in a persistent location, for example a shared JDBC database or a local file system. JBoss Data Grid supports two caching modes:

- Write-Through (Synchronous)
- Write-Behind (Asynchronous)

Report a bug

8.2. WRITE-THROUGH CACHING

8.2.1. About Write-Through Caching

The Write-Through (or Synchronous) mode in JBoss Data Grid ensures that when clients update a cache entry (usually via a **Cache.put()** invocation), the call does not return until JBoss Data Grid has located and updated the underlying cache store. This feature allows updates to the cache store to be concluded within the client thread boundaries.

Report a bug

8.2.2. Write-Through Caching Benefits

The primary advantage of the Write-Through mode is that the cache and cache store are updated simultaneously, which ensures that the cache store remains consistent with the cache contents. This is at the cost of reduced performance for cache operations caused by the cache store accesses and updates during cache operations.

Report a bug

8.2.3. Write-Through Caching Configuration (Library Mode)

No specific configuration operations are required to configure a Write-Through or synchronous cache store. All cache stores are Write-Through or synchronous unless explicitly marked as Write-Behind or asynchronous. The following is a sample configuration file of a Write-Through unshared local file cache store:

```
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:infinispan:config:5.0">
 <global />
 <default />
 <namedCache name="persistentCache">
  <loaders shared="false">
   <loader class="org.infinispan.loaders.file.FileCacheStore"
           fetchPersistentState="true" ignoreModifications="false"
           purgeOnStartup="false">
    <properties>
     <property name="location"
```

```
        value="${java.io.tmpdir}" />
    </properties>
   </loader>
  </loaders>
   </namedCache>
</infinispan>
```

Report a bug

8.3. WRITE-BEHIND CACHING

8.3.1. About Write-Behind Caching

In JBoss Data Grid's Write-Behind (Asynchronous) mode, cache updates are asynchronously written to the cache store. Asynchronous updates ensure that cache store updates are carried out by a thread different from the client thread interacting with the cache.

One of the foremost advantages of the Write-Behind mode is that the cache operation performance is not affected by the underlying store update. However, because of the asynchronous updates, for a brief period the cache store houses stale data when compared to the cache.

Report a bug

8.3.2. Write-Behind Caching Configuration

To configure write-behind caching for any cache store, add the **write-behind** element to a cache store. The following example illustrates adding the **write-behind** element to a remote cache store configuration specifically, but can be used with any cache store type in the same manner:

```
<remote-store cache="default"
              socket-timeout="60000"
              tcp-no-delay="true"
              fetch-state="false"
              passivation="true"
              preload="true"
              purge="false">
 <remote-server outbound-socket-binding="remote-store-hotrod-server" />
 <write-behind flush-lock-timeout="1"
                     modification-queue-size="1024"
                     shutdown-timeout="25000"
                     thread-pool-size="1" />
</remote-store>
```

NOTE

This configuration is only used in JBoss Data Grid's Remote Client-Server mode.

Report a bug

8.3.3. About Unscheduled Write-Behind Strategy

In the Unscheduled Write-Behind Strategy mode, JBoss Enterprise Data Grid attempts to store

changes as quickly as possible by applying pending changes in parallel. This results in multiple threads waiting for modifications to conclude. Once these modifications are concluded, the threads become available and are applied to the underlying cache store.

This strategy is ideal for cache stores with low latency and low operational costs. An example of this is a local unshared file based cache store in which the cache store is local to the cache itself. Using this strategy the period of time where an inconsistency exists between the contents of the cache and the contents of the cache store is reduced to the shortest possible interval.

Report a bug

8.3.4. Unscheduled Write-Behind Strategy Library Configuration

The following is a sample configuration file for the Unscheduled Write-Behind strategy in JBoss Data Grid's Library Mode:

```
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:infinispan:config:5.0">
 <global />
 <default />
 <namedCache name="persistentCache">
  <loaders shared="false">
   <loader class="org.infinispan.loaders.file.FileCacheStore"
    fetchPersistentState="true"
    ignoreModifications="false"
    purgeOnStartup="false">
    <properties>
     <property name="location"
        value="${java.io.tmpdir}" />
    </properties>
    <async enabled="true" threadPoolSize="10" />
   </loader>
  </loaders>
 </namedCache>
</infinispan>
```

The write-behind configuration occurs specifically at the following line in the provided configuration:

```
<async enabled="true" threadPoolSize="10" />
```

Report a bug

8.3.5. Unscheduled Write-Behind Strategy Remote Configuration

Adding the **`write-behind`** element within the **`file-store`** element allows configuration for Unscheduled Write-Behind strategy for all cache stores.

The following is a sample configuration for Unscheduled Write-Behind strategy in JBoss Data Grid's Remote Client-Server mode:

```
<file-store passivation="true"
     relative-to="temp"
     path="nc"
```

```
      purge="true"
      shared="false">
  <write-behind flush-lock-timeout="1"
         modification-queue-size="1024"
         shutdown-timeout="25000"
         thread-pool-size="1"/>
</file-store>
```

Report a bug

CHAPTER 9. LOCKING

9.1. ABOUT OPTIMISTIC LOCKING

Optimistic locking allows multiple transactions to complete simultaneously by deferring lock acquisition to the transaction prepare time.

Optimistic mode assumes that multiple transactions can complete without conflict. It is ideal where there is little contention between multiple transactions running concurrently, as transactions can commit without waiting for other transaction locks to clear. With ***`writeSkewCheck`*** enabled, transactions in optimistic locking mode roll back if one or more conflicting modifications are made to the data before the transaction completes.

Report a bug

9.2. ABOUT PESSIMISTIC LOCKING

Pessimistic locking is also known as eager locking.

Pessimistic locking prevents more than one transaction being written to a key by enforcing cluster-wide locks on each write operation. Locks are only released once the transaction is completed either through committing or being rolled back.

Pessimistic mode is used where a high contention on keys is occurring, resulting in inefficiencies and unexpected roll back operations.

Report a bug

9.3. PESSIMISTIC LOCKING TYPES

JBoss Data Grid includes explicit pessimistic locking and implicit pessimistic locking:

- Explicit Pessimistic Locking, which uses the JBoss Data Grid Lock API to allow cache users to explicitly lock cache keys for the duration of a transaction. The Lock call attempts to obtain locks on specified cache keys across all nodes in a cluster. This attempt either fails or succeeds for all specified cache keys. All locks are released during the commit or rollback phase.
- Implicit Pessimistic Locking ensures that cache keys are locked in the background as they are accessed for modification operations. Using Implicit Pessimistic Locking causes JBoss Data Grid to check and ensure that cache keys are locked locally for each modification operation. Discovering unlocked cache keys causes JBoss Data Grid to request a cluster-wide lock to acquire a lock on the unlocked cache key.

Report a bug

9.4. EXPLICIT PESSIMISTIC LOCKING EXAMPLE

The following is an example of explicit pessimistic locking that depicts a transaction that runs on one of the cache nodes:

```
tx.begin()
cache.lock(K)
cache.put(K,V5)
```

```
tx.commit()
```

When the line **`cache.lock(K)`** executes, a cluster-wide lock is acquired on **`K`**.

When the line **`cache.put(K,V5)`** executes, it guarantees success.

When the line **`tx.commit()`** executes, the locks held for this process are released.

Report a bug

9.5. IMPLICIT PESSIMISTIC LOCKING EXAMPLE

An example of implicit pessimistic locking using a transaction that runs on one of the cache nodes is as follows:

```
tx.begin()
cache.put(K,V)
cache.put(K2,V2)
cache.put(K,V5)
tx.commit()
```

- When the line **`cache.put(K,V)`** executes, a cluster-wide lock is acquired on **`K`**.
- When the line **`cache.put(K2,V2)`** executes, a cluster-wide lock is acquired on **`K2`**.
- When the line **`cache.put(K,V5)`** executes, the lock acquisition is non operational because a cluster-wide lock for **`K`** has been previously acquired. The **`put`** operation will still occur.
- When the line **`tx.commit()`** executes, all locks held for this transaction are released.

Report a bug

9.6. CONFIGURE OPTIMISTIC AND PESSIMISTIC LOCKING

To configure a locking mode in JBoss Data Grid, use the ***`locking`*** parameter within the **`transaction`** element.

Optimistic Locking

Configure optimistic locking as follows:

```
<transaction locking="OPTIMISTIC" />
```

Pessimistic Locking

Configure pessimistic locking as follows:

```
<transaction locking="PESSIMISTIC" />
```

Report a bug

9.7. LOCKING OPERATIONS

9.7.1. About the LockManager

The **LockManager** component is responsible for locking an entry before a write process initiates. The **LockManager** uses a **LockContainer** to locate, hold and create locks. The two types of **LockContainers** generally used in such implementations are available. The first type offers support for lock striping while the second type supports one lock per entry.

See Also:

- Section 9.8, "Lock Striping"

Report a bug

9.7.2. About Lock Acquisition

JBoss Data Grid acquires remote locks lazily by default. The node running a transaction locally acquires the lock while other cluster nodes attempt to lock cache keys that are involved in a two phase prepare/commit phase. JBoss Data Grid can lock cache keys in a pessimistic manner either explicitly or implicitly.

Report a bug

9.7.3. About Concurrency Levels

In JBoss Data Grid, concurrency levels determine the size of each striped lock container. Additionally, concurrency levels tune all related JDK **ConcurrentHashMap** based collections, such as those internal to **DataContainers**.

Report a bug

9.8. LOCK STRIPING

9.8.1. About Lock Striping

Lock Striping allocates locks from a shared collection of (fixed size) locks in the cache. Lock allocation is based on the hash code for each entry's key. Lock Striping provides a highly scalable locking mechanism with fixed overhead. However, this is at the cost of potentially unrelated entries being blocked by the same lock.

Lock Striping is disabled as a default in JBoss Data Grid, which creates a new lock for each entry.

Report a bug

9.8.2. Configure Lock Striping

Lock striping in JBoss Data Grid can be enabled by configuring the ***striping*** element to **true**.

For example:

```
<locking isolation="REPEATABLE_READ"
  acquire-timeout="20000"
  concurrency-level="500"
  striping="true" />
```

Report a bug

9.8.3. Alternatives to Lock Striping

If JBoss Data Grid's lock striping remains disabled, a new lock is created for each entry. This alternative approach can provide greater concurrent throughput, but also results in additional memory usage, garbage collection churn, and other disadvantages.

Report a bug

9.8.4. Configure the Shared Lock Collection Size

In JBoss Data Grid, lock striping is disabled by default. The size of the shared lock collection used by lock striping can be tuned using the concurrency-level attribute of the **`locking`** configuration element.

Report a bug

9.9. ISOLATION LEVELS

9.9.1. About Isolation Levels

Isolation levels determine when readers can view a concurrent write. ***`READ_COMMITTED`*** and ***`REPEATABLE_READ`*** are the two isolation modes offered in JBoss Data Grid.

- **`READ_COMMITTED`**. This is the default isolation level because it is applicable to a wide variety of requirements.
- **`REPEATABLE_READ`**. This can be configured using the **`locking`** configuration element.

Report a bug

9.9.2. About READ_COMMITTED

`READ_COMMITTED` is one of two isolation modes available in JBoss Data Grid.

In JBoss Data Grid's ***`READ_COMMITTED`*** mode, write operations are made to copies of data rather than the data itself. A write operation blocks other data from being written, however writes do not block read operations. As a result, both ***`READ_COMMITTED`*** and ***`REPEATABLE_READ`*** modes permit read operations at any time, regardless of when write operations occur.

In ***`READ_COMMITTED`*** mode multiple reads of the same key within a transaction can return different results due to write operations modifying data between reads. This phenomenon is known as non-repeatable reads and is avoided in ***`REPEATABLE_READ`*** mode.

Report a bug

9.9.3. About REPEATABLE_READ

`REPEATABLE_READ` is one of two isolation modes available in JBoss Data Grid.

Traditionally, ***`REPEATABLE_READ`*** does not allow write operations while read operations are in progress, nor does it allow read operations when write operations occur. This prevents the "non-

repeatable read" phenomenon, which occurs when a single transaction has two read operations on the same row but the retrieved values differ (possibly due to a write operating modifying the value between the two read operations).

JBoss Data Grid's ***REPEATABLE_READ*** isolation mode preserves the value of a row before a modification occurs. As a result, the "non-repeatable read" phenomenon is avoided because a second read operation on the same row retrieves the preserved value rather than the new modified value. As a result, the two values retrieved by the two read operations will always match, even if a write operation occurs between the two reads.

Report a bug

CHAPTER 10. CACHE STORES AND CACHE LOADERS

10.1. ABOUT CACHE STORES

The cache store connects JBoss Data Grid to the persistent data store. The cache store is used to:

- fetch data from the data store when a copy is not in the cache.
- push modifications made to the data in cache back to the data store.

Caches that share the same cache manager can have different cache store configurations, as cache stores are associated with individual caches.

Report a bug

10.2. ABOUT FILE SYSTEM BASED CACHE STORES

JBoss Data Grid includes one file system based cache store: the **`FileCacheStore`**.

The **`FileCacheStore`** is a simple, file system based implementation.

Due to its limitations, **`FileCacheStore`** can be used in a limited capacity in production environments. It should not be used on shared file system (such as NFS and Windows shares) due to a lack of proper file locking, resulting in data corruption. Furthermore, file systems are not inherently transactional, resulting in file writing failures during the commit phase if the cache is used in a transactional context.

The **`FileCacheStore`** is ideal for testing usage and is not suited to use in highly concurrent, transactional or stress-based environments.

Report a bug

10.3. FILE CACHE STORE CONFIGURATION

The following is an example of a File Cache Store configuration:

```
<local-cache name="default">
    <file-store />
</local-cache>
```

- The ***name*** parameter of the **`local-cache`** attribute is used to specify a name for the cache.
- The **`file-store`** element specifies configuration information for the file cache store. Attributes for this element include the ***`relative-to`*** parameter used to define a named path, and the ***`path`*** parameter used to specify a directory within ***`relative-to`***.

Report a bug

10.4. REMOTE CACHE STORES

10.4.1. About Remote Cache Stores

The **`RemoteCacheStore`** is an implementation of the cache loader that stores data in a remote JBoss Data Grid cluster. The **`RemoteCacheStore`** uses the Hot Rod client-server architecture to

communicate with the remote cluster.

For remote cache stores, Hot Rod provides load balancing, fault tolerance and the ability to fine tune the connection between the **RemoteCacheStore** and the cluster.

Report a bug

10.4.2. Remote Cache Store Configuration (Remote Client-Server Mode)

The following is a sample remote cache store configuration for JBoss Data Grid's Remote Client-Server mode.

```
<remote-store cache="default"
       socket-timeout="60000"
              tcp-no-delay="true"
              fetch-state="false"
              passivation="true"
              preload="true"
              purge="false">
 <remote-server outbound-socket-binding="remote-store-hotrod-server" />
</remote-store>
```

Report a bug

10.4.3. Remote Cache Store Configuration (Library Mode)

Prerequisites:

Create and include a file named **hotrod.properties** in the relevant classpath.

Remote Cache Store Configuration

The following is a sample remote cache store configuration for JBoss Data Grid's Library mode.

```
<loaders shared="true"
  preload="true"
  purge="false">
 <loader class="org.infinispan.loaders.remote.RemoteCacheStore">
  <properties>
   <property name="remoteCacheName"
      value="default"/>
   <property name="hotRodClientPropertiesFile"
      value="hotrod.properties" />
  </properties>
 </loader>
</loaders>
```

IMPORTANT

Define the outbound socket for the remote cache store when using this configuration. The remote cache store property named **hotRodClientPropertiesFile** refers to the **hotrod.properties** file. This file must be defined for the Remote Cache Store to operate correctly.

Report a bug

10.4.4. Remote Cache Store Configuration Attributes

The following configuration elements are specific to the remote cache configuration.

