# Red Hat JBoss Data Grid 6.3

## Administration and Configuration Guide

For use with Red Hat JBoss Data Grid 6.3.2

Last Updated: 2017-11-14

# Red Hat JBoss Data Grid 6.3 Administration and Configuration Guide

For use with Red Hat JBoss Data Grid 6.3.2

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

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

Rakesh Ghatvisave
Red Hat Engineering Content Services
rghatvis@redhat.com

## Abstract

This guide presents information about the administration and configuration of Red Hat JBoss Data Grid 6.3.2

# Table of Contents

# CHAPTER 1. SETTING UP RED HAT JBOSS DATA GRID

## 1.1. PREREQUISITES

The only prerequisites to set up Red Hat JBoss Data Grid is a Java Virtual Machine and that the most recent supported version of the product is installed on your system.

Report a bug

## 1.2. STEPS TO SET UP RED HAT JBOSS DATA GRID

The following steps outline the necessary (and optional, where stated) steps for a first time basic configuration of Red Hat JBoss Data Grid. It is recommended that the steps are followed in the order specified and not skipped unless they are identified as optional steps.

**Procedure 1.1. Set Up JBoss Data Grid**

1. **Set Up the Cache Manager**
   The first step in a JBoss Data Grid configuration is a cache manager. Cache managers can retrieve cache instances and create cache instances quickly and easily using previously specified configuration templates. For details about setting up a cache manager, see Part I, "Set Up a Cache Manager"

2. **Set Up JVM Memory Management**
   An important step in configuring your JBoss Data Grid is to set up memory management for your Java Virtual Machine (JVM). JBoss Data Grid offers features such as eviction and expiration to help manage the JVM memory.

   a. **Set Up Eviction**
      Use eviction to specify the logic used to remove entries from the in-memory cache implementation based on how often they are used. JBoss Data Grid offers different eviction strategies for finer control over entry eviction in your data grid. Eviction strategies and instructions to configure them are available in Chapter 3, *Set Up Eviction*.

   b. **Set Up Expiration**
      To set upper limits to an entry's time in the cache, attach expiration information to each entry. Use expiration to set up the maximum period an entry is allowed to remain in the cache and how long the retrieved entry can remain idle before being removed from the cache. For details, see Chapter 4, *Set Up Expiration*

3. **Monitor Your Cache**
   JBoss Data Grid uses logging via JBoss Logging to help users monitor their caches.

   a. **Set Up Logging**
      It is not mandatory to set up logging for your JBoss Data Grid, but it is highly recommended. JBoss Data Grid uses JBoss Logging, which allows the user to easily set up automated logging for operations in the data grid. Logs can subsequently be used to troubleshoot errors and identify the cause of an unexpected failure. For details, see Chapter 5, *Set Up Logging*

4. **Set Up Cache Modes**
   Cache modes are used to specify whether a cache is local (simple, in-memory cache) or a clustered cache (replicates state changes over a small subset of nodes). Additionally, if a cache is clustered, either replication, distribution or invalidation mode must be applied to

determine how the changes propagate across the subset of nodes. For details, see Part IV, “Set Up Cache Modes”

5. **Set Up Locking for the Cache**
   When replication or distribution is in effect, copies of entries are accessible across multiple nodes. As a result, copies of the data can be accessed or modified concurrently by different threads. To maintain consistency for all copies across nodes, configure locking. For details, see Part VI, “Set Up Locking for the Cache” and Chapter 16, *Set Up Isolation Levels*

6. **Set Up and Configure a Cache Store**
   JBoss Data Grid offers the passivation feature (or cache writing strategies if passivation is turned off) to temporarily store entries removed from memory in a persistent, external cache store. To set up passivation or a cache writing strategy, you must first set up a cache store.

   a. **Set Up a Cache Store**
      The cache store serves as a connection to the persistent store. Cache stores are primarily used to fetch entries from the persistent store and to push changes back to the persistent store. For details, see Part VII, “Set Up and Configure a Cache Store”

   b. **Set Up Passivation**
      Passivation stores entries evicted from memory in a cache store. This feature allows entries to remain available despite not being present in memory and prevents potentially expensive write operations to the persistent cache. For details, see Part VIII, “Set Up Passivation”

   c. **Set Up a Cache Writing Strategy**
      If passivation is disabled, every attempt to write to the cache results in writing to the cache store. This is the default Write-Through cache writing strategy. Set the cache writing strategy to determine whether these cache store writes occur synchronously or asynchronously. For details, see Part IX, “Set Up Cache Writing”

7. **Monitor Caches and Cache Managers**
   JBoss Data Grid includes two primary tools to monitor the cache and cache managers once the data grid is up and running.

   a. **Set Up JMX**
      JMX is the standard statistics and management tool used for JBoss Data Grid. Depending on the use case, JMX can be configured at a cache level or a cache manager level or both. For details, see Chapter 21, *Set Up Java Management Extensions (JMX)*

   b. **Set Up Red Hat JBoss Operations Network (JON)**
      Red Hat JBoss Operations Network (JON) is the second monitoring solution available for JBoss Data Grid. JBoss Operations Network (JON) offers a graphical interface to monitor runtime parameters and statistics for caches and cache managers. For details, see Chapter 22, *Set Up JBoss Operations Network (JON)*

8. **Introduce Topology Information**
   Optionally, introduce topology information to your data grid to specify where specific types of information or objects in your data grid are located. Server hinting is one of the ways to introduce topology information in JBoss Data Grid.

   a. **Set Up Server Hinting**
      When set up, server hinting provides high availability by ensuring that the original and backup copies of data are not stored on the same physical server, rack or data center. This is optional in cases such as a replicated cache, where all data is backed up on all servers, racks and data centers. For details, see Chapter 28, *High Availability Using Server Hinting*

The subsequent chapters detail each of these steps towards setting up a standard JBoss Data Grid configuration.

Report a bug

# PART I. SET UP A CACHE MANAGER

# CHAPTER 2. CACHE MANAGERS

A Cache Manager is the primary mechanism to retrieve a cache instance in Red Hat JBoss Data Grid, and is a starting point for using the cache.

In JBoss Data Grid, a cache manager is useful because:

- it creates multiple cache instances on demand using a provided standard.
- it retrieves existing cache instances (i.e. caches that have already been created).

Report a bug

## 2.1. TYPES OF CACHE MANAGERS

Red Hat JBoss Data Grid offers the following Cache Managers:

- **`EmbeddedCacheManager`** is a cache manager that runs within the Java Virtual Machine (JVM) used by the client. Currently, JBoss Data Grid offers only the **`DefaultCacheManager`** implementation of the **`EmbeddedCacheManager`** interface.
- **`RemoteCacheManager`** is used to access remote caches. When started, the **`RemoteCacheManager`** instantiates connections to the Hot Rod server (or multiple Hot Rod servers). It then manages the persistent **TCP** connections while it runs. As a result, **`RemoteCacheManager`** is resource-intensive. The recommended approach is to have a single **`RemoteCacheManager`** instance for each Java Virtual Machine (JVM).

Report a bug

## 2.2. CREATING CACHEMANAGERS

### 2.2.1. Create a New RemoteCacheManager

**Example 2.1. Configure a New `RemoteCacheManager`**

```
import org.infinispan.client.hotrod.configuration.Configuration;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;

Configuration conf = new
ConfigurationBuilder().addServer().host("localhost").port(11222).build()
;
RemoteCacheManager manager = new RemoteCacheManager(conf);
RemoteCache defaultCache = manager.getCache();
```

**Configuration Explanation**

An explanation of each line of the provided configuration is as follows:

1. Use the **`ConfigurationBuilder()`** method to configure a new builder. The ***`.addServer()`*** property adds a remote server, specified via the ***`.host(<hostname|ip>)`*** and ***`.port(<port>)`*** properties.

```
Configuration conf = new
ConfigurationBuilder().addServer().host(<hostname|ip>).port(<port>).
build();
```

2. Create a new **RemoteCacheManager** using the supplied configuration.

```
RemoteCacheManager manager = new RemoteCacheManager(conf);
```

3. Retrieve the default cache from the remote server.

```
RemoteCache defaultCache = manager.getCache();
```

Report a bug

### 2.2.2. Create a New Embedded Cache Manager

Use the following instructions to create a new EmbeddedCacheManager without using CDI:

**Procedure 2.1. Create a New Embedded Cache Manager**

1. Create a configuration XML file. For example, create the **my-config.file.xml** file on the classpath (in the **resources/** folder) and add the configuration information in this file.