The Remote-Store Element

The `remote-store` element specifies the configuration information for a remote cache store accessed using Hot Rod. Properties defined within the `remote-store` element are treated as Hot Rod client properties.

- The ***cache*** parameter specifies the name of the cache in use.
- The ***socket-timeout*** parameter specifies the socket timeout time for the remote cache.
- The ***tcp-no-delay*** parameter specifies whether TCP packets will be delayed and sent out in batches. Valid values for this parameter are `true` and `false`.
- The ***fetch-state*** parameter determines whether the persistent state is fetched when joining a cluster. Valid values for this parameter are `true` and `false`.
- The ***passivation*** parameter determines whether entries in the cache are passivated (`true`) or if the cache store retains a copy of the contents in memory (`false`).
- The ***preload*** parameter specifies whether to load entries into the cache during start up. Valid values for this parameter are `true` and `false`.
- The ***purge*** parameter specifies whether or not the cache store is purged when it is started. Valid values for this parameter are `true` and `false`.

The Remote-Server Element

The `remote-server` element provides information about the remote server used by the remote cache store.

- The ***outbound-socket-binding*** parameter specifies the outbound socket for the remote cache store.

IMPORTANT

Define the outbound socket for the remote cache store in the `standalone.xml` file as well to use the remote cache store.

Report a bug

10.4.5. The hotrod.properties File

To use a Remote Cache Store configuration, the hotrod.properties file must be created and included in the relevant classpath for a Remote Cache Store configuration.

The hotrod.properties file contains one or more properties. The most simple version of a working hotrod.properties file can contain the following:

```
infinispan.client.hotrod.server_list=remote-server:11222
```

-

Properties that can be included in hotrod.properties are:

`infinispan.client.hotrod.request_balancing_strategy`

For replicated (vs distributed) Hot Rod server clusters, the client balances requests to the servers according to this strategy.

The default value for this property is **`org.infinispan.client.hotrod.impl.transport.tcp.RoundRobinBalancingStrategy`**.

`infinispan.client.hotrod.server_list`

This is the initial list of Hot Rod servers to connect to, specified in the following format: host1:port1;host2:port2... At least one host:port must be specified.

The default value for this property is **`127.0.0.1:11222`**.

`infinispan.client.hotrod.force_return_values`

Whether or not to implicitly Flag.FORCE_RETURN_VALUE for all calls.

The default value for this property is **`false`**.

`infinispan.client.hotrod.tcp_no_delay`

Affects TCP NODELAY on the TCP stack.

The default value for this property is **`true`**.

`infinispan.client.hotrod.ping_on_startup`

If true, a ping request is sent to a back end server in order to fetch cluster's topology.

The default value for this property is **`true`**.

`infinispan.client.hotrod.transport_factory`

Controls which transport will be used. Currently only the TcpTransport is supported.

The default value for this property is **`org.infinispan.client.hotrod.impl.transport.tcp.TcpTransportFactory`**.

`infinispan.client.hotrod.marshaller`

Allows you to specify a custom Marshaller implementation to serialize and deserialize user objects.

The default value for this property is **`org.infinispan.marshall.jboss.GenericJBossMarshaller`**.

`infinispan.client.hotrod.async_executor_factory`

Allows you to specify a custom asynchronous executor for async calls.

The default value for this property is **`org.infinispan.client.hotrod.impl.async.DefaultAsyncExecutorFactory`**.

`infinispan.client.hotrod.default_executor_factory.pool_size`

If the default executor is used, this configures the number of threads to initialize the executor with.

The default value for this property is **10**.

infinispan.client.hotrod.default_executor_factory.queue_size

If the default executor is used, this configures the queue size to initialize the executor with.

The default value for this property is **100000**.

infinispan.client.hotrod.hash_function_impl.1

This specifies the version of the hash function and consistent hash algorithm in use, and is closely tied with the Hot Rod server version used.

The default value for this property is the **Hash function specified by the server in the responses as indicated in ConsistentHashFactory**.

infinispan.client.hotrod.key_size_estimate

This hint allows sizing of byte buffers when serializing and deserializing keys, to minimize array resizing.

The default value for this property is **64**.

infinispan.client.hotrod.value_size_estimate

This hint allows sizing of byte buffers when serializing and deserializing values, to minimize array resizing.

The default value for this property is **512**.

infinispan.client.hotrod.socket_timeout

This property defines the maximum socket read timeout before giving up waiting for bytes from the server.

The default value for this property is **60000 (equals 60 seconds)**.

infinispan.client.hotrod.protocol_version

This property defines the protocol version that this client should use. Other valid values include 1.0.

The default value for this property is **1.1**.

infinispan.client.hotrod.connect_timeout

This property defines the maximum socket connect timeout before giving up connecting to the server.

The default value for this property is **60000 (equals 60 seconds)**.

Report a bug

10.4.6. Define the Outbound Socket for the Remote Cache Store

The Hot Rod server used by the remote cache store is defined using the **outbound-socket-binding** element in a **standalone.xml** file.

An example of this configuration in the **standalone.xml** file is as follows:

```
<outbound-socket-binding name="remote-store-hotrod-server">
 <remote-destination host="127.0.0.1"
       port="11322"/>
</outbound-socket-binding>
```

Report a bug

10.5. JDBC BASED CACHE STORES

10.5.1. About JDBC Based Cache Stores

JBoss Data Grid offers several cache stores for use with common data storage formats. JDBC based cache stores are used with any cache store that exposes a JDBC driver. JBoss Data Grid offers the following JDBC based cache stores depending on the key to be persisted:

- **JdbcBinaryCacheStore**.
- **JdbcStringBasedCacheStore**.
- **JdbcMixedCacheStore**.

Report a bug

10.5.2. JDBC Cache Selection

The **JdbcStringBasedCacheStore** is ideal for when you control the key types because it offers better throughput when under stress.

The **JdbcStringBasedCacheStore** can only be used when you can write a **Key2StringMapper** to map keys to string objects. If the **JdbcStringBasedCacheStore** cannot be used, **JdbcBinaryCacheStore** or **JdbcMixedCacheStore** can be used instead. The **JdbcMixedCacheStore** is more appropriate when **JdbcStringBasedCacheStore** is handling the majority of the keys, but some keys cannot be converted using **Key2StringMapper**.

Report a bug

10.5.3. JdbcBinaryCacheStores

10.5.3.1. About JdbcBinaryCacheStore

The **JdbcBinaryCacheStore** supports all key types. It stores all keys with the same hash value (**hashCode** method on the key) in the same table row/blob. The hash value common to the included keys is set as the primary key for the table row/blob. As a result of this hash value, **JdbcBinaryCacheStore** offers excellent flexibility but at the cost of concurrency and throughput.

As an example, if three keys (**k1**, **k2** and **k3**) have the same hash code, they are stored in the same table row. If three different threads attempt to concurrently update **k1**, **k2** and **k3**, they must do it sequentially because all three keys share the same row and therefore cannot be simultaneously

updated.

Report a bug

10.5.3.2. JdbcBinaryCacheStore Remote Configuration with Passivation Enabled

The following is a configuration for **JdbcBinaryCacheStore** using JBoss Data Grid's Remote Client-Server mode with Passivation enabled.

```
<subsystem xmlns="urn:jboss:domain:infinispan:1.2" default-cache-
container="default">
 <cache-container name="default"
    default-cache="default"
    listener-executor="infinispan-listener"
    start="EAGER">
  <local-cache name="default"
        start="EAGER"
        batching="false"
        indexing="NONE">
   <locking isolation="REPEATABLE_READ"
     acquire-timeout="20000"
     concurrency-level="500"
     striping="false" />
   <transaction mode="NONE" />
   <eviction strategy="LRU"
      max-entries="2" />
   <binary-keyed-jdbc-store datasource="java:jboss/datasources/JdbcDS"
       passivation="true"
       preload="false"
       purge="false">
                          <property
name="databaseType">${database.type}</property>
                          <binary-keyed-table prefix="JDG">
                            <id-column name="id"
         type="${id.column.type}"/>
                            <data-column name="datum"
           type="${data.column.type}"/>
                            <timestamp-column name="version"
         type="${timestamp.column.type}"/>
                          </binary-keyed-table>
                      </binary-keyed-jdbc-store>
                </local-cache>
 </cache-container>
</subsystem>
```

Report a bug

10.5.3.3. JdbcBinaryCacheStore Remote Configuration with Passivation Disabled

The following is a configuration for **JdbcBinaryCacheStore** using JBoss Data Grid's Remote Client-Server mode with Passivation disabled.

```
<subsystem xmlns="urn:jboss:domain:infinispan:1.2" default-cache-
container="default">
```

```
 <cache-container name="default"
    default-cache="default"
    listener-executor="infinispan-listener"
    start="EAGER">
  <local-cache name="default"
        start="EAGER"
                          batching="false"
                          indexing="NONE">
                  <locking isolation="REPEATABLE_READ"
     acquire-timeout="20000"
     concurrency-level="500"
     striping="false" />
   <transaction mode="NONE" />
   <binary-keyed-jdbc-store datasource="java:jboss/datasources/JdbcDS"
       passivation="false"
       preload="true"
       purge="false">
                         <property
name="databaseType">${database.type}</property>
                          <binary-keyed-table prefix="JDG">
                               <id-column name="id"
         type="${id.column.type}"/>
     <data-column name="datum"
           type="${data.column.type}"/>
     <timestamp-column name="version"
           type="${timestamp.column.type}"/>
    </binary-keyed-table>
                      </binary-keyed-jdbc-store>
  </local-cache>
 </cache-container>
</subsystem>
```

Report a bug

10.5.3.4. JdbcBinaryCacheStore Remote Configuration Attributes

The following is a list of elements and parameters used to configure a **JdbcBinaryCacheStore** in JBoss Data Grid's Remote Client-Server mode.

The cache-container Element

The **cache-container** element specifies information about the cache container using the following parameters:

- The ***name*** parameter defines the name of the cache container.
- The ***default-cache*** parameter defines the name of the default cache used with the cache container.
- The ***listener-executor*** defines the executor used for asynchronous cache listener notifications.
- The ***start*** parameter indicates where the cache container starts, i.e. whether it will start lazily when requested or when the server starts up. Valid values for this parameter are **EAGER** and **LAZY**.

The local-cache Element

The `local-cache` element specifies information about the local cache used with the cache container using the following parameters:

- The ***name*** parameter specifies the name of the local cache to use.
- The ***start*** parameter indicates where the cache starts, i.e. whether it will start lazily when requested or when the server starts up. Valid values for this parameter are **EAGER** and **LAZY**.
- The ***batching*** parameter specifies whether batching is enabled for the local cache.
- The ***indexing*** parameter specifies the type of indexing used for the local cache. Valid values for this parameter are **NONE**, **LOCAL** and **ALL**.

The locking Element

The `locking` element details the locking configuration for the local cache.

- The ***isolation*** parameter defines the isolation level used for the local cache. Valid values for this parameter are **REPEATABLE_READ** and **READ_COMMITTED**.
- The ***acquire-timeout*** parameter specifies the number of milliseconds after which an acquire operation will time out.
- The ***concurrency-level*** parameter defines the number of lock stripes used by the LockManager.
- The ***striping*** parameter specifies whether lock striping will be used for the local cache.

The transaction Element

The `transaction` element specifies transaction related settings for the local cache.

- The ***mode*** parameter specifies the transaction mode used for the local cache. Valid values for this parameter are **NONE**, **NON_XA** (does not use XAResource), **NON_DURABLE_XA** (uses XAResource without recovery) and **FULL_XA** (used XAResource with recovery).

The eviction Element

The `eviction` element specifies eviction configuration information for the local cache.

- The ***strategy*** parameter specifies the eviction strategy or algorithm used. Valid values for this parameter include **NONE**, **FIFO**, **LRU**, **UNORDERED** and **LIRS**.
- The ***max-entries*** parameter specifies the maximum number of entries in a cache instance.

The binary-keyed-jdbc-store Element

The `binary-keyed-jdbc-store` element specifies the configuration for a binary keyed cache JDBC store.

- The ***datasource*** parameter defines the name of a JNDI for the datasource.
- The ***passivation*** parameter determines whether entries in the cache are passivated (`true`) or if the cache store retains a copy of the contents in memory (`false`).

- The ***preload*** parameter specifies whether to load entries into the cache during start up. Valid values for this parameter are **true** and **false**.
- The ***purge*** parameter specifies whether or not the cache store is purged when it is started. Valid values for this parameter are **true** and **false**.

The property Element

The **property** element contains information about properties related to the cache store.

- The ***name*** parameter specifies the name of the cache store.
- The value *${database.type}* must be replaced by a valid database type value, such as **DB2_390**, **SQL_SERVER**, **MYSQL**, **ORACLE**, **POSTGRES** or **SYBASE**.

The binary-keyed-table Element

The **binary-keyed-table** element specifies information about the database table used to store binary cache entries.

- The ***prefix*** parameter specifies a prefix string for the database table name.

The id-column Element

The **id-column** element specifies information about a database column that holds cache entry IDs.

- The ***name*** parameter specifies the name of the database column.
- The ***type*** parameter specifies the type of the database column.

The data-column Element

The **data-column** element contains information about a database column that holds cache entry data.

- The ***name*** parameter specifies the name of the database column.
- The ***type*** parameter specifies the type of the database column.

The timestamp-column Element

The **timestamp-column** element specifies information about the database column that holds cache entry timestamps.

- The ***name*** parameter specifies the name of the database column.
- The ***type*** parameter specifies the type of the database column.

Report a bug

10.5.3.5. JdbcBinaryCacheStore Library Mode Configuration

The following is a sample configuration for the **JdbcBinaryCacheStore**:

```
<loaders>
   <loader class="org.infinispan.loaders.jdbc.binary.JdbcBinaryCacheStore"
fetchPersistentState="false"ignoreModifications="false"
purgeOnStartup="false">
      <properties>
```

```
        <property name="bucketTableNamePrefix"
value="ISPN_BUCKET_TABLE"/>
        <property name="idColumnName" value="ID_COLUMN"/>
        <property name="dataColumnName" value="DATA_COLUMN"/>
        <property name="timestampColumnName" value="TIMESTAMP_COLUMN"/>
        <property name="timestampColumnType" value="BIGINT"/>
        <property name="connectionFactoryClass"
value="org.infinispan.loaders.jdbc.connectionfactory.PooledConnectionFacto
ry"/>
        <property name="connectionUrl"
value="jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1"/>
        <property name="userName" value="sa"/>
        <property name="driverClass" value="org.h2.Driver"/>
        <property name="idColumnType" value="VARCHAR(255)"/>
        <property name="dataColumnType" value="BINARY"/>
        <property name="dropTableOnExit" value="true"/>
        <property name="createTableOnStart" value="true"/>
      </properties>
   </loader>
</loaders>
```

Report a bug

10.5.4. JdbcStringBasedCacheStores

10.5.4.1. About JdbcStringBasedCacheStore

The **JdbcStringBasedCacheStore** stores each entry its own row in the table, instead of grouping multiple entries into each row, resulting in increased throughput under a concurrent load. It also uses a (pluggable) bijection that maps each key to a **String** object. The **Key2StringMapper** interface defines the bijection.

JBoss Data Grid includes a default implementation called **DefaultTwoWayKey2StringMapper** that handles primitive types.

Report a bug

10.5.4.2. JdbcStringBasedCacheStore Remote Configuration for Multiple Nodes

The following is a configuration for the **JdbcStringBasedCacheStore** in JBoss Data Grid's Remote Client-Server mode. This configuration is used when multiple nodes must be used.

```
<subsystem xmlns="urn:jboss:domain:infinispan:1.2" default-cache-
container="default">
 <cache-container name="default"
    default-cache="memcachedCache"
    listener-executor="infinispan-listener"
    start="EAGER">
  <transport stack="${stack}"
      executor="infinispan-transport"
      lock-timeout="240000"/>
               <replicated-cache name="memcachedCache"
                         start="EAGER"
                        mode="SYNC"
```

```
                                    batching="false"
                            indexing="NONE"
                          remote-timeout="60000">
   <locking isolation="REPEATABLE_READ"
                            acquire-timeout="30000"
                            concurrency-level="1000"
                            striping="false" />
                  <transaction mode="NONE" />
                  <state-transfer enabled="true"
                            timeout="60000" />
                  <string-keyed-jdbc-store
datasource="java:jboss/datasources/JdbcDS"
                             fetch-state="true"
                             passivation="false"
                              preload="false"
                             purge="false"
                             shared="false"
                             singleton="true">
    <property name="databaseType">${database.type}</property>
                   <string-keyed-table prefix="JDG">
                               <id-column name="id"
             type="${id.column.type}"/>
                              <data-column name="datum"
          type="${data.column.type}"/>
                               <timestamp-column name="version"
            type="${timestamp.column.type}"/>
    </string-keyed-table>
   </string-keyed-jdbc-store>
                 </replicated-cache>
 </cache-container>
</subsystem>
```

Report a bug

10.5.4.3. JdbcStringBasedCacheStore Remote Configuration with Passivation Enabled

The following is a sample **JdbcStringBasedCacheStore** for JBoss Data Grid's Remote Client-Server mode with Passivation enabled.

```
<subsystem xmlns="urn:jboss:domain:infinispan:1.2" default-cache-
container="default">
 <cache-container name="default"
    default-cache="memcachedCache"
    listener-executor="infinispan-listener"
    start="EAGER">
  <local-cache name="memcachedCache"
                      start="EAGER"
                       batching="false"
                         indexing="NONE">
   <locking isolation="REPEATABLE_READ"
                            acquire-timeout="20000"
                             concurrency-level="500"
                              striping="false" />
   <transaction mode="NONE" />
          <eviction strategy="LRU"
```

```
                          max-entries="2" />
            <string-keyed-jdbc-store
datasource="java:jboss/datasources/JdbcDS"
       passivation="true"
       preload="false"
       purge="false">
                          <property
name="databaseType">${database.type}</property>
                          <string-keyed-table prefix="JDG">
                               <id-column name="id"
         type="${id.column.type}"/>
     <data-column name="datum"
           type="${data.column.type}"/>
     <timestamp-column name="version"
         type="${timestamp.column.type}"/>
                          </string-keyed-table>
   </string-keyed-jdbc-store>
  </local-cache>
 </cache-container>
</subsystem>
```

Report a bug

10.5.4.4. JdbcStringBasedCacheStore Remote Configuration with Passivation Disabled

The following is a configuration for the **JdbcStringBasedCacheStore** in JBoss Data Grid's Remote Client-Server mode with Passivation disabled.

```
<subsystem xmlns="urn:jboss:domain:infinispan:1.2" default-cache-
container="default">
 <cache-container name="default"
    default-cache="memcachedCache"
    listener-executor="infinispan-listener"
    start="EAGER">
  <local-cache name="memcachedCache"
                      start="EAGER"
                       batching="false"
                        indexing="NONE">
   <locking isolation="REPEATABLE_READ"
                         acquire-timeout="20000"
                         concurrency-level="500"
                          striping="false" />
                     <transaction mode="NONE" />
                     <string-keyed-jdbc-store
datasource="java:jboss/datasources/JdbcDS"
       passivation="false"
       preload="true"
       purge="false">
                          <property
name="databaseType">${database.type}</property>
                          <string-keyed-table prefix="JDG">
                               <id-column name="id"
         type="${id.column.type}"/>
                               <data-column name="datum"
```

```
            type="${data.column.type}"/>
                                <timestamp-column name="version"
          type="${timestamp.column.type}"/>
                            </string-keyed-table>
                        </string-keyed-jdbc-store>
                    </local-cache>
  </cache-container>
</subsystem>
```

Report a bug

10.5.4.5. JdbcStringBasedCacheStore Remote Configuration Attributes

The following is a list of attributes and parameters used to configure a **JdbcStringBasedCacheStore** in JBoss Data Grid's Remote Client-Server mode.

The cache-container Element

The **cache-container** element specifies information about the cache container.

- The ***name*** parameter defines the name of the cache container.
- The ***default-cache*** parameter defines the name of the default cache used with the cache container.
- The ***listener-executor*** defines the executor used for asynchronous cache listener notifications.
- The ***start*** parameter indicates where the cache container starts, i.e. whether it will start lazily when requested or when the server starts up. Valid values for this parameter are **EAGER** and **LAZY**.

The local-cache Element

The **local-cache** element specifies information about the local cache used with the cache container.

- The ***name*** parameter specifies the name of the local cache to use.
- The ***start*** parameter indicates where the cache starts, i.e. whether it will start lazily when requested or when the server starts up. Valid values for this parameter are **EAGER** and **LAZY**.
- The ***batching*** parameter specifies whether batching is enabled for the local cache.
- The ***indexing*** parameter specifies the type of indexing used for the local cache. Valid values for this parameter are **NONE**, **LOCAL** and **ALL**.

The transport Element

The **transport** element contains a description of the transport used by the cache container.

- The ***stack*** parameter specified the JGroups stack used for the transport.
- The ***executor*** parameter specifies the executor used for the transport.
- The ***lock-timeout*** parameter specifies the timeout value for locks for the transport.

The replicated-cache Element

The **replicated-cache** element specifies information about the replicated cache in use.

- The ***name*** parameter specifies the name of the replicated cache.
- The ***start*** parameter specifies whether the cache starts up on demand or immediately. Valid values for this parameter are **EAGER** and **LAZY**.
- The ***mode*** parameter specifies the cache mode for the replicated cache. Valid values for this parameter are **SYNC** and **ASYNC**.
- The ***batching*** parameter specifies whether batching can be used with the replicated cache.
- The ***indexing*** parameter specifies whether entries added to the cache are indexed. If enabled, indexes update as entries change or are removed from the replicated cache.
- The ***remote-timeout*** parameter specifies the time period a remote call waits for acknowledgment. After the specified interval, the remote call aborts and an exception is thrown. This parameter is used in conjunction with the **ASYNC** mode parameter.

The locking Element

The **locking** element details the locking configuration for the local cache.

- The ***isolation*** parameter defines the isolation level used for the local cache. Valid values for this parameter are **REPEATABLE_READ** and **READ_COMMITTED**.
- The ***acquire-timeout*** parameter specifies the number of milliseconds after which an acquire operation will time out.
- The ***concurrency-level*** parameter defines the number of lock stripes used by the LockManager.
- The ***striping*** parameter specifies whether lock striping will be used for the local cache.

The transaction Element

The **transaction** element specifies transaction related settings for the local cache.

- The ***mode*** parameter specifies the transaction mode used for the local cache. Valid values for this parameter are **NONE**, **NON_XA** (does not use XAResource), **NON_DURABLE_XA** (uses XAResource without recovery) and **FULL_XA** (used XAResource with recovery).

The eviction Element

The **eviction** element specifies eviction configuration information for the local cache.

- The ***strategy*** parameter specifies the eviction strategy or algorithm used. Valid values for this parameter include **NONE**, **FIFO**, **LRU**, **UNORDERED** and **LIRS**.
- The ***max-entries*** parameter specifies the maximum number of entries in a cache instance.

The string-keyed-jdbc-store Element

The **string-keyed-jdbc-store** element specifies the configuration for a string based keyed cache JDBC store.

- The ***datasource*** parameter defines the name of a JNDI for the datasource.

- The ***passivation*** parameter determines whether entries in the cache are passivated (**true**) or if the cache store retains a copy of the contents in memory (**false**).
- The ***preload*** parameter specifies whether to load entries into the cache during start up. Valid values for this parameter are **true** and **false**.
- The ***purge*** parameter specifies whether or not the cache store is purged when it is started. Valid values for this parameter are **true** and **false**.
- The ***shared*** parameter is used when multiple cache instances share a cache store. This parameter can be set to prevent multiple cache instances writing the same modification multiple times. Valid values for this parameter are **ENABLED** and **DISABLED**.
- The ***singleton*** parameter enables a singleton store that is used if a cluster interacts with the underlying store.

The property Element

The **property** element contains information about properties related to the cache store.

- The ***name*** parameter specifies the name of the cache store.
- The value *${database.type}* must be replaced by a valid database type value, such as **DB2_390**, **SQL_SERVER**, **MYSQL**, **ORACLE**, **POSTGRES** or **SYBASE**.

The string-keyed-table Element

The **string-keyed-table** element specifies information about the database table used to store string based cache entries.

- The ***prefix*** parameter specifies a prefix string for the database table name.

The id-column Element

The **id-column** element specifies information about a database column that holds cache entry IDs.

- The ***name*** parameter specifies the name of the database column.
- The ***type*** parameter specifies the type of the database column.

The data-column Element

The **data-column** element contains information about a database column that holds cache entry data.

- The ***name*** parameter specifies the name of the database column.
- The ***type*** parameter specifies the type of the database column.

The timestamp-column Element

The **timestamp-column** element specifies information about the database column that holds cache entry timestamps.

- The ***name*** parameter specifies the name of the database column.
- The ***type*** parameter specifies the type of the database column.

Report a bug

10.5.4.6. JdbcStringBasedCacheStore Library Mode Configuration

The following is a sample configuration for the **JdbcStringBasedCacheStore**:

```
<loaders>

   <loader
class="org.infinispan.loaders.jdbc.stringbased.JdbcStringBasedCacheStore"
fetchPersistentState="false" ignoreModifications="false"
purgeOnStartup="false">
      <properties>
         <property name="stringsTableNamePrefix"
value="ISPN_STRING_TABLE"/>
         <property name="idColumnName" value="ID_COLUMN"/>
         <property name="dataColumnName" value="DATA_COLUMN"/>
         <property name="timestampColumnName" value="TIMESTAMP_COLUMN"/>
         <property name="timestampColumnType" value="BIGINT"/>
         <property name="connectionFactoryClass"
value="org.infinispan.loaders.jdbc.connectionfactory.PooledConnectionFacto
ry"/>
         <property name="connectionUrl"
value="jdbc:h2:mem:string_based_db;DB_CLOSE_DELAY=-1"/>
         <property name="userName" value="sa"/>
         <property name="driverClass" value="org.h2.Driver"/>
         <property name="idColumnType" value="VARCHAR(255)"/>
         <property name="dataColumnType" value="BINARY"/>
         <property name="dropTableOnExit" value="true"/>
         <property name="createTableOnStart" value="true"/>
      </properties>
   </loader>
</loaders>
```

Report a bug

10.5.5. JdbcMixedCacheStore

10.5.5.1. About JdbcMixedCacheStore

The **JdbcMixedCacheStore** is a hybrid implementation that delegates keys based on their type to either the **JdbcBinaryCacheStore** or **JdbcStringBasedCacheStore**.

Report a bug

10.5.5.2. JdbcMixedCacheStore Remote Configuration with Passivation Enabled

The following is a configuration for a **JdbcMixedCacheStore** for JBoss Data Grid's Remote Client-Server mode with Passivation enabled.

```
<subsystem xmlns="urn:jboss:domain:infinispan:1.2" default-cache-
container="default">
 <cache-container name="default"
    default-cache="memcachedCache"
    listener-executor="infinispan-listener"
```

```
    start="EAGER">
  <local-cache name="memcachedCache"
                      start="EAGER"
                       batching="false"
                        indexing="NONE">
   <locking isolation="REPEATABLE_READ"
                          acquire-timeout="20000"
                          concurrency-level="500"
                           striping="false" />
                     <transaction mode="NONE" />
                     <eviction strategy="LRU"
                           max-entries="2" />
            <mixed-keyed-jdbc-store
datasource="java:jboss/datasources/JdbcDS"
      passivation="true"
      preload="false"
      purge="false">
    <property name="databaseType">${database.type}</property>
                          <binary-keyed-table prefix="MIX_BKT2">
                               <id-column name="id"
        type="${id.column.type}"/>
     <data-column name="datum"
           type="${data.column.type}"/>
     <timestamp-column name="version"
        type="${timestamp.column.type}"/>
                          </binary-keyed-table>
                          <string-keyed-table prefix="MIX_STR2">
                               <id-column name="id"
        type="${id.column.type}"/>
     <data-column name="datum"
           type="${data.column.type}"/>
     <timestamp-column name="version"
        type="${timestamp.column.type}"/>
     </string-keyed-table>
                     </mixed-keyed-jdbc-store>
  </local-cache>
 </cache-container>
</subsystem>
```

Report a bug

10.5.5.3. JdbcMixedCacheStore Remote Configuration with Passivation Disabled

The following is a configuration for **JdbcMixedCacheStore** using JBoss Data Grid's Remote Client-Server mode with Passivation disabled.

```
<subsystem xmlns="urn:jboss:domain:infinispan:1.2" default-cache-
container="default">
 <cache-container name="default"
    default-cache="memcachedCache"
    listener-executor="infinispan-listener"
    start="EAGER">
  <local-cache name="memcachedCache"
                          start="EAGER"
```

```
                        batching="false"
                        indexing="NONE">
                    <locking isolation="REPEATABLE_READ"
                        acquire-timeout="20000"
                        concurrency-level="500"
                        striping="false" />
                    <transaction mode="NONE" />
                    <mixed-keyed-jdbc-store
datasource="java:jboss/datasources/JdbcDS"
      passivation="false"
      preload="true"
      purge="false">
                        <property
name="databaseType">${database.type}</property>
                        <binary-keyed-table prefix="MIX_BKT">
                         <id-column name="id"
         type="${id.column.type}"/>
                            <data-column name="datum"
           type="${data.column.type}"/>
                            <timestamp-column name="version"
         type="${timestamp.column.type}"/>
                        </binary-keyed-table>
                        <string-keyed-table prefix="MIX_STR">
                            <id-column name="id"
         type="${id.column.type}"/>
                            <data-column name="datum"
           type="${data.column.type}"/>
                            <timestamp-column name="version"
         type="${timestamp.column.type}"/>
                        </string-keyed-table>
                    </mixed-keyed-jdbc-store>
  </local-cache>
 </cache-container>
</subsystem>
```

Report a bug

10.5.5.4. JdbcMixedCacheStore Remote Configuration Attributes

The following is a list of elements and parameters used to configure a **JdbcMixedCacheStore** in JBoss Data Grid's Remote Client-Server mode.

The cache-container Element

The **cache-container** element specifies information about the cache container using the following parameters:

- The ***name*** parameter defines the name of the cache container.
- The ***default-cache*** parameter defines the name of the default cache used with the cache container.
- The ***listener-executor*** defines the executor used for asynchronous cache listener notifications.

- The ***start*** parameter indicates where the cache container starts, i.e. whether it will start lazily when requested or when the server starts up. Valid values for this parameter are **EAGER** and **LAZY**.

The local-cache Element

The **local-cache** element specifies information about the local cache used with the cache container using the following parameters:

- The ***name*** parameter specifies the name of the local cache to use.
- The ***start*** parameter indicates where the cache starts, i.e. whether it will start lazily when requested or when the server starts up. Valid values for this parameter are **EAGER** and **LAZY**.
- The ***batching*** parameter specifies whether batching is enabled for the local cache.
- The ***indexing*** parameter specifies the type of indexing used for the local cache. Valid values for this parameter are **NONE**, **LOCAL** and **ALL**.

The locking Element

The **locking** element details the locking configuration for the local cache.

- The ***isolation*** parameter defines the isolation level used for the local cache. Valid values for this parameter are **REPEATABLE_READ** and **READ_COMMITTED**.
- The ***acquire-timeout*** parameter specifies the number of milliseconds after which an acquire operation will time out.
- The ***concurrency-level*** parameter defines the number of lock stripes used by the LockManager.
- The ***striping*** parameter specifies whether lock striping will be used for the local cache.

The transaction Element

The **transaction** element specifies transaction related settings for the local cache.

- The ***mode*** parameter specifies the transaction mode used for the local cache. Valid values for this parameter are **NONE**, **NON_XA** (does not use XAResource), **NON_DURABLE_XA** (uses XAResource without recovery) and **FULL_XA** (used XAResource with recovery).