2. Use the following programmatic configuration to create a cache manager using the configuration file:

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-
file.xml");
Cache defaultCache = manager.getCache();
```

The outlined procedure creates a new EmbeddedCacheManager using the basic configuration specified.

Report a bug

### 2.2.3. Create a New Embedded Cache Manager Using CDI

Use the following steps to create a new EmbeddedCacheManager instance using CDI:

**Procedure 2.2. Use CDI to Create a New EmbeddedCacheManager**

1. Specify a default configuration:

```
public class Config
   @Produces
   public EmbeddedCacheManager defaultCacheManager() {
      Configuration configuration = new ConfigurationBuilder();

builder.eviction().strategy(EvictionStrategy.LRU).maxEntries(100).bu
ild();
```

```
        return new DefaultCacheManager(configuration);
    }
}
```

2. Inject the default cache manager.

```
...
@Inject
EmbeddedCacheManager cacheManager;
...
```

Report a bug

## 2.3. MULTIPLE CACHE MANAGERS

Cache managers are an entry point to the cache and Red Hat JBoss Data Grid allows users to create multiple cache managers. Each cache manager is configured with a different global configuration, which includes settings for things like JMX, executors and clustering.

Report a bug

### 2.3.1. Create Multiple Caches with a Single Cache Manager

Red Hat 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

### 2.3.2. Using Multiple Cache Managers

Red Hat JBoss Data Grid allows multiple cache managers to be used. In most cases, such as with replication 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

### 2.3.3. Create Multiple Cache Managers

Red Hat JBoss Data Grid allows users to create multiple cache managers of various types by repeating the procedure used to create the first cache manager (and adjusting the contents of the configuration file, if required).

To use the declarative API to create multiple new cache managers, copy the contents of the **infinispan.xml** file to a new configuration file. Edit the new file for the desired configuration and then use the new file for a new cache manager.

Report a bug

# PART II. SET UP JVM MEMORY MANAGEMENT

# CHAPTER 3. SET UP EVICTION

## 3.1. 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.

Red Hat 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.

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.

In JBoss Data Grid, eviction provides a mechanism to efficiently remove entries from the in-memory representation of a cache and push them to a persistent store. This ensures that the memory can always accommodate new entries as they are fetched and that evicted entries are preserved in the cluster instead of lost.

Additionally, eviction strategies can be used as required for your configuration to set up which entries are evicted and when eviction occurs.

**See Also:**

- Section 4.3, “Eviction and Expiration Comparison”

Report a bug

## 3.2. EVICTION STRATEGIES

Each eviction strategy has specific benefits and use cases, as outlined below:

**Table 3.1. Eviction Strategies**

| Strategy Name | Operations | Details |
|---|---|---|
| `EvictionStrategy.NONE` | No eviction occurs. | This is the default eviction strategy in Red Hat JBoss Data Grid. |
| `EvictionStrategy.LRU` | Least Recently Used eviction strategy. This strategy evicts entries that have not been used for the longest period. This ensures that entries that are reused periodically remain in memory. | |

| Strategy Name | Operations | Details |
|---|---|---|
| `EvictionStrategy.UNORDERED` | Unordered eviction strategy. This strategy evicts entries without any ordered algorithm and may therefore evict entries that are required later. However, this strategy saves resources because no algorithm related calculations are required before eviction. | This strategy is recommended for testing purposes and not for a real work implementation. |
| `EvictionStrategy.LIRS` | Low Inter-Reference Recency Set eviction strategy. | LIRS is an eviction algorithm that suits a large variety of production use cases. |

Report a bug

### 3.2.1. LRU Eviction Algorithm Limitations

In the Least Recently Used (LRU) eviction algorithm, the least recently used entry is evicted first. The entry that has not been accessed the longest gets evicted first from the cache. However, LRU eviction algorithm sometimes does not perform optimally in cases of weak access locality. The weak access locality is a technical term used for entries which are put in the cache and not accessed for a long time and entries to be accessed soonest are replaced. 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

## 3.3. USING EVICTION

In Red Hat JBoss Data Grid, eviction is disabled by default. If an empty *`<eviction />`* element is used to enable eviction without any strategy or maximum entries settings, the following default values are automatically implemented:

- Strategy: If no eviction strategy is specified, *`EvictionStrategy.NONE`* is assumed as a default.
- max-entries/maxEntries: If no value is specified, the *`max-entries`*/maxEntries value is set to `-1`, which allows unlimited entries.

Report a bug

### 3.3.1. 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`*, Red Hat JBoss Data Grid runs out of memory.

The following procedure outlines the steps to initialize eviction in JBoss Data Grid:

**Procedure 3.1. Initialize Eviction**

1. **Add the Eviction Tag**
   Add the <eviction> tag to your project's <cache> tags as follows:

   ```
   <eviction />
   ```

2. **Set the Eviction Strategy**
   Set the ***strategy*** value to set the eviction strategy employed. Possible values are **LRU**, **UNORDERED** and **LIRS** (or **NONE** if no eviction is required). The following is an example of this step:

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

3. **Set the Maximum Entries**
   Set the maximum number of entries allowed in memory. The default value is **-1** for unlimited entries.

   a. In Library mode, set the ***maxEntries*** parameter as follows:

      ```
      <eviction strategy="LRU" maxEntries="200" />
      ```

   b. In Remote Client Server mode, set the ***max-entries*** as follows:

      ```
      <eviction strategy="LRU" max-entries="200" />
      ```

**Result**

Eviction is configured for the target cache.

Report a bug

### 3.3.2. Eviction Configuration Examples

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

- A sample XML configuration for Library mode is as follows:

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

- A sample XML configuration for Remote Client Server Mode is as follows:

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

- A sample programmatic configuration for Library Mode is as follows:

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