The eviction Element

The **eviction** element specifies eviction configuration information for the local cache.

- The ***strategy*** parameter specifies the eviction strategy or algorithm used. Valid values for this parameter include **NONE**, **FIFO**, **LRU**, **UNORDERED** and **LIRS**.
- The ***max-entries*** parameter specifies the maximum number of entries in a cache instance.

The mixed-keyed-jdbc-store Element

The **mixed-keyed-jdbc-store** element specifies the configuration for a mixed keyed cache JDBC store.

- The ***datasource*** parameter defines the name of a JNDI for the datasource.

- The ***passivation*** parameter determines whether entries in the cache are passivated (**true**) or if the cache store retains a copy of the contents in memory (**false**).
- The ***preload*** parameter specifies whether to load entries into the cache during start up. Valid values for this parameter are **true** and **false**.
- The ***purge*** parameter specifies whether or not the cache store is purged when it is started. Valid values for this parameter are **true** and **false**.

The property Element

The **property** element contains information about properties related to the cache store.

- The ***name*** parameter specifies the name of the cache store.
- The value *${database.type}* must be replaced by a valid database type value, such as **DB2_390**, **SQL_SERVER**, **MYSQL**, **ORACLE**, **POSTGRES** or **SYBASE**.

The mixed-keyed-table Element

The **mixed-keyed-table** element specifies information about the database table used to store mixed cache entries.

- The ***prefix*** parameter specifies a prefix string for the database table name.

The string-keyed-table Element

The **string-keyed-table** element specifies information about the database table used to store string based cache entries.

- The ***prefix*** parameter specifies a prefix string for the database table name.

The id-column Element

The **id-column** element specifies information about a database column that holds cache entry IDs.

- The ***name*** parameter specifies the name of the database column.
- The ***type*** parameter specifies the type of the database column.

The data-column Element

The **data-column** element contains information about a database column that holds cache entry data.

- The ***name*** parameter specifies the name of the database column.
- The ***type*** parameter specifies the type of the database column.

The timestamp-column Element

The **timestamp-column** element specifies information about the database column that holds cache entry timestamps.

- The ***name*** parameter specifies the name of the database column.
- The ***type*** parameter specifies the type of the database column.

Report a bug

10.5.5.5. JdbcMixedCacheStore Library Mode Configuration

The following is a sample configuration for the **JdbcMixedCacheStore**:

```
<loaders>
   <loader class="org.infinispan.loaders.jdbc.mixed.JdbcMixedCacheStore"
    fetchPersistentState="false"
           ignoreModifications="false"
    purgeOnStartup="false">
      <properties>
         <property name="tableNamePrefixForStrings"
value="ISPN_MIXED_STR_TABLE"/>
         <property name="tableNamePrefixForBinary"
value="ISPN_MIXED_BINARY_TABLE"/>
         <property name="idColumnNameForStrings" value="ID_COLUMN"/>
         <property name="idColumnNameForBinary" value="ID_COLUMN"/>
         <property name="dataColumnNameForStrings" value="DATA_COLUMN"/>
         <property name="dataColumnNameForBinary" value="DATA_COLUMN"/>
         <property name="timestampColumnNameForStrings"
value="TIMESTAMP_COLUMN"/>
         <property name="timestampColumnNameForBinary"
value="TIMESTAMP_COLUMN"/>
         <property name="timestampColumnTypeForStrings" value="BIGINT"/>
         <property name="timestampColumnTypeForBinary" value="BIGINT"/>
         <property name="connectionFactoryClass"
value="org.infinispan.loaders.jdbc.connectionfactory.PooledConnectionFacto
ry"/>
         <property name="connectionUrl"
value="jdbc:h2:mem:infinispan_mixed_cs;DB_CLOSE_DELAY=-1"/>
         <property name="userName" value="sa"/>
         <property name="driverClass" value="org.h2.Driver"/>
         <property name="idColumnTypeForStrings" value="VARCHAR(255)"/>
         <property name="idColumnTypeForBinary" value="VARCHAR(255)"/>
         <property name="dataColumnTypeForStrings" value="BINARY"/>
         <property name="dataColumnTypeForBinary" value="BINARY"/>
         <property name="dropTableOnExitForStrings" value="false"/>
         <property name="dropTableOnExitForBinary" value="false"/>
         <property name="createTableOnStartForStrings" value="true"/>
         <property name="createTableOnStartForBinary" value="true"/>
         <property name="createTableOnStartForStrings" value="true"/>
         <property name="createTableOnStartForBinary" value="true"/>
      </properties>
   </loader>
</loaders>
```

Report a bug

10.5.6. Custom Cache Stores

10.5.6.1. About Custom Cache Stores

Custom cache stores are a customized implementation of JBoss Data Grid cache stores.

Report a bug

10.5.6.2. Custom Cache Store Configuration (Remote Mode)

The following is a sample configuration for a custom cache store in JBoss Data Grid's Remote Client-Server mode:

```
<local-cache name="default">
     <store class="my.package.CustomCacheStore">
         <properties>
             <property name="customStoreProperty" value="10" />
         </properties>
     </store>
</local-cache>
```

IMPORTANT

To allow JBoss Data Grid to locate the defined class, create a module using the module of another (relevant) cache store as a template and add it to the **`org.jboss.as.clustering.infinispan`** module dependencies.

Report a bug

10.5.6.3. Custom Cache Store Configuration (Library Mode)

The following is a sample configuration for a custom cache store in JBoss Data Grid's Library mode:

```
<loaders shared="true"
  preload="true"
  purge="false">
 <loader class="org.infinispan..custom.CustomCacheStore">
  <properties>
   <property name="CustomCacheName"
      value="default"/>
  </properties>
 </loader>
</loaders>
```

Report a bug

10.6. FREQUENTLY ASKED QUESTIONS

10.6.1. About Asynchronous Cache Store Modifications

In JBoss Data Grid, modifications to asynchronous cache stores are coalesced or aggregated for the interval that is being applied by the modification thread. This ensures that multiple modifications for the same key are detected, only the last state of the key is applied. This improves efficiency by reducing the number of calls to the cache store.

Report a bug

10.7. CACHE STORE TROUBLESHOOTING

10.7.1. IOExceptions with JdbcStringBasedCacheStore

An IOException **`Unsupported protocol version 48`** error when using **`JdbcStringBasedCacheStore`** indicates that your data column type is set to **`VARCHAR`**, **`CLOB`** or something similar instead of the correct type, **`BLOB`** or **`VARBINARY`**. Despite its name, **`JdbcStringBasedCacheStore`** only requires that the keys are strings while the values can be any data type, so that they can be stored in a binary column.

Report a bug

10.8. CACHE LOADERS

10.8.1. About Cache Loaders

The cache loader provides JBoss Data Grid's connection to a persistent data store. The cache loader retrieves data from a data store when the required data is not present in the cache. If a cache loader is extended, it can be used as a cache store and can copy the modified data to the data store.

Cache loaders are associated with individual caches. Different caches attached to the same cache manager can have different cache store configurations.

Report a bug

10.8.2. Cache Loaders and Cache Stores

JBoss Cache originally shipped with a **`CacheLoader`** interface and a number of implementations. JBoss Data Grid has divided these into two distinct interfaces, a **`CacheLoader`** and a **`CacheStore`**. The **`CacheLoader`** loads a previously existing state from another location, while the **`CacheStore`** (which extends **`CacheLoader`**) exposes methods to store states as well as loading them. This division allows easier definition of read-only sources.

JBoss Data Grid ships with several high performance implementations of these interfaces.

Report a bug

10.8.3. Shared Cache Loaders

10.8.3.1. About Shared Cache Loaders

A shared cache loader is a cache loader that is shared by multiple cache instances.

A cache loader is useful when all instances in a cluster communicate with the same remote, shared database using the same JDBC settings. In such an instance, configuring a shared cache loader prevents the unnecessary repeated write operations that occur when various cache instances attempt to write the same data to the cache loader.

Report a bug

10.8.3.2. Enable Shared Cache Loaders

Library Mode

In JBoss Data Grid's Library mode, toggle cache loader sharing using the ***shared*** parameter within the `loader` element. This parameter is set to **FALSE** as a default. Enable cache loader sharing by setting the ***shared*** parameter to **TRUE**.

Remote Client-Server Mode

In JBoss Data Grid's Remote Client-Server mode, toggle cache loader sharing using the ***shared*** parameter within the `store` element. This parameter is set to **FALSE** as a default. Enable cache loader sharing by setting the ***shared*** parameter to **TRUE**. For example:

```
<jdbc-store shared="true">
...
</jdbc-store>
```

Report a bug

10.8.3.3. Invalidation Mode and Shared Cache Loaders

When used in conjunction with a shared cache loader, JBoss Data Grid's invalidation mode causes remote caches to refer to the shared cache loader to retrieve modified data.

The benefits of using invalidation mode in conjunction with shared cache loaders include the following:

- Compared to replication messages, which contain the updated data, invalidation messages are much smaller and result in reduced network traffic.
- The remaining cluster caches look up modified data from the shared cache loader lazily and only when required to do so, resulting in further reduced network traffic.

Report a bug

10.8.3.4. The Cache Loader and Cache Passivation

In JBoss Data Grid, a cache loader can be used to enforce the passivation of entries and to activate eviction in a cache. Whether passivation mode or activation mode are used, the configured cache loader both reads from and writes to the data store.

When passivation is disabled in JBoss Data Grid, after the modification, addition or removal of an element is carried out the cache loader steps in to persist the changes in the store.

For an example of how cache loaders behave with passivation disabled and enabled, refer to Section 16.5.3, “Eviction Example when Passivation is Disabled” and Section 16.5.4, “Eviction Example when Passivation is Enabled” respectively.

Report a bug

10.8.3.5. Application Cacheloader Registration

It is not necessary to register an application cache loader for an isolated deployment. This is not a requirement in JBoss Data Grid because lazy deserialization is used to work around this problem.

Report a bug

10.8.4. Connection Factories

10.8.4.1. About Connection Factories

In JBoss Data Grid, all JDBC cache loaders rely on a **`ConnectionFactory`** implementation to obtain a database connection. This process is also known as connection management or pooling.

A connection factory can be specified using the ***`ConnectionFactoryClass`*** configuration attribute. JBoss Data Grid includes the following **`ConnectionFactory`** implementations:

- ManagedConnectionFactory
- SimpleConnectionFactory.

Report a bug

10.8.4.2. About ManagedConnectionFactory

`ManagedConnectionFactory` is a connection factory that is ideal for use within managed environments such as application servers. This connection factory can explore a configured location in the JNDI tree and delegate connection management to the **`DataSource`**.
`ManagedConnectionFactory` is used within a managed environment that contains a **`DataSource`**. This **`Datasource`** is delegated the connection pooling.

Report a bug

10.8.4.3. About SimpleConnectionFactory

`SimpleConnectionFactory` is a connection factory that creates database connections on a per invocation basis. This connection factory is not designed for use in a production environment.

Report a bug

CHAPTER 11. CACHE MANAGERS

11.1. ABOUT CACHE MANAGERS

A Cache Manager is the primary mechanism used to retrieve a Cache instance in JBoss Data Grid, and can be used as a starting point for using the Cache.

Cache Managers are resource intensive and therefore a general implementation requires a single Cache Manager for each Java Virtual Machine (JVM) unless a specific configuration is used which requires multiple Cache Managers.

Report a bug

11.2. MULTIPLE CACHE MANAGERS

11.2.1. Create Multiple Caches with a Single Cache Manager

JBoss Data Grid allows using the same cache manager to create multiple caches, each with a different cache mode (synchronous and asynchronous cache modes).

Report a bug

11.2.2. Using Multiple Cache Managers

JBoss Data Grid allows multiple cache managers to be used. In most cases, such as with RPC and networking components, Cache instances share internal components and a single cache manager is sufficient.

However, if multiple caches are required to have different network characteristics, for example if one cache uses the TCP protocol and the other uses the UDP protocol, multiple cache managers must be used.

Report a bug

CHAPTER 12. EVICTION

12.1. ABOUT EVICTION

Eviction is the process of removing entries from memory to prevent running out of memory. Entries that are evicted from memory remain in cache stores and the rest of the cluster to prevent permanent data loss.

Report a bug

12.2. EVICTION OPERATIONS

Eviction occurs individually on a per node basis, rather than occurring as a cluster-wide operation. Each node uses an eviction thread to analyze the contents of its in-memory container to determine which entries require eviction. The free memory in the Java Virtual Machine (JVM) is not a consideration during the eviction analysis, even as a threshold to initialize entry eviction.

Report a bug

12.3. EVICTION USAGE

JBoss Data Grid executes eviction tasks by utilizing user threads which are already interacting with the data container. JBoss Data Grid uses a separate thread to prune expired cache entries from the cache.

See Also:

- Section 13.3, “Eviction and Expiration Comparison”

Report a bug

12.4. EVICTION STRATEGIES

12.4.1. About Eviction Strategies

JBoss Data Grid provides the following eviction strategies:

- **`EvictionStrategy.NONE`**: No eviction strategy set.
- **`EvictionStrategy.FIFO`**: First In, First Out eviction strategy.
- **`EvictionStrategy.LRU`**: Least Recently Used eviction strategy. This is the default eviction algorithm due to its compatibility with requirements for a wide variety of deployments.
- **`EvictionStrategy.UNORDERED`**: Unordered eviction strategy.
- **`EvictionStrategy.LIRS`**: Low Inter-reference Recency Set eviction strategy.

Report a bug

12.4.2. LRU Eviction Algorithm Limitations

Despite being simple to understand, the Least Recently Used (LRU) eviction algorithm does not

perform optimally in specific weak access locality use cases. In such cases, problems such as the following can appear:

- Single use access entries are not replaced in time.
- Entries that are accessed first are unnecessarily replaced.

Report a bug

12.5. USING EVICTION

12.5.1. Initialize Eviction

To initialize eviction, set the eviction element's ***max-entries*** attributes value to a number greater than zero. Adjust the value set for ***max-entries*** to discover the optimal value for your configuration. It is important to remember that if too large a value is set for ***max-entries***, JBoss Data Grid runs out of memory.

Report a bug

12.5.2. Default Eviction Configuration

In JBoss Data Grid, eviction is disabled by default. If an empty ***eviction*** element is used to enable eviction, the following default values are used:

- Strategy: If no eviction strategy is specified, ***EvictionStrategy.NONE*** is assumed as a default.
- max-entries: If no value is specified, the ***max-entries*** value is set to `-1`, which allows unlimited entries.
  - Set the ***max-entries*** value to `0` to disallow any entries. As a result, the eviction thread strives to keep the cache empty.

Report a bug

12.5.3. Eviction Configuration Example

Configure eviction in JBoss Data Grid using the configuration bean or the XML file. Eviction configuration is done on a per-cache basis.

XML Configuration (Library Mode)

An example of a valid eviction XML configuration for JBoss Data Grid's Library Mode is:

```
<eviction strategy="LRU" maxEntries="2000"/>
```

Programmatic Configuration (Library Mode)

Eviction configuration can be defined programmatically in JBoss Data Grid's Library Mode using:

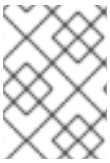
```
Configuration c = new
ConfigurationBuilder().eviction().strategy(EvictionStrategy.LRU)
              .maxEntries(2000)
```

```
                .build();
```

XML Configuration (Remote Client-Server Mode):

An example of eviction configuration using XML in JBoss Data Grid's Remote Client-Server Mode is:

```
<eviction strategy="FIFO" max-entries="20"/>
```



NOTE

Note that JBoss Data Grid's Library mode uses the ***maxEntries*** parameter while Remote Client-Server mode uses the ***max-entries*** parameter to configure eviction.