```
                    .build();
```

**NOTE**

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

### 3.3.3. Eviction Configuration Troubleshooting

In Red Hat 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

### 3.3.4. 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

# CHAPTER 4. SET UP EXPIRATION

## 4.1. ABOUT EXPIRATION

Red Hat 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.

Expiration can be specified on a per-entry or per-cache basis and the per-entry configuration overrides per-cache configurations. If expiration is not configured at the cache level, cache entries are created immortal (i.e. they will never expire) as a default. Conversely, if expiration is configured at the cache level, the expiration defaults apply to all entries which do not explicitly specify a ***`lifespan`*** or ***`maxIdle`*** value.

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

Expiration automates the removal of entries that have not been used for a specified period of time from the memory. Expiration and eviction are different because:

- expiration removes entries based on the period they have been in memory. Expiration only removes entries when the life span period concludes or when an entry has been idle longer than the specified idle time.
- eviction removes entries based on how recently (and often) they are used. Eviction only removes entries when too many entries are present in the memory. If a cache store has been configured, evicted entries are persisted in the cache store.

Report a bug

## 4.2. EXPIRATION OPERATIONS

Expiration in Red Hat 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

## 4.3. EVICTION AND EXPIRATION COMPARISON

Expiration is a top-level construct in Red Hat 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

## 4.4. CACHE ENTRY EXPIRATION NOTIFICATIONS

Red Hat 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:

- 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.

If a user requests an entry that is expired but not yet evicted, a null value is sent to the user. This mechanism ensures that the user never receives an expired entry. The entry is eventually removed by the eviction thread.

Report a bug

## 4.5. CONFIGURE EXPIRATION

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

**Procedure 4.1. Configure Expiration**

1. **Add the Expiration Tag**
   Add the <expiration> tag to your project's <cache> tags as follows:

   ```
   <expiration />
   ```

2. **Set the Expiration Lifespan**
   Set the ***lifespan*** value to set the period of time (in milliseconds) an entry can remain in memory. The following is an example of this step:

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

3. **Set the Maximum Idle Time**
   Set the time that entries are allowed to remain idle (unused) after which they are removed (in milliseconds). The default value is **-1** for unlimited time.

   a. In Library mode, set the ***maxIdle*** parameter as follows:

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

      - In Remote Client Server mode, set the ***max-idle*** as follows:

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

**Result**

Expiration is now configured for the cache implementation.

Report a bug

## 4.6. MORTAL AND IMMORTAL DATA

In addition to storing entities, Red Hat 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.

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.

Set entry mortality (or its expiration values) 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

## 4.7. TROUBLESHOOTING EXPIRATION

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 Red Hat 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 as follows:

- Enabling the eviction feature causes the eviction thread to periodically detect and purge expired entries.

Any attempt to use **`get()`** or **`containsKey()`** for the expired entry causes JBoss Data Grid to return a null value. The expired entry is later removed by the eviction thread.

Report a bug

# PART III. MONITOR YOUR CACHE

# CHAPTER 5. SET UP LOGGING

## 5.1. ABOUT LOGGING

Red Hat JBoss Data Grid 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).

After a JBoss Data Grid cache is configured with operations such as eviction and expiration, logging tracks relevant activity (including errors or failures).

When set up correctly, logging provides a detailed account at what occurred in the environment and when. Logging also helps track activity that occurred just before a crash or problem in the environment. This information is useful when troubleshooting or when attempting to identify the source of a crash or error.

Report a bug

## 5.2. SUPPORTED APPLICATION LOGGING FRAMEWORKS

Red Hat JBoss LogManager supports the following logging frameworks:

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

Report a bug

### 5.2.1. 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, Red Hat JBoss Data Grid 6 also uses JBoss Logging.

JBoss Logging provides an easy way to add logging to an application. Add code to the 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

### 5.2.2. 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

## 5.3. BOOT LOGGING

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

Report a bug

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

Report a bug

### 5.3.2. Default Log File Locations

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

**Table 5.1. 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

## 5.4. LOGGING ATTRIBUTES

### 5.4.1. 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.

Red Hat JBoss Data Grid 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 severity):

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 numeric value 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

## 5.4.2. Supported Log Levels

The following table lists log levels that are supported in Red Hat 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 5.2. 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. |

| Log Level | Value | Description |
|---|---|---|
| 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 | - |
| 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. |
| SEVERE | 1000 | - |
| FATAL | 1100 | Used to indicate events that could cause critical service failure and application shutdown and possibly cause JBoss Data Grid to shut down. |

Report a bug

### 5.4.3. 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 higher (with greater or equal numeric value) are captured by the log category and sent to the specified log handlers. As an example, the **WARNING** log level results in log values of **900**, **1000** and **1100** are captured.

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

Report a bug

### 5.4.4. 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

### 5.4.5. About Log Handlers

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

- **Console**
- **File**
- **Periodic**
- **Size**
- **Async**
- **Custom**

Log handlers direct specified log objects to a variety of outputs (including the console or specified log files). Some log handlers used in JBoss Data Grid are wrapper log handlers, used to direct other log handlers' behavior.

Log handlers are used to direct log outputs to specific files for easier sorting or to write logs for specific intervals of time. They are primarily useful to specify the kind of logs required and where they are stored or displayed or the logging behavior in JBoss Data Grid.

Report a bug

### 5.4.6. Log Handler Types

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

**Table 5.3. Log Handler Types**

| Log Handler Type | Description | Use Case |
|---|---|---|

| Log Handler Type | Description | Use Case |
|---|---|---|
| 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 is run from a command line prompt. | The Console log handler is preferred when JBoss Data Grid is administered using the command line. In such a case, 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. | File log handlers are most useful if the requirement is to store all log entries according to the time in one place. |
| 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. | The Periodic file handler can be used to accumulate log messages on a weekly, daily, hourly or other basis depending on the requirements of the environment. |
| 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. | The Size handler is best suited to an environment where the log file size must be consistent. |
| 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. | The Async log handlers are best suited to an environment where high latency is a problem or when writing to a network file system. |

| Log Handler Type | Description | Use Case |
|---|---|---|
| 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. | Custom log handlers create customized log handler types and are recommended for advanced users. |

Report a bug

### 5.4.7. Selecting Log Handlers

The following are the most common uses for each of the log handler types available for Red Hat JBoss Data Grid:

- The **`Console`** log handler is preferred when JBoss Data Grid is administered using the command line. In such a case, errors and log messages appear on the console window and are not saved unless separately configured to do so.
- The **`File`** log handler is used to direct log entries into a specified file. This simplicity is useful if the requirement is to store all log entries according to the time in one place.
- The **`Periodic`** log handler is similar to the **`File`** handler but creates files according to the specified period. As an example, this handler can be used to accumulate log messages on a weekly, daily, hourly or other basis depending on the requirements of the environment.
- The **`Size`** log handler also writes log messages to a specified file, but only while the log file size is within a specified limit. Once the file size reaches the specified limit, log files are written to a new log file. This handler is best suited to an environment where the log file size must be consistent.
- The **`Async`** log handler is a wrapper that forces other log handlers to operate asynchronously. This is best suited to an environment where high latency is a problem or when writing to a network file system.
- The **`Custom`** log handler creates new, customized types of log handlers. This is an advanced log handler.

Report a bug

### 5.4.8. 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.

See http://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html for more information.

Report a bug

## 5.5. LOGGING SAMPLE CONFIGURATIONS

### 5.5.1. Sample XML Configuration for the Root Logger

The following procedure demonstrates a sample configuration for the root logger.

**Procedure 5.1. Configure the Root Logger**

1. **Set the *`level`* Property**
   The *`level`* property sets the maximum level of log message that the root logger records.

   ```
   <subsystem xmlns="urn:jboss:domain:logging:1.2">
      <root-logger>
         <level name="INFO"/>
   ```

2. **List *`handlers`***
   *`handlers`* is a list of log handlers that are used by the root logger.

   ```
   <subsystem xmlns="urn:jboss:domain:logging:1.2">
        <root-logger>
           <level name="INFO"/>
           <handlers>
              <handler name="CONSOLE"/>
              <handler name="FILE"/>
           </handlers>
        </root-logger>
     </subsystem>
   ```

Report a bug

### 5.5.2. Sample XML Configuration for a Log Category

The following procedure demonstrates a sample configuration for a log category.

**Procedure 5.2. Configure a Log Category**

1. **Define the Category**
   Use the *`category`* property to specify the log category from which log messages will be captured.

   The *`use-parent-handlers`* is set to **`"true"`** by default. When set to **`"true"`**, this category will use the log handlers of the root logger in addition to any other assigned handlers.

   ```
   <subsystem xmlns="urn:jboss:domain:logging:1.2">
      <logger category="com.company.accounts.rec" use-parent-
   handlers="true">
   ```

2. **Set the *`level`* property**
   Use the *`level`* property to set the maximum level of log message that the log category records.

   ```
   <subsystem xmlns="urn:jboss:domain:logging:1.2">
   ```

```
   <logger category="com.company.accounts.rec" use-parent-
handlers="true">
      <level name="WARN"/>
```

3. **List *handlers***
   ***handlers*** is a list of log handlers.

   ```
   <subsystem xmlns="urn:jboss:domain:logging:1.2">
      <logger category="com.company.accounts.rec" use-parent-
   handlers="true">
         <level name="WARN"/>
         <handlers>
            <handler name="accounts-rec"/>
         </handlers>
      </logger>
   </subsystem>
   ```

Report a bug

### 5.5.3. Sample XML Configuration for a Console Log Handler

The following procedure demonstrates a sample configuration for a console log handler.

**Procedure 5.3. Configure the Console Log Handler**

1. **Add the Log Handler Identifier Information**
   The ***name*** property sets the unique identifier for this log handler.

   When ***autoflush*** is set to **"true"** the log messages will be sent to the handler's target immediately upon request.

   ```
   <subsystem xmlns="urn:jboss:domain:logging:1.2">
      <console-handler name="CONSOLE" autoflush="true">
   ```

2. **Set the *level* Property**
   The ***level*** property sets the maximum level of log messages recorded.

   ```
   <subsystem xmlns="urn:jboss:domain:logging:1.2">
      <console-handler name="CONSOLE" autoflush="true">
         <level name="INFO"/>
   ```

3. **Set the *encoding* Output**
   Use ***encoding*** to set the character encoding scheme to be used for the output.

   ```
   <subsystem xmlns="urn:jboss:domain:logging:1.2">
      <console-handler name="CONSOLE" autoflush="true">
         <level name="INFO"/>
         <encoding value="UTF-8"/>
   ```

4. **Define the *target* Value**

The ***target*** property defines the system output stream where the output of the log handler goes. This can be **System.err** for the system error stream, or **System.out** for the standard out stream.

```
<subsystem xmlns="urn:jboss:domain:logging:1.2">
   <console-handler name="CONSOLE" autoflush="true">
      <level name="INFO"/>
      <encoding value="UTF-8"/>
      <target value="System.out"/>
```

5. **Define the *filter-spec* Property**
   The ***filter-spec*** property is an expression value that defines a filter. The example provided defines a filter that does not match a pattern: **not(match("JBAS.*"))**.

```
<subsystem xmlns="urn:jboss:domain:logging:1.2">
   <console-handler name="CONSOLE" autoflush="true">
      <level name="INFO"/>
      <encoding value="UTF-8"/>
      <target value="System.out"/>
      <filter-spec value="not(match(&quot;JBAS.*&quot;))"/>
```

6. **Specify the *formatter***
   Use ***formatter*** to list the log formatter used by the log handler.