Report a bug

12.5.4. Eviction Configuration Troubleshooting

In JBoss Data Grid, the size of a cache can be larger than the value specified for the ***max-entries*** parameter of the **configuration** element. This is because although the ***max-entries*** value can be configured to a value that is not a power of two, the underlying algorithm will alter the value to **V**, where **V** is the closest power of two value that is larger than the ***max-entries*** value. Eviction algorithms are in place to ensure that the size of the cache container will never exceed the value **V**.

Report a bug

12.6. EVICTION AND PASSIVATION

12.6.1. About Eviction and Passivation

To ensure that a single copy of an entry remains, either in memory or in a cache store, use passivation in conjunction with eviction.

The primary reason to use passivation instead of a normal cache store is that updating entries require less resources when passivation is in use. This is because passivation does not require an update to the cache store.

See Also:

- Section 16.5, “Eviction and Passivation”

Report a bug

CHAPTER 13. EXPIRATION

13.1. ABOUT EXPIRATION

JBoss Data Grid uses expiration to attach one or both of the following values to an entry:

- A lifespan value.
- A maximum idle time value.

Both the life span and maximum idle time values are assigned the default value `-1` when created.

Expired entries, unlike evicted entries, are removed globally, which removes them from memory, cache stores and the cluster.

Report a bug

13.2. EXPIRATION OPERATIONS

Expiration in JBoss Data Grid allows you to set a life span or maximum idle time value for each key/value pair stored in the cache.

The life span or maximum idle time can be set to apply cache-wide or defined for each key/value pair using the cache API. The life span (***`lifespan`***) or maximum idle time (***`maxIdle`*** in Library Mode and ***`max-idle`*** in Remote Client-Server Mode) defined for an individual key/value pair overrides the cache-wide default for the entry in question.

Report a bug

13.3. EVICTION AND EXPIRATION COMPARISON

Expiration is a top-level construct in JBoss Data Grid, and is represented in the global configuration, as well as the cache API.

Eviction is limited to the cache instance it is used in, whilst expiration is cluster-wide. Expiration life spans (***`lifespan`***) and idle time (***`maxIdle`*** in Library Mode and ***`max-idle`*** in Remote Client-Server Mode) values are replicated alongside each cache entry.

Report a bug

13.4. CACHE ENTRY EXPIRATION NOTIFICATIONS

JBoss Data Grid does not guarantee that an eviction occurs immediately upon timeout. Instead, a number of mechanisms are used in collaboration to ensure efficient eviction. An expired entry is removed from the cache when either:

- A user thread requests an entry and discovers that the entry has expired.
- An entry is passivated/overflowed to disk and is discovered to have expired.
- The eviction maintenance thread discovers that an entry it has found is expired.

Report a bug

13.5. EXPIRATION CONFIGURATION

In JBoss Data Grid, expiration is configured in a manner similar to eviction.

Library Mode Configuration:

The following sample configuration depicts how to configure expiration in JBoss Data Grid's Library Mode:

```
<expiration lifespan="2000" maxIdle="1000" />
```

Remote Client-Server Mode Configuration:

The following sample configuration depicts how to configure expiration in JBoss Data Grids' Remote Client-Server Mode:

```
<expiration lifespan="2000" max-idle="1000" />
```

Report a bug

13.6. MORTAL AND IMMORTAL DATA

13.6.1. About Data Mortality

In addition to storing entities, JBoss Data Grid allows you to attach mortality information to data. For example, using the standard **`put(key, value)`** creates an entry that will never expire, called an immortal entry. Alternatively, an entry created using **`put(key, value, lifespan, timeunit)`** is a mortal entry that has a specified fixed life span, after which it expires.

In addition to the ***`lifespan`*** parameter, JBoss Data Grid also provides a ***`maxIdle`*** parameter used to determine expiration. The ***`maxIdle`*** and ***`lifespan`*** parameters can be used in various combinations to set the life span of an entry.

Report a bug

13.6.2. Default Data Mortality

As a default, newly created entries do not have a life span or maximum idle time value set. Without these two values, a data entry will never expire and is therefore known as immortal data.

Report a bug

13.6.3. Configure Data Mortality

Entry mortality (or its expiration values) can be set by setting the life span and maximum idle time values for the entry. After being set, these values must be persisted in the cache stores to ensure that they survive eviction and passivation.

Report a bug

13.7. TROUBLESHOOTING

13.7.1. Expiration Troubleshooting

If expiration does not appear to be working, it may be due to an entry being marked for expiration but not being removed.

Multiple-cache operations such as `put()` are passed a life span value as a parameter. This value defines the interval after which the entry must expire. In cases where eviction is not configured and the life span interval expires, it can appear as if JBoss Data Grid has not removed the entry. For example, when viewing JMX statistics, such as the `number of entries`, you may see an out of date count, or the persistent store associated with JBoss Data Grid may still contain this entry. Behind the scenes, JBoss Data Grid has marked it as an expired entry, but has not removed it. Removal of such entries happens in one of two ways:

- Any attempt to use `get()` or `containsKey()` for the expired entry, causes JBoss Data Grid to detect the entry as an expired one and remove it.
- Enabling the eviction feature causes the eviction thread to periodically detect and purge expired entries.

Report a bug

CHAPTER 14. THE L1 CACHE

14.1. ABOUT THE L1 CACHE

The Level 1 (or L1) cache stores remote cache entries after they are initially accessed, preventing unnecessary remote fetch operations for each subsequent use of the same entries. The L1 cache is created when JBoss Data Grid's cache mode is set to distribution, and is `disabled` by default.

The temporary location used by the L1 cache can be changed from the default location.

Report a bug

14.2. L1 CACHE ENTRIES

14.2.1. L1 Cache Entries

The L1 cache contains entries retrieved from a remote cache. When the location of an entry changes in the cluster, the corresponding L1 cache entry is invalidated to prevent outdated cache entries.

When L1 is enabled, the cache consults the L1 cache before fetching an entry from a remote location to prevent an unnecessary fetch operation.

Report a bug

14.2.2. L1 Cache Entry Life Spans

Each L1 cache entry has a default life span of `600,000` milliseconds (10 minutes). The life span value can be altered to meet user requirements.

Report a bug

14.3. L1 CACHE OPERATIONS

14.3.1. The L1 Cache and Invalidation

An invalidation message is generated each time a key is updated. This message is multicast to each node that contains data that corresponds to current L1 cache entries. The invalidation message ensures that each of these nodes marks the relevant entry as invalidated.

Report a bug

14.3.2. Using the L1 Cache with GET Operations

Multiple **GET** operations performed on the same key generate repeated remote calls. To reduce the number of unnecessary **GET** operations on the same key, enable L1 caching.

The L1 cache provides a local (and temporary) cache that houses values retrieved from remote caches. The location used for the L1 cache can be configured.

Report a bug

CHAPTER 15. LISTENERS AND NOTIFICATIONS

15.1. ABOUT THE LISTENER API

JBoss Data Grid provides a listener API that provides notifications for events as they occur. Clients can choose to register with the listener API for relevant notifications. This annotation-driven API operates on cache-level events and cache manager-level events.

Report a bug

15.2. LISTENER EXAMPLE

The following example defines a listener in JBoss Data Grid that prints some information each time a new entry is added to the cache:

```
@Listener
public class PrintWhenAdded {
  @CacheEntryCreated
  public void print(CacheEntryCreatedEvent event) {
    System.out.println("New entry " + event.getKey() + " created in the
cache");
  }
}
```

Report a bug

15.3. CACHE ENTRY MODIFIED LISTENER CONFIGURATION

In a cache entry modified listener, the modified value cannot be retrieved via **`Cache.get()`** when ***`isPre`*** equals **`false`**

Instead, use **`CacheEntryModifiedEvent.getValue()`** to retrieve the new value of the modified entry.

Report a bug

15.4. NOTIFICATIONS

15.4.1. About Listener Notifications

Each cache event triggers a notification that is dispatched to listeners. A listener is a simple POJO annotated with **`@Listener`**. A Listenable is an interface that denotes that the implementation can have listeners attached to it. Each listener is registered using methods defined in the Listenable.

A listener can be attached to both the cache and Cache Manager to allow them to receive cache-level or cache manager-level notifications.

Report a bug

15.4.2. About Cache-level Notifications

In JBoss Data Grid, cache-level events occur on a per-cache basis, and are global and cluster-wide. Examples of cache-level events include the addition, removal and modification of entries, which trigger notifications to listeners registered on the relevant cache.

See Also:

- Section 13.4, "Cache Entry Expiration Notifications"

Report a bug

15.4.3. Cache Manager-level Notifications

Examples of events that occur in JBoss Data Grid at the cache manager-level are:

- Nodes joining or leaving a cluster;
- The starting and stopping of caches

Cache manager-level events are located globally and used cluster-wide, but are restricted to events within caches created by a single cache manager.

Report a bug

15.4.4. About Synchronous and Asynchronous Notifications

By default, notifications in JBoss Data Grid are dispatched in the same thread that generated the event. Therefore it is important that a listener is written in a way that does not block or prevent the thread from progressing.

Alternatively, the listener can be annotated as asynchronous, which dispatches notifications in a separate thread and prevents blocking the operations of the original thread.

Annotate listeners using the following:

```
@Listener (sync = false)public class MyAsyncListener { .... }
```

Use the ***<asyncListenerExecutor/>*** element in the configuration file to tune the thread pool that is used to dispatch asynchronous notifications.

Report a bug

15.5. NOTIFYING FUTURES

15.5.1. About NotifyingFutures

Methods in JBoss Data Grid do not return Java Development Kit (JDK) **`Futures`**, but a sub-interface known as a **`NotifyingFuture`**. Unlike a JDK **`Future`**, a listener can be attached to a **`NotifyingFuture`** to notify the user about a completed future.

NOTE

In JBoss Data Grid, **`NotifyingFutures`** are only available in Library mode.

Report a bug

15.5.2. NotifyingFutures Example

The following is an example depicting how to use **`NotifyingFutures`** in JBoss Data Grid:

```
FutureListener futureListener = new FutureListener() {

public void futureDone(Future future) {
      try {
         future.get();
      } catch (Exception e) {
         // Future did not complete successfully
         System.out.println("Help!");
      }
   }
};

cache.putAsync("key", "value").attachListener(futureListener);
```

Report a bug

CHAPTER 16. ACTIVATION AND PASSIVATION MODES

16.1. ABOUT ACTIVATION MODE

Activation is the process of loading an entry into memory and removing it from the cache store. Activation occurs when a thread attempts to access an entry that is in the store but not the memory (namely a passivated entry).

Report a bug

16.2. ABOUT PASSIVATION MODE

Passivation mode allows entries to be stored in the cache store after they are evicted from memory. Passivation prevents unnecessary and potentially expensive writes to the cache store. It is used for entries that are frequently used or referenced and therefore not evicted from memory.

While passivation is enabled, the cache store is used as an overflow tank, similar to virtual memory implementation in operating systems that swap memory pages to disk.

Report a bug

16.3. PASSIVATION MODE BENEFITS

The primary benefit of passivation mode is that it prevents unnecessary and potentially expensive writes to the cache store. This is particularly useful if an entry is frequently used or referenced and therefore is not evicted from memory.

Report a bug

16.4. ABOUT THE PASSIVATION FLAG

The passivation flag is used to toggle passivation mode, a mode that stores entries in the cache store only after they are evicted from memory.

Report a bug

16.5. EVICTION AND PASSIVATION

16.5.1. About Eviction and Passivation

To ensure that a single copy of an entry remains, either in memory or in a cache store, use passivation in conjunction with eviction.

The primary reason to use passivation instead of a normal cache store is that updating entries require less resources when passivation is in use. This is because passivation does not require an update to the cache store.

Report a bug

16.5.2. Eviction and Passivation Usage

If the eviction policy caused the eviction of an entry from the cache while passivation is enabled, the following occur as a result:

- A notification regarding the passivated entry is emitted to the cache listeners.
- The evicted entry is stored.

When an attempt to retrieve an evicted entry is made, the entry is lazily loaded into memory from the cache loader. After the entry and its children are loaded, they are removed from the cache loader and a notification regarding the entry's activation is sent to the cache listeners.

Report a bug

16.5.3. Eviction Example when Passivation is Disabled

The following example indicates the state of the memory and the persistent store during eviction operations with passivation disabled.

Table 16.1. Eviction when Passivation is Disabled

Step	Key in Memory	Key on Disk
Insert **keyOne**	Memory: **keyOne**	Disk: **keyOne**
Insert **keyTwo**	Memory: **keyOne**, **keyTwo**	Disk: **keyOne**, **keyTwo**
Eviction thread runs, evicts **keyOne**	Memory: **keyTwo**	Disk: **keyOne**, **keyTwo**
Read **keyOne**	Memory: **keyOne**, **keyTwo**	Disk: **keyOne**, **keyTwo**
Eviction thread runs, evicts **keyTwo**	Memory: **keyOne**	Disk: **keyOne**, **keyTwo**
Remove **keyTwo**	Memory: **keyOne**	Disk: **keyOne**

Report a bug

16.5.4. Eviction Example when Passivation is Enabled

The following example indicates the state of the memory and the persistent store during eviction operations with passivation enabled.

Table 16.2. Eviction when Passivation is Enabled

Step	Key in Memory	Key on Disk
Insert **keyOne**	Memory: **keyOne**	Disk:
Insert **keyTwo**	Memory: **keyOne**, **keyTwo**	Disk:

Step	Key in Memory	Key on Disk
Eviction thread runs, evicts **keyOne**	Memory: **keyTwo**	Disk: **keyOne**
Read **keyOne**	Memory: **keyOne**, **keyTwo**	Disk:
Eviction thread runs, evicts **keyTwo**	Memory: **keyOne**	Disk: **keyTwo**
Remove **keyTwo**	Memory: **keyOne**	Disk:

Report a bug

CHAPTER 17. CUSTOM INTERCEPTORS

17.1. ABOUT CUSTOM INTERCEPTORS

Custom interceptors can be added to JBoss Data Grid declaratively or programmatically. Custom interceptors extend JBoss Data Grid by allowing it to influence or respond to cache modifications. Examples of such cache modifications are the addition, removal or updating of elements or transactions.

Report a bug

17.2. CUSTOM INTERCEPTOR DESIGN

To design a custom interceptor in JBoss Data Grid, adhere to the following guidelines:

- A custom interceptor must extend the **`CommandInterceptor`**.
- A custom interceptor must declare a public, empty constructor to allow for instantiation.
- A custom interceptor must have JavaBean style setters defined for any property that is defined through the **`property`** element.

Report a bug

17.3. ADDING CUSTOM INTERCEPTORS

17.3.1. Adding Custom Interceptors Declaratively

Each named cache in JBoss Data Grid has it's own interceptor stack. As a result, custom interceptors can be added on a per named cache basis.

A custom interceptor can be added using XML. For example:

```
<namedCache name="cacheWithCustomInterceptors">
   <!--
   Define custom interceptors.  All custom interceptors need to extend
org.jboss.cache.interceptors.base.CommandInterceptor
   -->
   <customInterceptors>
      <interceptor position="FIRST"
class="com.mycompany.CustomInterceptor1">
         <properties>
            <property name="attributeOne" value="value1" />
            <property name="attributeTwo" value="value2" />
         </properties>
      </interceptor>
      <interceptor position="LAST"
class="com.mycompany.CustomInterceptor2"/>
      <interceptor index="3" class="com.mycompany.CustomInterceptor1"/>
      <interceptor before="org.infinispanpan.interceptors.CallInterceptor"
class="com.mycompany.CustomInterceptor2"/>
      <interceptor after="org.infinispanpan.interceptors.CallInterceptor"
class="com.mycompany.CustomInterceptor1"/>
   </customInterceptors>
```

```
</namedCache>
```

NOTE

This configuration is only valid for JBoss Data Grid's Library Mode.

Report a bug

17.3.2. Adding Custom Interceptors Programmatically

To add a custom interceptor programmatically in JBoss Data Grid, first obtain a reference to the **AdvancedCache**.