```
<subsystem xmlns="urn:jboss:domain:logging:1.2">
   <console-handler name="CONSOLE" autoflush="true">
      <level name="INFO"/>
      <encoding value="UTF-8"/>
      <target value="System.out"/>
      <filter-spec value="not(match(&quot;JBAS.*&quot;))"/>
      <formatter>
         <pattern-formatter pattern="%K{level}%d{HH:mm:ss,SSS} %-5p
[%c] (%t) %s%E%n"/>
      </formatter>
   </console-handler>
</subsystem>
```

Report a bug

### 5.5.4. Sample XML Configuration for a File Log Handler

The following procedure demonstrates a sample configuration for a file log handler.

**Procedure 5.4. Configure the File Log Handler**

1. **Add the File Log Handler Identifier Information**
   The ***name*** property sets the unique identifier for this log handler.

   When ***autoflush*** is set to **"true"** the log messages will be sent to the handler's target immediately upon request.

```
<file-handler name="accounts-rec-trail" autoflush="true">
```

2. **Set the *level* Property**

   The ***level*** property sets the maximum level of log message that the root logger records.

   ```
   <file-handler name="accounts-rec-trail" autoflush="true">
       <level name="INFO"/>
   ```

3. **Set the *encoding* Output**

   Use ***encoding*** to set the character encoding scheme to be used for the output.

   ```
   <file-handler name="accounts-rec-trail" autoflush="true">
       <level name="INFO"/>
       <encoding value="UTF-8"/>
   ```

4. **Set the *file* Object**

   The ***file*** object represents the file where the output of this log handler is written to. It has two configuration properties: ***relative-to*** and ***path***.

   The ***relative-to*** property 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.

   The ***path*** property 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.

   ```
   <file-handler name="accounts-rec-trail" autoflush="true">
       <level name="INFO"/>
       <encoding value="UTF-8"/>
       <file relative-to="jboss.server.log.dir" path="accounts-rec-
   trail.log"/>
   ```

5. **Specify the *formatter***

   Use ***formatter*** to list the log formatter used by the log handler.

   ```
   <file-handler name="accounts-rec-trail" autoflush="true">
       <level name="INFO"/>
       <encoding value="UTF-8"/>
       <file relative-to="jboss.server.log.dir" path="accounts-rec-
   trail.log"/>
       <formatter>
           <pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t)
   %s%E%n"/>
       </formatter>
   ```

6. **Set the *append* Property**

   When the ***append*** property is set to **"true"**, all messages written by this handler will be appended to an existing file. 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.

   ```
   <file-handler name="accounts-rec-trail" autoflush="true">
       <level name="INFO"/>
       <encoding value="UTF-8"/>
   ```

```
    <file relative-to="jboss.server.log.dir" path="accounts-rec-
trail.log"/>
    <formatter>
        <pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t)
%s%E%n"/>
    </formatter>
    <append value="true"/>
</file-handler>
```

Report a bug

### 5.5.5. Sample XML Configuration for a Periodic Log Handler

The following procedure demonstrates a sample configuration for a periodic log handler.

**Procedure 5.5. Configure the Periodic Log Handler**

1. **Add the Periodic Log Handler Identifier Information**
   The ***name*** property sets the unique identifier for this log handler.

   When ***autoflush*** is set to **"true"** the log messages will be sent to the handler's target immediately upon request.

   ```
   <periodic-rotating-file-handler name="FILE" autoflush="true">
   ```

2. **Set the *level* Property**
   The ***level*** property sets the maximum level of log message that the root logger records.

   ```
   <periodic-rotating-file-handler name="FILE" autoflush="true">
      <level name="INFO"/>
   ```

3. **Set the *encoding* Output**
   Use ***encoding*** to set the character encoding scheme to be used for the output.

   ```
   <periodic-rotating-file-handler name="FILE" autoflush="true">
      <level name="INFO"/>
      <encoding value="UTF-8"/>
   ```

4. **Specify the *formatter***
   Use ***formatter*** to list the log formatter used by the log handler.

   ```
   <periodic-rotating-file-handler name="FILE" autoflush="true">
      <level name="INFO"/>
      <encoding value="UTF-8"/>
      <formatter>
         <pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t)
   %s%E%n"/>
      </formatter>
   ```

5. **Set the *file* Object**
   The ***file*** object represents the file where the output of this log handler is written to. It has two configuration properties: ***relative-to*** and ***path***.

The ***relative-to*** property 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.

The ***path*** property 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.

```
<periodic-rotating-file-handler name="FILE" autoflush="true">
   <level name="INFO"/>
   <encoding value="UTF-8"/>
   <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"/>
```

6. **Set the *suffix* Value**
   The ***suffix*** is appended to the 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, **yyyy-MM-dd** will result in daily log rotation. See http://docs.oracle.com/javase/6/docs/api/index.html?java/text/SimpleDateFormat.html

```
<periodic-rotating-file-handler name="FILE" autoflush="true">
   <level name="INFO"/>
   <encoding value="UTF-8"/>
   <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"/>
```

7. **Set the *append* Property**
   When the ***append*** property is set to **"true"**, all messages written by this handler will be appended to an existing file. 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.

```
<periodic-rotating-file-handler name="FILE" autoflush="true">
   <level name="INFO"/>
   <encoding value="UTF-8"/>
   <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

### 5.5.6. Sample XML Configuration for a Size Log Handler

The following procedure demonstrates a sample configuration for a size log handler.

**Procedure 5.6. Configure the Size Log Handler**

1. **Add the Size Log Handler Identifier Information**
   The *name* property sets the unique identifier for this log handler.

   When ***autoflush*** is set to **"true"** the log messages will be sent to the handler's target immediately upon request.

   ```
   <size-rotating-file-handler name="accounts_debug" autoflush="false">
   ```

2. **Set the *level* Property**
   The ***level*** property sets the maximum level of log message that the root logger records.

   ```
   <size-rotating-file-handler name="accounts_debug" autoflush="false">
      <level name="DEBUG"/>
   ```

3. **Set the *encoding* Output**
   Use ***encoding*** to set the character encoding scheme to be used for the output.

   ```
   <size-rotating-file-handler name="accounts_debug" autoflush="false">
      <level name="DEBUG"/>
      <encoding value="UTF-8"/>
   ```

4. **Set the *file* Object**
   The ***file*** object represents the file where the output of this log handler is written to. It has two configuration properties: ***relative-to*** and ***path***.

   The ***relative-to*** property 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.

   The ***path*** property 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.

   ```
   <size-rotating-file-handler name="accounts_debug" autoflush="false">
      <level name="DEBUG"/>
      <encoding value="UTF-8"/>
      <file relative-to="jboss.server.log.dir" path="accounts-
   debug.log"/>
   ```

5. **Specify the *rotate-size* Value**
   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. For example: **50m** for 50 megabytes.

   ```
   <size-rotating-file-handler name="accounts_debug" autoflush="false">
      <level name="DEBUG"/>
   ```

```
   <encoding value="UTF-8"/>
   <file relative-to="jboss.server.log.dir" path="accounts-
debug.log"/>
   <rotate-size value="500k"/>
```

6. **Set the *max-backup-index* Number**

   The maximum number of rotated logs that are kept. When this number is reached, the oldest log is reused.

```
<size-rotating-file-handler name="accounts_debug" autoflush="false">
   <level name="DEBUG"/>
   <encoding value="UTF-8"/>
   <file relative-to="jboss.server.log.dir" path="accounts-
debug.log"/>
   <rotate-size value="500k"/>
   <max-backup-index value="5"/>
```

7. **Specify the *formatter***

   Use ***formatter*** to list the log formatter used by the log handler.

```
<size-rotating-file-handler name="accounts_debug" autoflush="false">
   <level name="DEBUG"/>
   <encoding value="UTF-8"/>
   <file relative-to="jboss.server.log.dir" path="accounts-
debug.log"/>
   <rotate-size value="500k"/>
   <max-backup-index value="5"/>
   <formatter>
        <pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t)
%s%E%n"/>
    </formatter>
```

8. **Set the *append* Property**

   When the ***append*** property is set to **"true"**, all messages written by this handler will be appended to an existing file. 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.

```
<size-rotating-file-handler name="accounts_debug" autoflush="false">
   <level name="DEBUG"/>
   <encoding value="UTF-8"/>
   <file relative-to="jboss.server.log.dir" path="accounts-
debug.log"/>
   <rotate-size value="500k"/>
   <max-backup-index value="5"/>
   <formatter>
        <pattern-formatter pattern="%d{HH:mm:ss,SSS} %-5p [%c] (%t)
%s%E%n"/>
    </formatter>
   <append value="true"/>
</size-rotating-file-handler>
```

Report a bug

### 5.5.7. Sample XML Configuration for a Async Log Handler

The following procedure demonstrates a sample configuration for an async log handler

**Procedure 5.7. Configure the Async Log Handler**

1. **Add the Async Log Handler Identifier Information**
   The ***name*** property sets the unique identifier for this log handler.

   ```
   <async-handler name="Async_NFS_handlers">
   ```

2. **Set the *level* Property**
   The ***level*** property sets the maximum level of log message that the root logger records.

   ```
   <async-handler name="Async_NFS_handlers">
      <level name="INFO"/>
   ```

3. **Define the *queue-length***
   The ***queue-length*** defines the maximum number of log messages that will be held by this handler while waiting for sub-handlers to respond.

   ```
   <async-handler name="Async_NFS_handlers">
      <level name="INFO"/>
      <queue-length value="512"/>
   ```

4. **Set Overflow Response**
   The ***overflow-action*** defines 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.

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

5. **List *subhandlers***
   The ***subhandlers*** list is the list of log handlers to which this async handler passes its log messages.

   ```
   <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

# PART IV. SET UP CACHE MODES

# CHAPTER 6. CACHE MODES

Red Hat JBoss Data Grid provides two modes:

- Local mode is the only non-clustered cache mode offered in JBoss Data Grid. In local mode, JBoss Data Grid operates as a simple single-node in-memory data cache. Local mode is most effective when scalability and failover are not required and provides high performance in comparison with clustered modes.

- Clustered mode replicates 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. For details about configuring JGroups, see Section 26.2, “Configure JGroups (Library Mode)”

Report a bug

## 6.1. ABOUT CACHE CONTAINERS

Cache containers are used in Red Hat JBoss Data Grid's Remote Client-Server mode as a starting point for a cache. The **`cache-container`** element acts as a parent of one or more (local or clustered) caches. To add clustered caches to the container, transport must be defined.

The following procedure demonstrates a sample cache container configuration:

**Procedure 6.1. How to Configure the Cache Container**

1. **Specify the Cache Container**
   The **`cache-container`** element specifies information about the cache container using the following parameters:

   ```
   <subsystem xmlns="urn:infinispan:server:core:6.1"
       default-cache-container="default">
   ```

   a. **Set the Cache Container Name**
      The ***`name`*** parameter defines the name of the cache container.

      ```
      <subsystem xmlns="urn:infinispan:server:core:6.1"
          default-cache-container="default">
       <cache-container name="default" />
      ```

   b. **Specify the Default Cache**
      The ***`default-cache`*** parameter defines the name of the default cache used with the cache container.

      ```
      <subsystem xmlns="urn:infinispan:server:core:6.1"
          default-cache-container="default">
       <cache-container name="default"
          default-cache="default" />
      ```

   c. **Enable/Disable Statistics**
      The ***`statistics`*** attribute is optional and is `true` by default. Statistics are useful in monitoring JBoss Data Grid via JMX or JBoss Operations Network, however they

adversely affect performance. Disable this attribute by setting it to **`false`** if it is not required.

```
<subsystem xmlns="urn:infinispan:server:core:6.1"
    default-cache-container="default">
 <cache-container name="default"
    default-cache="default"
    statistics="true"/>
```

d. **Define the Listener Executor**
   The ***`listener-executor`*** defines the executor used for asynchronous cache listener notifications.

```
<subsystem xmlns="urn:infinispan:server:core:6.1"
    default-cache-container="default">
 <cache-container name="default"
    default-cache="default"
    statistics="true"
    listener-executor="infinispan-listener" />
```

e. **Set the Cache Container Start Mode**
   The ***`start`*** parameter indicates when the cache container starts, i.e. whether it will start lazily when requested or "eagerly" when the server starts up. Valid values for this parameter are **EAGER** and **LAZY**.

```
<subsystem xmlns="urn:infinispan:server:core:6.1"
    default-cache-container="default">
 <cache-container name="default"
    default-cache="default"
    statistics="true"
    listener-executor="infinispan-listener"
    start="EAGER">
```

2. **Per-cache Statistics**
   If ***`statistics`*** are enabled at the container level, per-cache statistics can be selectively disabled for caches that do not require monitoring by setting the ***`statistics`*** attribute to **`false`**.

```
<subsystem xmlns="urn:infinispan:server:core:6.1"
    default-cache-container="default">
 <cache-container name="default"
    default-cache="default"
    statistics="true"
    listener-executor="infinispan-listener"
    start="EAGER">
  <local-cache name="default"
     statistics="true">
   ...
  </local-cache>
 </cache-container>
</subsystem>
```

Report a bug

## 6.2. LOCAL MODE

Using Red Hat 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-based 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

### 6.2.1. Configure Local Mode (Remote Client-Server Mode)

A local cache can be added to any cache container. The following example demonstrates how to add the **`local-cache`** element.

**Procedure 6.2. The `local-cache` Element**

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

1. **Add the Local Cache Name**
   The ***name*** parameter specifies the name of the local cache to use.

```
<cache-container name="local"
                 default-cache="default"
                 statistics="true">
        <local-cache name="default" >
```

2. **Set the Cache Container Start Mode**
   The ***start*** parameter indicates when the cache container starts, i.e. whether it will start lazily when requested or "eagerly" when the server starts up. Valid values for this parameter are **EAGER** and **LAZY**.

```
<cache-container name="local"
                 default-cache="default"
                 statistics="true">
        <local-cache name="default"
                     start="EAGER" >
```

3. **Configure Batching**
   The ***batching*** parameter specifies whether batching is enabled for the local cache.

```
<cache-container name="local"
```

```
                    default-cache="default"
                    statistics="true">
          <local-cache name="default"
                           start="EAGER"
                           batching="false" >
```

4. **Per-cache Statistics**
   If ***statistics*** are enabled at the container level, per-cache statistics can be selectively disabled for caches that do not require monitoring by setting the ***statistics*** attribute to **false**.

```
<cache-container name="local"
                    default-cache="default"
                    statistics="true">
          <local-cache name="default"
                           start="EAGER"
                           batching="false"
                           statistics="true">
```

5. **Specify Indexing Type**
   The ***indexing*** parameter specifies the type of indexing used for the local cache. Valid values for this parameter are **NONE**, **LOCAL** and **ALL**.

```
<cache-container name="local"
                    default-cache="default"
                    statistics="true">
          <local-cache name="default"
                           start="EAGER"
                           batching="false"
                           statistics="true">
                    <indexing index="NONE">
                   <property
name="default.directory_provider">ram</property>
                    </indexing>
          </local-cache>
```

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

### 6.2.2. Configure Local Mode (Library Mode)

In Red Hat JBoss Data Grid's Library mode, setting a cache's ***mode*** parameter to **local** equals not specifying a clustering mode at all. In the case of the latter, the cache defaults to local mode, even if its cache manager defines a transport.

Set the cluster mode to local as follows:

```
<clustering mode="local" />
```

Report a bug

## 6.3. CLUSTERED MODES

Red Hat 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 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

### 6.3.1. 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

### 6.3.2. Cache Mode Troubleshooting

#### 6.3.2.1. 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

#### 6.3.2.2. About Asynchronous Communications

In Red Hat 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.

**Example 6.1. Asynchronous Communications Example Configuration**

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

#### 6.3.2.3. 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

## 6.4. STATE TRANSFER

State transfer is a basic data grid or clustered cache functionality. Without state transfer, data would be lost as nodes are added to or removed from the cluster.

State transfer adjusts the cache’s internal state in response to a change in a cache membership. The change can be when a node joins or leaves, when two or more cluster partitions merge, or a combination of joins, leaves, and merges. State transfer occurs automatically in Red Hat JBoss Data Grid whenever a node joins or leaves the cluster.

In Red Hat JBoss Data Grid's replication mode, a new node joining the cache receives the entire cache state from the existing nodes. In distribution mode, the new node receives only a part of the state from the existing nodes, and the existing nodes remove some of their state in order to keep **`numOwners`** copies of each key in the cache (as determined through consistent hashing). In invalidation mode the initial state transfer is similar to replication mode, the only difference being that the nodes are not guaranteed to have the same state. When a node leaves, a replicated mode or invalidation mode cache does not perform any state transfer. A distributed cache needs to make additional copies of the keys that were stored on the leaving nodes, again to keep **`numOwners`** copies of each key.

State transfer, transfers both in-memory and persistent state by default, but both can be disabled in the configuration.

Report a bug

### 6.4.1. Non-Blocking State Transfer

Non-Blocking State Transfer in Red Hat JBoss Data Grid minimizes the time in which a cluster or node is unable to respond due to a state transfer in progress. Non-blocking state transfer is a core architectural improvement with the following goals:

- Minimize the interval(s) where the entire cluster cannot respond to requests because of a state transfer in progress.
- Minimize the interval(s) where an existing member stops responding to requests because of a state transfer in progress.
- Allow state transfer to occur with a drop in the performance of the cluster. However, the drop in the performance during the state transfer does not throw any exception, and allows processes to continue.

For simplicity, the total order-based commit protocol uses a blocking version of the currently implemented state transfer mechanism. The main differences between the regular state transfer and the total order state transfer are:

- The blocking protocol queues the transaction delivery during the state transfer.
- State transfer control messages (such as CacheTopologyControlCommand) are sent according to the total order information.

The total order-based commit protocol works with the assumption that all the transactions are delivered in the same order and they see the same data set. So, no transactions are validated during the state transfer because all the nodes must have the most recent key or values in memory.

Using the state transfer and blocking protocol in this manner allows the state transfer and transaction delivery on all on the nodes to be synchronized. However, transactions that are already involved in a state transfer (sent before the state transfer began and delivered after it concludes) must be resent. When resent, these transactions are treated as new joiners and assigned a new total order value.

Report a bug

### 6.4.2. Suppress State Transfer via JMX

State transfer can be suppressed using JMX in order to bring down and relaunch a cluster for maintenance. This operation permits a more efficient cluster shutdown and startup, and removes the risk of Out Of Memory errors when bringing down a grid.

When a new node joins the cluster and rebalancing is suspended, the **`getCache()`** call will timeout after **`stateTransfer.timeout`** expires unless rebalancing is re-enabled or **`stateTransfer.awaitInitialTransfer`**is set to **`false`**.

Disabling state transfer and rebalancing can be used for partial cluster shutdown or restart, however there is the possibility that data may be lost in a partial cluster shutdown due to state transfer being disabled.

Report a bug

### 6.4.3. The rebalancingEnabled Attribute

Suppressing rebalancing can only be triggered via the **`rebalancingEnabled`** JMX attribute, and requires no specific configuration.

The **`rebalancingEnabled`** attribute can be modified for the entire cluster from the **`LocalTopologyManager`** JMX Mbean on any node. This attribute is **`true`** by default, and is configurable programmatically.

Servers such as Hot Rod attempt to start all caches declared in the configuration during startup. If rebalancing is disabled, the cache will fail to start. Therefore, it is mandatory to use the following setting in a server environment:

```
<await-initial-transfer="false"/>
```

Report a bug

# CHAPTER 7. SET UP DISTRIBUTION MODE

## 7.1. ABOUT DISTRIBUTION MODE

When enabled, Red Hat 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

## 7.2. DISTRIBUTION MODE'S CONSISTENT HASH ALGORITHM

The hashing algorithm in Red Hat JBoss Data Grid is based on consistent hashing. The term consistent hashing is still used for this implementation, despite some divergence from a traditional consistent hash.

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. Unlike generic consistent hashing, the implementation used in JBoss Data Grid splits the key space into fixed segments. The number of segments is configurable using ***numSegments*** and cannot be changed without restarting the cluster. The mapping of keys to segments is also fixed — a key maps to the same segment, regardless of how the topology of the cluster changes.

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.

Each hash segment is mapped to a list of nodes called owners. The order is important because the first owner (also known as the primary owner) has a special role in many cache operations (for example, locking). The other owners are called backup owners. There is no rule about mapping segments to owners, although the hashing algorithms simultaneously balance the number of segments allocated to each node and minimize the number of segments that have to move after a node joins or leaves the cluster.

Report a bug

## 7.3. LOCATING ENTRIES IN DISTRIBUTION MODE

The consistent hash algorithm used in Red Hat 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

## 7.4. RETURN VALUES IN DISTRIBUTION MODE

In Red Hat 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

## 7.5. CONFIGURE DISTRIBUTION MODE (REMOTE CLIENT-SERVER MODE)

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

**Procedure 7.1. The `distributed-cache` Element**

The **`distributed-cache`** element configures settings for the distributed cache using the following parameters:

1. **Add the Cache Name**
   The **name** parameter provides a unique identifier for the cache.

```
<cache-container name="clustered"
   default-cache="default"
   statistics="true">
 <transport executor="infinispan-transport" lock-timeout="60000"/>
  <distributed-cache name="default" />
```

2. **Set the Clustered Cache Start Mode**
   The **mode** parameter sets the clustered cache mode. Valid values are **SYNC** (synchronous) and **ASYNC** (asynchronous).

```
<cache-container name="clustered"
   default-cache="default"
   statistics="true">
 <transport executor="infinispan-transport" lock-timeout="60000"/>
 <distributed-cache name="default"
      mode="SYNC" />
```

3. **Specify Number of Segments**
   The (optional) **`segments`** parameter specifies the number of hash space segments per cluster. The recommended value for this parameter is ten multiplied by the cluster size and the default value is **`20`**.

```
<cache-container name="clustered"
   default-cache="default"
   statistics="true">
 <transport executor="infinispan-transport" lock-timeout="60000"/>
 <distributed-cache name="default"
      mode="SYNC"
      segments="20" />
```

■

4. **Set the Cache Start Mode**
   The `start` parameter specifies whether the cache starts when the server starts up or when it is requested or deployed.

```
<cache-container name="clustered"
   default-cache="default"
   statistics="true">
 <transport executor="infinispan-transport" lock-timeout="60000"/>
 <distributed-cache name="default"
      mode="SYNC"
      segments="20"
      start="EAGER"/>
```

5. **Per-cache Statistics**
   If ***`statistics`*** are enabled at the container level, per-cache statistics can be selectively disabled for caches that do not require monitoring by setting the ***`statistics`*** attribute to **`false`**.

```
<cache-container name="clustered"
   default-cache="default"
   statistics="true">
 <transport executor="infinispan-transport" lock-timeout="60000"/>
 <distributed-cache name="default"
      mode="SYNC"
      segments="20"
      start="EAGER"
      statistics="true">
   ...
 </distributed-cache>
</cache-container>
```

**IMPORTANT**

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

For details about the **`cache-container`**, **`locking`**, and **`transaction`** elements, see the appropriate chapter.

Report a bug

## 7.6. CONFIGURE DISTRIBUTION MODE (LIBRARY MODE)

The following procedure shows a distributed cache configuration in Red Hat JBoss Data Grid's Library mode.

**Procedure 7.2. Distributed Cache Configuration**

1. **Set the Clustered Mode**
   The **`clustering`** element's ***mode*** parameter's value determines the clustering mode selected for the cache.

```
<clustering mode="distribution">
```

2. **Specify the Remote Call Timeout**
   The **sync** element's ***replTimeout*** parameter specifies the maximum time period in milliseconds for an acknowledgment after a remote call. If the time period ends without any acknowledgment, an exception is thrown.

```
<clustering mode="distribution">
        <sync replTimeout="${TIME}" />
```

3. **Define State Transfer Settings**
   The **stateTransfer** element specifies how state is transferred when a node leaves or joins the cluster. It uses the following parameters:

   a. **Specify State Transfer Batch Size**
      The ***chunkSize*** parameter is the number of cache entries to be transferred in one chunk. The default ***chunkSize*** value is 512. The ***chunkSize*** depends on many parameters (for example, size of entries) and the best setting needs to be measured. To change the value for larger cache entries, smaller chunks are advisable and for smaller cache entries, increasing the ***chunkSize*** offers better performance.

```
<clustering mode="distribution">
        <sync replTimeout="${TIME}" />
        <stateTransfer chunkSize="${SIZE}" />
```

   b. **Set *fetchInMemoryState* Parameter**
      The ***fetchInMemoryState*** parameter when set to **true**, requests state information from neighboring caches on start up. This impacts the start up time for the cache.

```
<clustering mode="distribution">
        <sync replTimeout="${TIME}" />
        <stateTransfer chunkSize="${SIZE}"
                       fetchInMemoryState="{true/false}" />
```

   c. **Define the *awaitInitialTransfer* Parameter**
      The ***awaitInitialTransfer*** parameter causes the first call to method **CacheManager.getCache()** on the joiner node to block and wait until the joining is complete and the cache has finished receiving state from neighboring caches (if ***fetchInMemoryState*** is enabled). This option applies to distributed and replicated caches only and is enabled by default.