For example:

```
CacheManager cm = getCacheManager();
Cache aCache = cm.getCache("aName");
AdvancedCache advCache = aCache.getAdvancedCache();
```

Then use an ***addInterceptor()*** method to add the interceptor.

For example:

```
advCache.addInterceptor(new MyInterceptor(), 0);
```

Report a bug

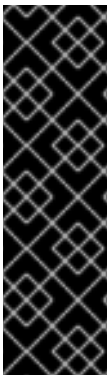
CHAPTER 18. TRANSACTIONS

18.1. ABOUT JAVA TRANSACTION API TRANSACTIONS

JBoss Data Grid supports configuring, use of and participation in JTA compliant transactions. However, disabling transaction support is the equivalent of using the automatic commit feature in JDBC calls, where modifications are potentially replicated after every change, if replication is enabled.

JBoss Data Grid does the following for each cache operation:

1. First, it retrieves the transactions currently associated with the thread.
2. If not already done, it registers **`XAResource`** with the transaction manager to receive notifications when a transaction is committed or rolled back.



IMPORTANT

With JBoss Data Grid 6.0.x, it is recommended to disable transactions in Remote Client-Server Mode. However, if an error displays warning of an **`ExceptionTimeout`** where JBoss Data Grid is **`Unable to acquire lock after {time} on key {key} for requester {thread}`**, enable transactions. This occurs because non-transactional caches acquire locks on each node they write on. Using transactions prevents deadlocks because caches acquire locks on a single node.

This problem is resolved in JBoss Data Grid 6.1.

Report a bug

18.2. TRANSACTIONS SPANNING MULTIPLE CACHE INSTANCES

Each cache operates as a separate, standalone Java Transaction API (JTA) resource. However, components can be internally shared by JBoss Data Grid for optimization, but this sharing does not affect how caches interact with a Java Transaction API (JTA) Manager.

Report a bug

18.3. TRANSACTION/BATCHING AND INVALIDATION MESSAGES

When making modifications in invalidation mode without the use of batching or transactions, invalidation messages are dispatched after each modification occurs. If transactions or batching are in use, invalidation messages are sent following a successful commit.

Using invalidation with transactions or batching provides greater efficiency as transmitting messages in bulk results in less network traffic.

Report a bug

18.4. THE TRANSACTION MANAGER

18.4.1. About JTA Transaction Manager Lookup Classes

In order to execute a cache operation, the cache requires a reference to the environment's Transaction

Manager. Configure the cache with the class name that belongs to an implementation of the **`TransactionManagerLookup`** interface. When initialized, the cache creates an instance of the specified class and invokes its **`getTransactionManager()`** method to locate and return a reference to the Transaction Manager.

JBoss Data Grid includes the following transaction manager lookup classes:

- The **`DummyTransactionManagerLookup`** provides a transaction manager for testing purposes. This testing transaction manager is not for use in a production environment and is severely limited in terms of functionality, specifically for concurrent transactions and recovery.
- The **`JBossStandaloneJTAManagerLookup`** is the default transaction manager when JBoss Data Grid runs in a standalone environment. It is a fully functional JBoss Transactions based transaction manager that overcomes the functionality limits of the **`DummyTransactionManagerLookup`**.
- The **`GenericTransactionManagaerLookup`** is a lookup class used to locate transaction managers in most Java EE application servers. If no transaction manager is located, it defaults to **`DummyTransactionManagerLookup`**.
- The **`JBossTransactionManagerLookup`** is a lookup class that locates a transaction manager within a JBoss Application Server instance.

NOTE

In Remote Client-Server mode, all JBoss Data Grid operations are non transactional. As a result, the listed JTA Transaction Manager Lookup classes can only be used in JBoss Data Grid's Library Mode.

Report a bug

18.4.2. Use the Transaction Manager

18.4.2.1. Obtain the Transaction Manager From the Cache

Use the following configuration after initialization to obtain the TransactionManager from the cache:

Procedure 18.1. Obtain the Transaction Manager from the Cache

1. Define a **`transactionManagerLookupClass`** by adding the following property to your **`BasicCacheContainer`**'s configuration location properties:

```
Configuration config = new ConfigurationBuilder()
...
  .transaction().transactionManagerLookup(new
GenericTransactionManagerLookup())
```

2. Call **`TransactionManagerLookup.getTransactionManager`** as follows:

```
TransactionManager tm =
cache.getAdvancedCache().getTransactionManager();
```

Report a bug

18.4.2.2. Transaction Configuration

JBoss Data Grid transactions are configured at the cache level. The following is an example of how to configure transactions:

```
<cache>
..
 <transaction mode="NONE"
              stop-timeout="30000"
              locking="OPTIMISTIC" />
...
</cache>
```

- The ***mode*** attribute sets the cache transaction mode. Valid values for this attribute are **NONE** (default), **NON_XA**, **NON_DURABLE_XA**, **FULL_XA**.
- The ***stop-timeout*** attribute indicates the number of milliseconds JBoss Data Grid waits for ongoing remote and local transactions to conclude when the cache is stopped.
- The ***locking*** attribute specifies the locking mode used for the cache. Valid values for this attribute are **OPTIMISTIC** (default) and **PESSIMISTIC**.

Report a bug

18.4.2.3. Transaction Manager and XAResources

Despite being specific to the Transaction Manager, the transaction recovery process must provide a reference to a **XAResource** implementation to run **XAResource.recover** on it.

Report a bug

18.4.2.4. Obtain a XAResource Reference

To obtain a reference to a JBoss Data Grid **XAResource**, use the following API:

```
XAResource xar = cache.getAdvancedCache().getXAResource();
```

Report a bug

18.4.2.5. Default Distributed Transaction Behavior

JBoss Data Grid's default behavior is to register itself as a first class participant in distributed transactions through **XAResource**. In situations where JBoss Data Grid does not need to be a participant in a transaction, it can be notified about the lifecycle status (for example, prepare, complete, etc.) of the transaction via a synchronization.

Report a bug

18.5. TRANSACTION SYNCHRONIZATION

18.5.1. About Transaction (JTA) Synchronizations

JTA synchronizations inform JBoss Data Grid about transaction life cycle events. Registering a synchronization also allows JBoss Data Grid to respond to transactional events without being registered as a **XAResource**. As a result, the Transaction Manager optimizes transaction operations so that they require the one phase commit (1PC) algorithm instead of the two phase commit (2PC) algorithm.

In JBoss Data Grid, JTA synchronizations are only available in Library mode.

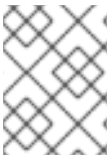
Report a bug

18.5.2. Enable Synchronization

In JBoss Data Grid, synchronization is automatically used for transactions where the mode is set to a non-XA option.

To enable synchronization, set the transaction element's mode parameter to either:

- **NONE** (synchronous), or;
- **NO_XA** (synchronous).



NOTE

JBoss Data Grid only allows transaction enlistment through synchronization in Library mode.

Report a bug

18.6. STATE RECONCILIATION

18.6.1. About State Reconciliation

To reconcile states in JBoss Data Grid, the Transaction Manager passes information about transactions that require manual intervention to the system administrator in a proprietary manner.

The system administrator must know the relevant transaction's XID as a prerequisite for the transaction recovery task. The transaction's XID is stored as a byte array.

Report a bug

18.6.2. About Transaction Recovery

The Transaction Manager coordinates the recovery process and works with JBoss Data Grid to determine which transactions require manual intervention to complete operations. This process is known as transaction recovery.

Report a bug

18.6.3. Enable Transaction Recovery

Transaction recovery is disabled by default in JBoss Data Grid. When disabled, the Transaction Manager cannot determine which transactions require manual intervention.

Using XML

Enable transaction recovery using XML configuration as follows:

```
<transaction useEagerLocking="true" eagerLockSingleNode="true">
    <recovery enabled="true" recoveryInfoCacheName="noRecovery"/>
</transaction>
```

The ***recoveryInfoCacheName*** attribute is optional.

Programmatic Configuration

Alternatively, enable transaction recovery through the fluent configuration API as follows:

Procedure 18.2. Configure Transaction Recovery Programmatically

1. To enable JBoss Data Grid's Transaction Recovery, call **.recovery**:

```
configuration.fluent().transaction().recovery();
```

2. To check Transactions Recovery's status, use the following programmatic configuration:

```
boolean isRecoveryEnabled =
configuration.isTransactionRecoveryEnabled();
```

3. To disable JBoss Data Grid's Transaction recovery, use the following programmatic configuration:

```
configuration.fluent().transaction().recovery().disable();
```

Transaction recovery can be enabled or disabled on a per cache basis. For example, it is possible for a transaction to span one cache with transaction recovery enabled and then another cache with transaction recovery disabled.

Report a bug

18.6.4. Transaction Recovery Process

The following process outlines the transaction recovery process in JBoss Data Grid.

Procedure 18.3. The Transaction Recovery Process

1. The Transaction Manager creates a list of transactions that require intervention.

2. The system administrator, connected to JBoss Data Grid using JMX, is presented with the list of transactions (including transaction IDs) using email or logs. The status of each transaction is either **COMMITTED** or **PREPARED**. If some transactions are in both **COMMITTED** and **PREPARED** states, it indicates that the transaction was committed on some nodes while in the preparation state on others.

3. The System Administrator visually maps the XID received from the Transaction Manager to a JBoss Data Grid internal ID. This step is necessary because the XID (a byte array) cannot be conveniently passed to the JMX tool and then reassembled by JBoss Data Grid without this mapping.

4. The system administrator forces the commit or rollback process for a transaction based on the mapped internal ID.

Report a bug

18.6.5. Transaction Recovery Example

The following example describes how transactions are used in a situation where money is transferred from an account stored in a database to an account stored in JBoss Data Grid.

> **Example 18.1. Money Transfer from an Account Stored in a Database to an Account in JBoss Data Grid**
>
> 1. The `TransactionManager.commit()` method is invoked to to run the two phase commit protocol between the source (the database) and the destination (JBoss Data Grid) resources.
>
> 2. The `TransactionManager` tells the database and JBoss Data Grid to initiate the prepare phase (the first phase of a Two Phase Commit).
>
> During the commit phase, the database applies the changes but JBoss Data Grid fails before receiving the Transaction Manager's commit request. As a result, the system is in an inconsistent state due to an incomplete transaction. Specifically, the amount to be transferred has been subtracted from the database but is not yet visible in JBoss Data Grid because the prepared changes could not be applied.
>
> Transaction recovery is used here to reconcile the inconsistency between the database and JBoss Data Grid entries.

Report a bug

18.6.6. Transaction Memory and JMX Support

It is important to note that to use JMX to manage transaction recoveries, JMX support must be explicitly enabled.

Report a bug

18.6.7. Forced Commit and Rollback Operations

JBoss Data Grid uses JMX for operations that explicitly force transactions to commit or rollback. These methods receive byte arrays that describe the XID instead of the number associated with the relevant transactions.

The System Administrator can use such JMX operations to facilitate automatic job completion for transactions that require manual intervention. This process uses the Transaction Manager's transaction recovery process and has access to the Transaction Manager's XID objects.

Report a bug

18.6.8. Transactions and Exceptions

If a cache method returns a **`CacheException`** (or a subclass of the **`CacheException`**) within the scope of a JTA transaction, the transaction is automatically marked to be rolled back.

Report a bug

18.7. DEADLOCK DETECTION

18.7.1. About Deadlock Detection

A deadlock occurs when multiple processes or tasks wait for the other to release a mutually required resource. Deadlocks can significantly reduce the throughput of a system, particularly when multiple transactions operate against one key set.

JBoss Data Grid provides deadlock detection to identify such deadlocks. Deadlock detection is set to **`disabled`** by default.

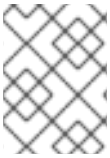
Report a bug

18.7.2. Enable Deadlock Detection

Deadlock detection in JBoss Data Grid is set to **`disabled`** by default but can be enabled and configured for each cache using the ***namedCache*** configuration element by adding the following:

```
<deadlockDetection enabled="true" spinDuration="1000"/>
```

Deadlock detection can only be applied to individual caches. Deadlocks that are applied on more than one cache cannot be detected by JBoss Data Grid.



NOTE

JBoss Data Grid only allows deadlock detection to be configured in Library mode. This configuration is not available in Remote Client-Server mode.

Report a bug

CHAPTER 19. MARSHALLING

19.1. ABOUT MARSHALLING

Marshalling is the process of converting Java objects into a format that is transferable over the wire. Unmarshalling is the reversal of this process where data read from a wire format is converted into Java objects.

Report a bug

19.2. MARSHALLING BENEFITS

JBoss Data Grid uses marshalling and unmarshalling for the following purposes:

- To transform data for relay to other JBoss Data Grid nodes within the cluster.
- To transform data to be stored in underlying cache stores.

Report a bug

19.3. ABOUT MARSHALLING FRAMEWORK

JBoss Data Grid uses the JBoss Marshalling Framework to marshall and unmarshall Java POJOs. Using the JBoss Marshalling Framework offers a significant performance benefit, and is therefore used instead of Java Serialization. Additionally, the JBoss Marshalling Framework can efficiently marshall Java POJOs, including Java classes.

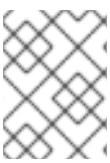
The Java Marshalling Framework uses high performance **`java.io.ObjectOutput`** and **`java.io.ObjectInput`** implementations compared to the standard **`java.io.ObjectOutputStream`** and **`java.io.ObjectInputStream`**.

Report a bug

19.4. SUPPORT FOR NON-SERIALIZABLE OBJECTS

A common user concern is whether JBoss Data Grid supports the storage of non-serializable objects. In JBoss Data Grid, marshalling is supported for non-serializable key-value objects; users can provide externalizer implementations for non-serializable objects.

If you are unable to retrofit **`Serializable`** or **`Externalizable`** support into your classes, you could (as an example) use XStream to convert the non-serializable objects into a String that can be stored in JBoss Data Grid.



NOTE

XStream slows down the process of storing key-value objects due to the required XML transformations.

Report a bug

19.5. TROUBLESHOOTING

19.5.1. Marshalling Troubleshooting

In JBoss Data Grid, the marshalling layer and JBoss Marshalling in particular, can produce errors when marshalling or unmarshalling a user object. The exception stack trace contains further information to help you debug the problem.

The following is an example of such a stack trace:

```
java.io.NotSerializableException: java.lang.Object
at
org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.
java:857)
at
org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.ja
va:407)
at
org.infinispan.marshall.exts.ReplicableCommandExternalizer.writeObject(Rep
licableCommandExternalizer.java:54)
at
org.infinispan.marshall.jboss.ConstantObjectTable$ExternalizerAdapter.writ
eObject(ConstantObjectTable.java:267)
at
org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.
java:143)
at
org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.ja
va:407)
at
org.infinispan.marshall.jboss.JBossMarshaller.objectToObjectStream(JBossMa
rshaller.java:167)
at
org.infinispan.marshall.VersionAwareMarshaller.objectToBuffer(VersionAware
Marshaller.java:92)
at
org.infinispan.marshall.VersionAwareMarshaller.objectToByteBuffer(VersionA
wareMarshaller.java:170)
at
org.infinispan.marshall.VersionAwareMarshallerTest.testNestedNonSerializab
le(VersionAwareMarshallerTest.java:415)
Caused by: an exception which occurred:
in object java.lang.Object@b40ec4
in object org.infinispan.commands.write.PutKeyValueCommand@df661da7
... Removed 22 stack frames
```

In object messages and stack traces are read in the same way: the highest in object is the innermost one and the outermost in object is the lowest.

The provided example indicates that a **`java.lang.Object`** instance within an **`org.infinispan.commands.write.PutKeyValueCommand`** instance cannot be serialized because **`java.lang.Object@b40ec4`** is not serializable.