```
<clustering mode="distribution">
        <sync replTimeout="${TIME}" />
        <stateTransfer chunkSize="${SIZE}"
                       fetchInMemoryState="{true/false}"
                       awaitInitialTransfer="{true/false}" />
```

   d. **Set *timeout* Value**
      The ***timeout*** parameter specifies the maximum time (in milliseconds) the cache waits for responses from neighboring caches with the requested states. If no response is received within the the ***timeout*** period, the start up process aborts and an exception is thrown.

```
<clustering mode="distribution">
        <sync replTimeout="${TIME}" />
        <stateTransfer chunkSize="${SIZE}"
                       fetchInMemoryState="{true/false}"
                       awaitInitialTransfer="{true/false}"
                       timeout="${TIME}" />
```

4. **Specify Transport Configuration**
   The `transport` element defines the transport configuration for the cache container as follows:

   a. **Specify the Cluster Name**
      The ***`clusterName`*** parameter specifies the name of the cluster. Nodes can only connect to clusters that share the same name.

```
<global>
    <transport clusterName="${NAME}" />
</global>
```

   b. **Set the *`distributedSyncTimeout`* Value**
      The ***`distributedSyncTimeout`*** parameter specifies the time to wait to acquire a lock on the distributed lock. This distributed lock ensures that a single cache can transfer state or rehash state at a time.

```
<global>
    <transport clusterName="${NAME}"
        distributedSyncTimeout="${TIME}" />
</global>
```

   c. **Set the Network Transport**
      The ***`transportClass`*** parameter specifies a class that represents a network transport for the cache container.

```
<global>
    <transport clusterName="${NAME}"
        distributedSyncTimeout="${TIME}"
        transportClass="${CLASS}" />
</global>
```

Report a bug

## 7.7. 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 7.1. 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

## 7.8. GET AND PUT USAGE 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 Red Hat JBoss Data Grid must wait for the return value.

Report a bug

### 7.8.1. 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.

Use 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 8. SET UP REPLICATION MODE

## 8.1. ABOUT REPLICATION MODE

Red Hat 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.

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

Report a bug

## 8.2. 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.

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

Report a bug

## 8.3. CONFIGURE REPLICATION MODE (REMOTE CLIENT-SERVER MODE)

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

**Procedure 8.1. The *replicated-cache* Element**

The **replicated-cache** element configures settings for the distributed cache using the following parameters:

1. **Add the Cache Name**
   The **name** parameter provides a unique identifier for the cache.

```
<cache-container name="clustered"
   default-cache="default"
   statistics="true">
 <replicated-cache name="default">
```

2. **Set the Clustered Cache Start Mode**
   The **mode** parameter sets the clustered cache mode. Valid values are **SYNC** (synchronous) and **ASYNC** (asynchronous).

```
<cache-container name="clustered"
   default-cache="default"
   statistics="true">
```

```
    <replicated-cache name="default"
        mode="SYNC">
```

3. **Set the Cache Start Mode**
   The **start** parameter specifies whether the cache starts when the server starts up or when it is requested or deployed.

```
  <cache-container name="clustered"
     default-cache="default"
     statistics="true">
   <replicated-cache name="default"
      mode="SYNC"
      start="EAGER">
```

4. **Per-cache Statistics**
   If ***statistics*** are enabled at the container level, per-cache statistics can be selectively disabled for caches that do not require monitoring by setting the ***statistics*** attribute to **false**.

```
  <cache-container name="clustered"
     default-cache="default"
     statistics="true">
   <replicated-cache name="default"
      mode="SYNC"
      start="EAGER"
      statistics="true">
          ...
   </replicated-cache>
  </cache-container>
```

5. **Set Up Transactions**
   The **transaction** element sets up the transaction mode for the replicated cache.

**IMPORTANT**

In Remote Client-Server mode, the transactions element is set to **NONE** unless JBoss Data Grid is used in a compatibility mode where the cluster contains both JBoss Data Grid server and library instances. In this case, if transactions are configured in the library mode instance, they must also be configured in the server instance.

```
  <cache-container name="clustered"
     default-cache="default"
     statistics="true">
   <replicated-cache name="default"
      mode="SYNC"
      start="EAGER"
      statistics="true">
    <transaction mode="NONE" />
   </replicated-cache>
  </cache-container>
```

**IMPORTANT**

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

For details about the `cache-container` and `locking`, see the appropriate chapter.

Report a bug

## 8.4. CONFIGURE REPLICATION MODE (LIBRARY MODE)

The following procedure shows a replication mode configuration in Red Hat JBoss Data Grid's Library mode.

**Procedure 8.2. Replication Mode Configuration**

1. **Set the Clustered Mode**
   The `clustering` element's ***mode*** parameter's value determines the clustering mode selected for the cache.

   ```
   <clustering mode="replication">
   ```

2. **Specify the Remote Call Timeout**
   The sync element's ***replTimeout*** parameter specifies the maximum time period in milliseconds for an acknowledgment after a remote call. If the time period ends without any acknowledgment, an exception is thrown.

   ```
   <clustering mode="replication">
           <sync replTimeout="${TIME}" />
   ```

3. **Define State Transfer Settings**
   The `stateTransfer` element specifies how state is transferred when a node leaves or joins the cluster. It uses the following parameters:

   a. **Specify State Transfer Batch Size**
      The ***chunkSize*** parameter is the number of cache entries to be transferred in one chunk. The default ***chunkSize*** value is 512. The ***chunkSize*** depends on many parameters (for example, size of entries) and the best setting needs to be measured. To change the value for larger cache entries, smaller chunks are advisable and for smaller cache entries, increasing the ***chunkSize*** offers better performance.

      ```
      <clustering mode="replication">
              <sync replTimeout="${TIME}" />
              <stateTransfer chunkSize="${SIZE}" />
      ```

   b. **Set *fetchInMemoryState* Parameter**
      The ***fetchInMemoryState*** parameter when set to `true`, requests state information from neighboring caches on start up. This impacts the start up time for the cache.

      ```
      <clustering mode="replication">
              <sync replTimeout="${TIME}" />
              <stateTransfer chunkSize="${SIZE}"
      ```

```
                        fetchInMemoryState="{true/false}" />
```

c. **Define the *awaitInitialTransfer* Parameter**
   The ***awaitInitialTransfer*** parameter causes the first call to method **`CacheManager.getCache()`** on the joiner node to block and wait until the joining is complete and the cache has finished receiving state from neighboring caches (if ***fetchInMemoryState*** is enabled). This option applies to distributed and replicated caches only and is enabled by default.

```
<clustering mode="replication">
        <sync replTimeout="${TIME}" />
        <stateTransfer chunkSize="${SIZE}"
                       fetchInMemoryState="{true/false}"
                       awaitInitialTransfer="{true/false}" />
```

d. **Set the *timeout* Value**
   The ***timeout*** parameter specifies the maximum time (in milliseconds) the cache waits for responses from neighboring caches with the requested states. If no response is received within the the ***timeout*** period, the start up process aborts and an exception is thrown.

```
<clustering mode="replication">
        <sync replTimeout="${TIME}" />
        <stateTransfer chunkSize="${SIZE}"
                       fetchInMemoryState="{true/false}"
                       awaitInitialTransfer="{true/false}"
                       timeout="${TIME}" />
```

4. **Specify Transport Configuration**
   The **`transport`** element defines the transport configuration for the cache container as follows:

a. **Specify the Cluster Name**
   The ***clusterName*** parameter specifies the name of the cluster. Nodes can only connect to clusters that share the same name.

```
<global>
    <transport clusterName="${NAME}" />
</global>
```

b. **Set the *distributedSyncTimeout* Value**
   The ***distributedSyncTimeout*** parameter specifies the time to wait to acquire a lock on the distributed lock. This distributed lock ensures that a single cache can transfer state or rehash state at a time.

```
<global>
    <transport clusterName="${NAME}"
        distributedSyncTimeout="${TIME}" />
</global>
```

c. **Set the Network Transport**
   The ***transportClass*** parameter specifies a class that represents a network transport for the cache container.