However, if the **DEBUG** or **TRACE** logging levels are enabled, marshalling exceptions will contain **`toString()`** representations of objects in the stack trace. The following is an example that depicts such a scenario:

```
java.io.NotSerializableException: java.lang.Object
```

```
...
Caused by: an exception which occurred:
in object java.lang.Object@b40ec4
-> toString = java.lang.Object@b40ec4
in object org.infinispan.commands.write.PutKeyValueCommand@df661da7
->  toString = PutKeyValueCommand{key=k, value=java.lang.Object@b40ec4,
putIfAbsent=false, lifespanMillis=0, maxIdleTimeMillis=0}
```

Displaying this level of information for unmarshalling exceptions is expensive in terms of resources. However, where possible, JBoss Data Grid displays class type information. The following example depicts such levels of information on display:

```
java.io.IOException: Injected failue!
at
org.infinispan.marshall.VersionAwareMarshallerTest$1.readExternal(VersionA
wareMarshallerTest.java:426)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadNewObject(RiverUnmarsh
aller.java:1172)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshall
er.java:273)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshall
er.java:210)
at
org.jboss.marshalling.AbstractUnmarshaller.readObject(AbstractUnmarshaller
.java:85)
at
org.infinispan.marshall.jboss.JBossMarshaller.objectFromObjectStream(JBoss
Marshaller.java:210)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(Versio
nAwareMarshaller.java:104)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(Versio
nAwareMarshaller.java:177)
at
org.infinispan.marshall.VersionAwareMarshallerTest.testErrorUnmarshalling(
VersionAwareMarshallerTest.java:431)
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.VersionAwareMarshallerTest$1
```

In the provided example, an **`IOException`** was thrown when an instance of the inner class **`org.infinispan.marshall.VersionAwareMarshallerTest$1`** is unmarshalled.

In a manner similar to marshalling exceptions, when **DEBUG** or **TRACE** logging levels are enabled, the class type's classloader information is provided. An example of this classloader information is as follows:

```
java.io.IOException: Injected failue!
...
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.VersionAwareMarshallerTest$1
-> classloader hierarchy:
```

```
-> type classloader = sun.misc.Launcher$AppClassLoader@198dfaf
-
>...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/ecl
ipse-testng.jar
-
>...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/lib
/testng-jdk15.jar
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/test-
classes/
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/classes/
->...file:/home/galder/.m2/repository/org/testng/testng/5.9/testng-5.9-
jdk15.jar
->...file:/home/galder/.m2/repository/net/jcip/jcip-annotations/1.0/jcip-
annotations-1.0.jar
-
>...file:/home/galder/.m2/repository/org/easymock/easymockclassextension/2
.4/easymockclassextension-2.4.jar
->...file:/home/galder/.m2/repository/org/easymock/easymock/2.4/easymock-
2.4.jar
->...file:/home/galder/.m2/repository/cglib/cglib-nodep/2.1_3/cglib-nodep-
2.1_3.jar
->...file:/home/galder/.m2/repository/javax/xml/bind/jaxb-api/2.1/jaxb-
api-2.1.jar
->...file:/home/galder/.m2/repository/javax/xml/stream/stax-api/1.0-
2/stax-api-1.0-2.jar
-
>...file:/home/galder/.m2/repository/javax/activation/activation/1.1/activ
ation-1.1.jar
->...file:/home/galder/.m2/repository/jgroups/jgroups/2.8.0.CR1/jgroups-
2.8.0.CR1.jar
->...file:/home/galder/.m2/repository/org/jboss/javaee/jboss-transaction-
api/1.0.1.GA/jboss-transaction-api-1.0.1.GA.jar
-
>...file:/home/galder/.m2/repository/org/jboss/marshalling/river/1.2.0.CR4
-SNAPSHOT/river-1.2.0.CR4-SNAPSHOT.jar
->...file:/home/galder/.m2/repository/org/jboss/marshalling/marshalling-
api/1.2.0.CR4-SNAPSHOT/marshalling-api-1.2.0.CR4-SNAPSHOT.jar
->...file:/home/galder/.m2/repository/org/jboss/jboss-common-
core/2.2.14.GA/jboss-common-core-2.2.14.GA.jar
->...file:/home/galder/.m2/repository/org/jboss/logging/jboss-logging-
spi/2.0.5.GA/jboss-logging-spi-2.0.5.GA.jar
->...file:/home/galder/.m2/repository/log4j/log4j/1.2.14/log4j-1.2.14.jar
-
>...file:/home/galder/.m2/repository/com/thoughtworks/xstream/xstream/1.2/
xstream-1.2.jar
->...file:/home/galder/.m2/repository/xpp3/xpp3_min/1.1.3.4.O/xpp3_min-
1.1.3.4.O.jar
->...file:/home/galder/.m2/repository/com/sun/xml/bind/jaxb-
impl/2.1.3/jaxb-impl-2.1.3.jar
-> parent classloader = sun.misc.Launcher$ExtClassLoader@1858610
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/localedata.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunpkcs11.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunjce_provider.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/dnsns.jar
... Removed 22 stack frames
```

Report a bug

19.5.2. State Receiver EOFExceptions

During a state transfer, if an EOFException is logged that states that the state receiver has **`Read past end of file`**, this can be dealt with depending on whether the state provider encounters an error when generating the state. For example, if the state provider is currently providing a state to a node, when another node requests a state, the state generator log can contain:

```
2010-12-09 10:26:21,533 20267 ERROR
[org.infinispan.remoting.transport.jgroups.JGroupsTransport]
(STREAMING_STATE_TRANSFER-sender-1,Infinispan-Cluster,NodeJ-2368:) Caught
while responding to state transfer request
org.infinispan.statetransfer.StateTransferException:
java.util.concurrent.TimeoutException: Could not obtain exclusive
processing lock
     at
org.infinispan.statetransfer.StateTransferManagerImpl.generateState(StateT
ransferManagerImpl.java:175)
     at
org.infinispan.remoting.InboundInvocationHandlerImpl.generateState(Inbound
InvocationHandlerImpl.java:119)
     at
org.infinispan.remoting.transport.jgroups.JGroupsTransport.getState(JGroup
sTransport.java:586)
     at
org.jgroups.blocks.MessageDispatcher$ProtocolAdapter.handleUpEvent(Message
Dispatcher.java:691)
     at
org.jgroups.blocks.MessageDispatcher$ProtocolAdapter.up(MessageDispatcher.
java:772)
     at org.jgroups.JChannel.up(JChannel.java:1465)
     at org.jgroups.stack.ProtocolStack.up(ProtocolStack.java:954)
     at org.jgroups.protocols.pbcast.FLUSH.up(FLUSH.java:478)
     at
org.jgroups.protocols.pbcast.STREAMING_STATE_TRANSFER$StateProviderHandler
.process(STREAMING_STATE_TRANSFER.java:653)
     at
org.jgroups.protocols.pbcast.STREAMING_STATE_TRANSFER$StateProviderThreadS
pawner$1.run(STREAMING_STATE_TRANSFER.java:582)
     at
java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.
java:886)
     at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java
:908)
     at java.lang.Thread.run(Thread.java:680)
Caused by: java.util.concurrent.TimeoutException: Could not obtain
exclusive processing lock
     at
org.infinispan.remoting.transport.jgroups.JGroupsDistSync.acquireProcessin
gLock(JGroupsDistSync.java:71)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.generateTransactionL
og(StateTransferManagerImpl.java:202)
     at
```

```
org.infinispan.statetransfer.StateTransferManagerImpl.generateState(StateT
ransferManagerImpl.java:165)
     ... 12 more
```

The implication of this exception is that the state generator was unable to generate the transaction log hence the output it was writing in now closed. In such a situation, the state receiver will often log an ***EOFException***, displayed as follows, when failing to read the transaction log that was not written by the sender:

```
2010-12-09 10:26:21,535 20269 TRACE
[org.infinispan.marshall.VersionAwareMarshaller] (Incoming-2,Infinispan-
Cluster,NodeI-38030:) Log exception reported
java.io.EOFException: Read past end of file
     at
org.jboss.marshalling.AbstractUnmarshaller.eofOnRead(AbstractUnmarshaller.
java:184)
     at
org.jboss.marshalling.AbstractUnmarshaller.readUnsignedByteDirect(Abstract
Unmarshaller.java:319)
     at
org.jboss.marshalling.AbstractUnmarshaller.readUnsignedByte(AbstractUnmars
haller.java:280)
     at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshall
er.java:207)
     at
org.jboss.marshalling.AbstractUnmarshaller.readObject(AbstractUnmarshaller
.java:85)
     at
org.infinispan.marshall.jboss.GenericJBossMarshaller.objectFromObjectStrea
m(GenericJBossMarshaller.java:175)
     at
org.infinispan.marshall.VersionAwareMarshaller.objectFromObjectStream(Vers
ionAwareMarshaller.java:184)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.processCommitLog(Sta
teTransferManagerImpl.java:228)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.applyTransactionLog(
StateTransferManagerImpl.java:250)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.applyState(StateTran
sferManagerImpl.java:320)
     at
org.infinispan.remoting.InboundInvocationHandlerImpl.applyState(InboundInv
ocationHandlerImpl.java:102)
     at
org.infinispan.remoting.transport.jgroups.JGroupsTransport.setState(JGroup
sTransport.java:603)
        ...
```

When this error occurs, the state receiver attempts the operation every few seconds until it is successful. In most cases, after the first attempt, the state generator has already finished processing the second node and is fully receptive to the state, as expected.

Report a bug

CHAPTER 20. JGROUPS INTERFACES

20.1. ABOUT JGROUPS INTERFACES

JGroups is the underlying group communication library used to connect JBoss Data Grid instances. JGroups allows users to bind to an interface type rather than a specific (unknown) IP address.

Report a bug

20.2. CONFIGURE JGROUPS INTERFACES

Before JGroups Interface configuration, configure the ***bind_addr*** property in the JGroups configuration file to a key word rather than a dotted decimal or symbolic IP address as follows:

```
<socket-binding name="jgroups-udp" ...  interface="site-local"/>
```

Then, configure the JGroups Interface as follows:

```
<interfaces>
 <interface name="link-local"><link-local-address/></interface>
 <interface name="site-local"><site-local-address/></interface>
 <interface name="global"><any-address/></interface>
 <interface name="non-loopback"><not><loopback/></not></interface>
</interfaces>
```

The configuration values used in the example are as follows:

- ***link-local***: Uses a **169.*x*.*x*.*x*** or **254.*x*.*x*.*x*** address. This suits the traffic within one box.
- ***site-local***: Uses a private IP address, for example **192.168.*x*.*x***. This prevents extra bandwidth charged from GoGrid, and similar providers.
- ***global***: Picks a public IP address. This should be avoided for replication traffic.
- ***non-loopback***: Uses the first address found on an active interface that is not a **127.*x*.*x*.*x*** address.

Report a bug

20.3. ABOUT BINDING SOCKETS

20.3.1. About Group and Individual Socket Binding

Group interfaces can be used to bind all socket bindings within a given socket binding group or to bind individual sockets.

Report a bug

20.3.2. Binding a Single Socket Example

The following is an example depicting the use of JGroups interface socket binding to bind an individual socket using the ***socket-binding*** element.

```
<socket-binding name="jgroups-udp" ... interface="site-local"/>
```

Report a bug

20.3.3. Binding a Group of Sockets Example

The following is an example depicting the use of Groups interface socket bindings to bind a group, using the ***socket-binding-group*** element:

```
<socket-binding-group name="ha-sockets" default-interface="global"> ...
</socket-binding-group>
```

Report a bug

20.4. JGROUPS FOR CLUSTERED MODES

20.4.1. Configure JGroups for Clusterd Modes

JBoss Data Grid must have an appropriate JGroups configuration in order to operate in clustered mode.

To configure JGroups programmatically use the following:

```
GlobalConfiguration gc = new GlobalConfigurationBuilder()
  .transport()
  .defaultTransport() // <<< This call is missing...
  .addProperty("configurationFile","jgroups.xml")
  .build();
```

To configure JGroups using XML use the following:

```
<infinispan>
  <global>
    <transport>
      <properties>
        <property name="configurationFile" value="jgroups.xml" />
      </properties>
    </transport>
  </global>

  ...

</infinispan>
```

In either programmatic or XML configuration methods, JBoss Data Grid searches for **`jgroups.xml`** in the classpath before searching for an absolute path name if it is not found in the classpath.

Report a bug

20.4.2. Pre-Configured JGroups Files

20.4.2.1. Using a Pre-Configured JGroups File

JBoss Data Grid ships with a number of pre-configured JGroups files packaged in `infinispan-core.jar`, and are available on the classpath by default. In order to use one of these files, specify one of these file names instead of using `jgroups.xml`.

The JGroups configuration files shipped with JBoss Data Grid are intended to be used as a starting point for a working project. JGroups will usually require fine-tuning for optimal network performance.

Available configurations are:

- `jgroups-udp.xml`
- `jgroups-tcp.xml`

Report a bug

20.4.2.2. jgroups-udp.xml

`jgroups-udp.xml` is a pre-configured JGroups file in JBoss Data Grid. The `jgroups-udp.xml` configuration

- uses UDP as a transport and UDP multicast for discovery.
- is suitable for large clusters (over 100 nodes).
- is suitable if using Invalidation or Replication modes.
- minimizes inefficient use of sockets.

The behavior of some of these settings can be altered by adding certain system properties to the JVM at startup. The settings that can be configured are shown in the following table.

Table 20.1. jgroups-udp.xml System Properties

System Property	Description	Default	Required?
jgroups.udp.mcast_addr	IP address to use for multicast (both for communications and discovery). Must be a valid Class D IP address, suitable for IP multicast.	228.6.7.8	No
jgroups.udp.mcast_port	Port to use for multicast socket	46655	No

System Property	Description	Default	Required?
jgroups.udp.ip_ttl	Specifies the time-to-live (TTL) for IP multicast packets. The value here refers to the number of network hops a packet is allowed to make before it is dropped	2	No

Report a bug

20.4.2.3. jgroups-tcp.xml

`jgroups-tcp.xml` is a pre-configured JGroups file in JBoss Data Grid. The `jgroups-tcp.xml` configuration

- uses TCP as a transport and UDP multicast for discovery.
- is better suited to smaller clusters (under 100 nodes) only when using distribution mode. This is because TCP is more efficient as a point-to-point protocol.

As with other pre-configured JGroups files, the behavior of some of these settings can be altered by adding certain system properties to the JVM at startup. The settings that can be configured are shown in the following table.

Table 20.2. jgroups-udp.xml System Properties

System Property	Description	Default	Required?
jgroups.tcp.address	IP address to use for the TCP transport.	127.0.0.1	No
jgroups.tcp.port	Port to use for TCP socket	7800	No
jgroups.udp.mcast_addr	IP address to use for multicast (for discovery). Must be a valid Class D IP address, suitable for IP multicast.	228.6.7.8	No
jgroups.udp.mcast_port	Port to use for multicast socket	46655	No
jgroups.udp.ip_ttl	Specifies the time-to-live (TTL) for IP multicast packets. The value here refers to the number of network hops a packet is allowed to make before it is dropped	2	No

Report a bug

CHAPTER 21. MANAGEMENT TOOLS IN JBOSS DATA GRID

21.1. JAVA MANAGEMENT EXTENSIONS (JMX)

21.1.1. About Java Management Extensions (JMX)

Java Management Extension (JMX) is a Java based technology that provides tools to manage and monitor applications, devices, system objects and service oriented networks. Each of these objects is managed and monitored by **MBeans**.

JMX is the de facto standard for middleware management and administration. As a result, JMX is used in JBoss Data Grid to expose management and statistical information.

Report a bug

21.1.2. Using JMX with JBoss Data Grid

Management in JBoss Data Grid instances aims to expose as much relevant statistical information as possible. This information allows administrators to view the state of each instance. While a single installation can comprise of tens or hundreds of such instances, it is essential to expose and present the statistical information for each of them in a clear and concise manner.

In JBoss Data Grid, JMX is used in conjunction with JBoss Operations Network (JON) to expose this information and present it in an orderly and relevant manner to the administrator.

Report a bug

21.1.3. JMX Statistic Levels

JMX statistics can be enabled at two levels:

- At the cache level, where management information is generated by individual cache instances.
- At the **CacheManager** level, where the **CacheManager** is the entity that governs all cache instances created from it. As a result, the management information is generated for all these cache instances instead of individual caches.

Report a bug

21.1.4. Enable JMX for Cache Instances

At the Cache level, JMX statistics can be enabled either declaratively or programmatically, as follows.

Enable JMX Declaratively at the Cache Level

Add the following snippet within either the <default> element for the default cache instance, or under the target <namedCache> element for a specific named cache:

```
<jmxStatistics enabled="true"/>
```

Enable JMX Programmatically at the Cache Level

Add the following code to programmatically enable JMX at the cache level:

```
Configuration configuration = ...
configuration.setExposeJmxStatistics(true);
```

Report a bug

21.1.5. Enable JMX for CacheManagers

At the **CacheManager** level, JMX statistics can be enabled either declaratively or programmatically, as follows.

Enable JMX Declaratively at the CacheManager Level

Add the following in the <global> element to enable JMX declaratively at the **CacheManager** level:

```
<globalJmxStatistics enabled="true"/>
```

Enable JMX Programmatically at the CacheManager Level

Add the following code to programmatically enable JMX at the **CacheManager** level:

```
GlobalConfiguration globalConfiguration = ...
globalConfiguration.setExposeGlobalJmxStatistics(true);
```

Report a bug

21.1.6. Multiple JMX Domains

Multiple JMX domains are used when multiple **CacheManager** instances exist on a single virtual machine, or if the names of cache instances in different **CacheManagers** clash.

To resolve this issue, name each **CacheManager** in manner that allows it to be easily identified and used by monitoring tools such as JMX and JBoss Operations Network.

Set a CacheManager Name Declaratively

Add the following snippet to the relevant **CacheManager** configuration:

```
<globalJmxStatistics enabled="true" cacheManagerName="Hibernate2LC"/>
```

Set a CacheManager Name Programmatically

Add the following code to set the **CacheManager** name programmatically:

```
GlobalConfiguration globalConfiguration = ...
globalConfiguration.setExposeGlobalJmxStatistics(true);
globalConfiguration.setCacheManagerName("Hibernate2LC");
```

Report a bug

21.1.7. About MBeans

An **MBean** represents a manageable resource such as a service, component, device or an application.

JBoss Data Grid provides **MBeans** that monitor and manage multiple aspects. For example, **MBeans**

that provide statistics on the transport layer are provided. If a JBoss Data Grid server is configured with JMX statistics, an **MBean** that provides information such as the hostname, port, bytes read, bytes written and the number of worker threads exists at the following location:

```
jboss.infinispan:type=Server,name=<Memcached|Hotrod>,component=Transport
```

Report a bug

21.1.8. Understanding MBeans

When JMX reporting is enabled at either the Cache Manager or Cache level, use a standard JMX GUI such as JConsole or VisualVM to connect to a Java Virtual Machine running JBoss Data Grid. When connected, the following **MBeans** are available:

- If Cache Manager-level JMX statistics are enabled, an **MBean** named ***jboss.infinispan:type=CacheManager,name="DefaultCacheManager"*** exists, with properties specified by the Cache Manager **MBean**.
- If the cache-level JMX statistics are enabled, multiple **MBeans** display depending on the configuration in use. For example, if a write behind cache store is configured, an **MBean** that exposes properties that belong to the cache store component is displayed. All cache-level **MBeans** use the same format:

  ```
  jboss.infinispan:type=Cache,name="<name-of-cache>(<cache-
  mode>)",manager="<name-of-cache-manager>",component=<component-name>
  ```

  In this format:

  - Specify the default name for the cache using the **cache-container** element's ***default-cache*** attribute.
  - The ***cache-mode*** is replaced by the cache mode of the cache. The lower case version of the possible enumeration values represents the cache mode.
  - The ***component-name*** is replaced by one of the JMX component names from the JMX reference documentation.

As an example, the cache store JMX component **MBean** for a default cache configured for synchronous distribution would be named as follows:

```
jboss.infinispan:type=Cache,name="default(dist_sync)",
manager="default",component=CacheStore
```

Each cache and cache manager name is within quotation marks to prevent the use of unsupported characters in these user-defined names.

Report a bug

21.1.9. Registering MBeans in Non-Default MBean Servers

The default location where all the MBeans used are registered is the standard JVM MBeanServer platform. Users can set up an alternative MBeanServer instance as well. Implement the MBeanServerLookup interface to ensure that the **getMBeanServer()** method returns the desired (non default) MBeanServer.

To set up a non default location to register your MBeans, create the implementation and then configure JBoss Data Grid with the fully qualified name of the class. An example is as follows:

To Add the Fully Qualified Domain Name Declaratively

Add the following snippet:

```
<globalJmxStatistics enabled="true"
mBeanServerLookup="com.acme.MyMBeanServerLookup"/>
```

To Add the Fully Qualified Domain Name Programmatically

Add the following code:

```
GlobalConfiguration globalConfiguration = ...
globalConfiguration.setExposeGlobalJmxStatistics(true);
globalConfiguration.setMBeanServerLookup("com.acme.MyMBeanServerLookup")
```

Report a bug

21.2. JBOSS OPERATIONS NETWORK (JON)

21.2.1. About JBoss Operations Network (JON)

The JBoss Operations Network (JON) is JBoss' administration and management platform used to develop, test, deploy and monitor the application life cycle. JBoss Operations Network is JBoss' enterprise management solution and is recommended for the management of multiple JBoss Data Grid instances across servers. JBoss Operations Network's agent and auto discovery features facilitate monitoring the Cache Manager and Cache instances in JBoss Data Grid. JBoss Operations Network presents graphical views of key runtime parameters and statistics and allows administrators to set thresholds and be notified if usage exceeds or falls under the set thresholds.

Report a bug

21.2.2. JBoss Operations Network (JON) and JMX

For management purposes, JBoss Data Grid exposes JMX statistical information that is formatted for consumption by JBoss Operations Network.

Report a bug

21.2.3. Configure JBoss Operations Network (JON)

To configure JBoss Operations Network to monitor JBoss Data Grid instances, use the following steps:

Procedure 21.1. Configure JBoss Operations Network

1. Download and install a JBoss Operations Network server and initiate one or more JBoss Operations Network agents. The JBoss Operations Network agent is responsible for relaying information about the JBoss Data Grid instance to the server. The server displays the received information using the GUI. The recommended set up is to install the JBoss Operations Network agent in the machine that runs JBoss Data Grid. If multiple machines are available, an agent can exist on each machine.

2. Download the most recent JBoss Data Grid version. In the **`modules/rhq-plugin`** directory,

locate the JBoss Operations Network plug-in jar file (named **`infinispan-rhq-plugin.jar`**).

3. JBoss Operations Network can now monitor JBoss Data Grid instances.

4. After each JBoss Data Grid instance is discovered, JBoss Operations Network displays a new resource in the JBoss Operations Network server's Inventory/Discovery Queue for each operational cache manager.

5. Import a displayed resource. Each cache manager consequently displays as many child cache resources as caches running in the cache manager.

Result

You can now use JBoss Operations Network to monitor JBoss Data Grid.

Report a bug

21.2.4. JBoss Operations Network Plug-in Quickstart

For testing or demonstrative purposes with a single JBoss Operations Network agent, upload the plug-in to the server then type "plugins update" at the agent command line to force a retrieval of the latest plugins from the server.

Report a bug

21.2.5. Remote JMX Port Values

A port value must be provided to allow JBoss Data Grid instances to be located. The value itself can be any available port.

Provide unique (and available) remote JMX ports to run multiple JBoss Data Grid instances on a single machine. A locally running JBoss Operations Network agent can discover each instance using the remote port values.

Report a bug

APPENDIX A. REFERENCES

A.1. ABOUT APACHE LUCENE INDEX

Apache Lucene is a text based searching framework that creates indexes using searchable data. An Apache Lucene Index can then be queried for data.

Report a bug

A.2. ABOUT CONSISTENCY

Consistency is the policy that states whether it is possible for a data record on one node to vary from the same data record on another node.

For example, due to network speeds, it is possible that a write operation performed on the master node has not yet been performed on another node in the store, a strong consistency guarantee will ensure that data which is not yet fully replicated is not returned to the application.

Report a bug

A.3. ABOUT CONSISTENCY GUARANTEE

Despite the locking of a single owner instead of all owners, JBoss Data Grid's consistency guarantee remains intact. The consistency guarantee is as follows:

1. If Key **K** is hashed to nodes **{A, B}** and transaction **TX1** acquires a lock for **K** on, for example, node **A**.

2. If another cache access occurs on node **B**, or any other node, and **TX2** attempts to lock **K**, it fails with a timeout because the transaction **TX1** already holds a lock on **K**.

This lock acquisition attempt always fails because the lock for key **K** is always deterministically acquired on the same node of the cluster, irrespective of the transaction's origin.

Report a bug

A.4. ABOUT JAVA MANAGEMENT EXTENSIONS (JMX)

Java Management Extension (JMX) is a Java based technology that provides tools to manage and monitor applications, devices, system objects and service oriented networks. Each of these objects is managed and monitored by **MBeans**.

JMX is the de facto standard for middleware management and administration. As a result, JMX is used in JBoss Data Grid to expose management and statistical information.

Report a bug

A.5. ABOUT JBOSS CACHE

JBoss Cache is a tree-structured, clustered, transactional cache that can also be used in a standalone, non-clustered environment. It caches frequently accessed data in-memory to prevent data retrieval or calculation bottlenecks that occur while enterprise features such as Java Transactional API (JTA) compatibility, eviction and persistence are provided.

JBoss Cache is the predecessor to Infinispan and JBoss Data Grid.

Report a bug

A.6. ABOUT JSON

The JavaScript Object Notation (JSON) is a lightweight data exchange format. It is readable and writable by both humans and machines. Despite being derived from JavaScript, JSON is language-independent.

Report a bug

A.7. ABOUT RETURN VALUES

Values returned by cache operations are referred to as return values. In JBoss Data Grid, these return values remain reliable irrespective of which cache mode is employed and whether synchronous or asynchronous communication is used.

Report a bug

A.8. ABOUT RUNNABLE INTERFACES

A Runnable Interface (also known as a Runnable) declares a single `run()` method, which executes the active part of the class' code. The Runnable object can be executed in its own thread after it is passed to a thread constructor.

Report a bug

A.9. ABOUT TWO PHASE COMMIT (2PC)

A Two Phase Commit protocol (2PC) is a consensus protocol used to atomically commit or roll back distributed transactions. It is successful when faced with cases of temporary system failures, including network node and communication failures, and is therefore widely utilized.

Report a bug

A.10. ABOUT KEY-VALUE PAIRS

A key-value pair (KVP) is a set of data consisting of a key and a value.

- A key is unique to a particular data entry and is composed from data attributes of the particular entry it relates to.
- A value is the data assigned to and identified by the key.

Report a bug

A.11. THE EXTERNALIZER

A.11.1. About Externalizer

An `Externalizer` is a class that can:

- Marshall a given object type to a byte array.
- Unmarshall the contents of a byte array into an instance of the object type.

Externalizers are used by JBoss Data Grid and allow users to specify how their object types are serialized. The marshalling infrastructure used in JBoss Data Grid builds upon JBoss Marshalling and provides efficient payload delivery and allows the stream to be cached. The stream caching allows data to be accessed multiple times, whereas normally a stream can only be read once.

Report a bug

A.11.2. Internal Externalizer Implementation Access

Externalizable objects should not access JBoss Data Grids Externalizer implementations. The following is an example of the incorrect method to deal with this:

```
public static class ABCMarshallingExternalizer implements
AdvancedExternalizer<ABCMarshalling> {
   @Override
   public void writeObject(ObjectOutput output, ABCMarshalling object)
throws IOException {
      MapExternalizer ma = new MapExternalizer();
      ma.writeObject(output, object.getMap());
   }

   @Override
   public ABCMarshalling readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
      ABCMarshalling hi = new ABCMarshalling();
      MapExternalizer ma = new MapExternalizer();
      hi.setMap((ConcurrentHashMap<Long, Long>) ma.readObject(input));
      return hi;
   }

   ...
```

End user externalizers do not need to interact with internal externalizer classes. The following is an example of the correct method to deal with this situation:

```
public static class ABCMarshallingExternalizer implements
AdvancedExternalizer<ABCMarshalling> {
   @Override
   public void writeObject(ObjectOutput output, ABCMarshalling object)
throws IOException {
      output.writeObject(object.getMap());
   }

   @Override
   public ABCMarshalling readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
      ABCMarshalling hi = new ABCMarshalling();
      hi.setMap((ConcurrentHashMap<Long, Long>) input.readObject());
      return hi;
   }
```

```
   ...
}
```

Report a bug

A.12. HASH SPACE ALLOCATION

A.12.1. About Hash Space Allocation

JBoss Data Grid is responsible for allocating a portion of the total available hash space to each node. During subsequent operations that must store an entry, JBoss Data Grid creates a hash of the relevant key and stores the entry on the node that owns that portion of hash space.

Report a bug

A.12.2. Locating a Key in the Hash Space

JBoss Data Grid always uses an algorithm to locate a key in the hash space. As a result, the node that houses the key is never manually specified. This scheme allows any node to know which node owns a particular key without such ownership information being distributed. This scheme reduces the amount of overhead and, more importantly, improves redundancy because the ownership information does not need to be replicated in case of node failure.

Report a bug

A.12.3. Requesting a Full Byte Array

How can I request the JBoss Data Grid return a full byte array instead of partial byte array contents?

As a default, JBoss Data Grid only partially prints byte arrays to logs to avoid unnecessarily printing large byte arrays. This occurs when either:

- JBoss Data Grid caches are configured for lazy deserialization. Lazy deserialization is not available in JBoss Data Grid's Remote Client-Server mode.
- A **Memcached** or **Hot Rod** server is run.

In such cases, only the first ten positions of the byte array display in the logs. To display the complete contents of the byte array in the logs, pass the ***-Dinfinispan.arrays.debug=true*** system property at start up.

Example A.1. Partial Byte Array Log

```
2010-04-14 15:46:09,342 TRACE [ReadCommittedEntry] (HotRodWorker-1-1)
Updating entry
(key=CacheKey{data=ByteArray{size=19, hashCode=1b3278a,
array=[107, 45, 116, 101, 115, 116, 82, 101, 112, 108, ..]}}
removed=false valid=true changed=true created=true
value=CacheValue{data=ByteArray{size=19,
array=[118, 45, 116, 101, 115, 116, 82, 101, 112, 108, ..]},
version=281483566645249}]
And here's a log message where the full byte array is shown:
2010-04-14 15:45:00,723 TRACE [ReadCommittedEntry] (Incoming-
```

```
2,Infinispan-Cluster,eq-6834) Updating entry
(key=CacheKey{data=ByteArray{size=19, hashCode=6cc2a4,
array=[107, 45, 116, 101, 115, 116, 82, 101, 112, 108, 105, 99, 97, 116,
101, 100, 80, 117, 116]}}
removed=false valid=true changed=true created=true
value=CacheValue{data=ByteArray{size=19,
array=[118, 45, 116, 101, 115, 116, 82, 101, 112, 108, 105, 99, 97, 116,
101, 100, 80, 117, 116]},
version=281483566645249}]
```

Report a bug

APPENDIX B. REVISION HISTORY

Revision 1.1-34 **Thu Jan 30 2014** **Misha Husnain Ali**

Removed unsupported information.

Revision 1.1-33.400 **2013-10-31** **Rüdiger Landmann**

Rebuild with publican 4.0.0

Revision 1.1-33 **Tue Aug 06 2013** **Misha Husnain Ali**

Built with new product name.

Revision 1.1-32 **Mon Jan 21 2013** **Misha Husnain Ali**

Minor update for BZ#886969 in the Transactions topic.