```
<global>
    <transport clusterName="${NAME}"
        distributedSyncTimeout="${TIME}"
        transportClass="${CLASS}" />
</global>
```

Report a bug

## 8.5. 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

### 8.5.1. 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 a replication queue is enabled (which is recommended for asynchronous mode).

Report a bug

## 8.6. THE REPLICATION QUEUE

In replication mode, Red Hat 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 disadvantage of using the replication queue 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. When the replication queue is disabled, the data is directly transmitted and therefore the data arrives at the cluster nodes faster.

A replication queue is used in conjunction with asynchronous mode.

Report a bug

### 8.6.1. Replication Queue Usage

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

- Disable asynchronous marshalling.
- 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:

**Example 8.1. Replication Queue in Asynchronous Mode**

```
<replicated-cache name="asyncCache"
                 start="EAGER"
                 mode="ASYNC"
                 batching="false"
                 indexing="NONE"
                 statistics="true"
                 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

## 8.7. ABOUT REPLICATION GUARANTEES

In a clustered cache, the user can receive synchronous replication guarantees as well as the parallelism associated with asynchronous replication. Red Hat 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

## 8.8. 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 Red Hat 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 9. SET UP INVALIDATION MODE

## 9.1. 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.

Red Hat JBoss Data Grid, in such a situation, is used as an optimization for a system that performs many read operations 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

## 9.2. CONFIGURE INVALIDATION MODE (REMOTE CLIENT-SERVER MODE)

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

**Procedure 9.1. The `invalidation-cache` Element**

The **`invalidation-cache`** element configures settings for the distributed cache using the following parameters:

1. **Add the Cache Name**
   The **name** parameter provides a unique identifier for the cache.

```
<cache-container name="local"
       default-cache="default"
       statistics="true">
 <invalidation-cache name="default">
```

2. **Set the Clustered Cache Start Mode**
   The **mode** parameter sets the clustered cache mode. Valid values are **SYNC** (synchronous) and **ASYNC** (asynchronous).

```
<cache-container name="local"
       default-cache="default"
       statistics="true">
 <invalidation-cache name="default"
      mode="ASYNC">
```

3. **Set the Cache Start Mode**
   The **`start`** parameter specifies whether the cache starts when the server starts up or when it is requested or deployed.

```
<cache-container name="local"
       default-cache="default"
       statistics="true">
```

```
 <invalidation-cache name="default"
       mode="ASYNC"
       start="EAGER">
```

4. **Per-cache Statistics**
   If ***statistics*** are enabled at the container level, per-cache statistics can be selectively disabled for caches that do not require monitoring by setting the ***statistics*** attribute to **false**.

```
<cache-container name="local"
        default-cache="default"
        statistics="true">
 <invalidation-cache name="default"
       mode="ASYNC"
       start="EAGER"
       statistics="true">
   ...
 </invalidation-cache>
</cache-container>
```

**IMPORTANT**

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

For details about the **cache-container**, **locking**, and **transaction** elements, see the appropriate chapter.

Report a bug

## 9.3. CONFIGURE INVALIDATION MODE (LIBRARY MODE)

The following procedure shows an invalidation mode cache configuration in Red Hat JBoss Data Grid's Library mode.

**Procedure 9.2. Invalidation Mode Configuration**

1. **Set the Clustered Mode**
   The **clustering** element's ***mode*** parameter's value determines the clustering mode selected for the cache.

```
<clustering mode="invalidation">
```

2. **Specify the Remote Call Timeout**
   The **sync** element's ***replTimeout*** parameter specifies the maximum time period in milliseconds for an acknowledgment after a remote call. If the time period ends without any acknowledgment, an exception is thrown.

```
<clustering mode="invalidation">
        <sync replTimeout="${TIME}" />
```

3. **Define State Transfer Settings**

The `stateTransfer` element specifies how state is transferred when a node leaves or joins the cluster. It uses the following parameters:

a. **Specify State Transfer Batch Size**
   The ***chunkSize*** parameter specifies the size of cache entry state batches to be transferred. If this value is greater than `0`, the value set is the size of chunks sent. If the value is less than `0`, all states are transferred at the same time.

```
<clustering mode="invalidation">
        <sync replTimeout="${TIME}" />
        <stateTransfer chunkSize="${SIZE}" />
```

b. **Set *fetchInMemoryState* Parameter**
   The ***fetchInMemoryState*** parameter when set to `true`, requests state information from neighboring caches on start up. This impacts the start up time for the cache.

```
<clustering mode="invalidation">
        <sync replTimeout="${TIME}" />
        <stateTransfer chunkSize="${SIZE}"
                       fetchInMemoryState="{true/false}" />
```

c. **Define the *awaitInitialTransfer* Parameter**
   The ***awaitInitialTransfer*** parameter causes the first call to method **`CacheManager.getCache()`** on the joiner node to block and wait until the joining is complete and the cache has finished receiving state from neighboring caches (if ***fetchInMemoryState*** is enabled). This option applies to distributed and replicated caches only and is enabled by default.

```
<clustering mode="invalidation">
        <sync replTimeout="${TIME}" />
        <stateTransfer chunkSize="${SIZE}"
                       fetchInMemoryState="{true/false}"
                       awaitInitialTransfer="{true/false}" />
```

d. **Set *timeout* Value**
   The ***timeout*** parameter specifies the maximum time (in milliseconds) the cache waits for responses from neighboring caches with the requested states. If no response is received within the the ***timeout*** period, the start up process aborts and an exception is thrown.

```
<clustering mode="invalidation">
        <sync replTimeout="${TIME}" />
        <stateTransfer chunkSize="${SIZE}"
                       fetchInMemoryState="{true/false}"
                       awaitInitialTransfer="{true/false}"
                       timeout="${TIME}" />
```

4. **Specify Transport Configuration**
   The **`transport`** element defines the transport configuration for the cache container as follows:

   a. **Specify the Cluster Name**
      The ***clusterName*** parameter specifies the name of the cluster. Nodes can only connect to clusters that share the same name.

```
<global>
    <transport clusterName="${NAME}" />
</global>
```

b. **Set the *`distributedSyncTimeout`* Value**

The *`distributedSyncTimeout`* parameter specifies the time to wait to acquire a lock on the distributed lock. This distributed lock ensures that a single cache can transfer state or rehash state at a time.

```
<global>
    <transport clusterName="${NAME}"
        distributedSyncTimeout="${TIME}" />
</global>
```

c. **Set the Network Transport**

The *`transportClass`* parameter specifies a class that represents a network transport for the cache container.

```
<global>
    <transport clusterName="${NAME}"
        distributedSyncTimeout="${TIME}"
        transportClass="${CLASS}" />
</global>
```

Report a bug

## 9.4. SYNCHRONOUS/ASYNCHRONOUS INVALIDATION

In Red Hat 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

## 9.5. 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

# PART V. REMOTE CLIENT-SERVER MODE INTERFACES

Red Hat JBoss Data Grid offers the following APIs to interact with the data grid in Remote Client-Server mode:

- 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

# CHAPTER 10. THE ASYNCHRONOUS API

In addition to synchronous API methods, Red Hat 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

## 10.1. ASYNCHRONOUS API BENEFITS

The asynchronous API does not block, which provides multiple benefits, such as:

- The guarantee of synchronous communication, with the added ability to handle failures and exceptions.
- Not being required to block a thread's operations until the call completes.

These benefits allow you to better harness the parallelism in your system, for example:

**Example 10.1. Using the Asynchronous API**

```
Set<Future<?>> futures = new HashSet<Future<?>>();
futures.add(cache.putAsync("key1", "value1"));
futures.add(cache.putAsync("key2", "value2"));
futures.add(cache.putAsync("key3", "value3"));
```

In the example, The following lines do not block the thread as they execute:

- **futures.add(cache.putAsync(*key1*, *value1*));**
- **futures.add(cache.putAsync(*key2*, *value2*));**
- **futures.add(cache.putAsync(*key3*, *value3*));**

The remote calls from the three put operations are executed in parallel. This is particularly useful when executed in distributed mode.

Report a bug

## 10.2. ABOUT ASYNCHRONOUS PROCESSES

For a typical write operation in Red Hat JBoss Data Grid, the following processes fall on the critical path, ordered from most resource-intensive to the least:

- Network calls

- Marshalling
- Writing to a cache store (optional)
- Locking

In JBoss Data Grid, using asynchronous methods removes network calls and marshalling from the critical path.

Report a bug

## 10.3. RETURN VALUES AND THE ASYNCHRONOUS API

When the asynchronous API is used in Red Hat JBoss Data Grid, the client code requires the asynchronous operation to return either the **`Future`** or the **`NotifyingFuture`** in order to query the previous value.

**NOTE**

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

Call the following operation to obtain the result of an asynchronous operation. This operation blocks threads when called.

```
Future.get()
```

Report a bug

# CHAPTER 11. THE REST INTERFACE

Red Hat JBoss Data Grid provides a REST interface. The primary benefit of the REST API is that it allows for loose coupling between the client and server. The need for specific versions of client libraries and bindings is also eliminated. The REST API introduces an overhead, and requires a REST client or custom code to understand and create REST calls.

To interact with JBoss Data Grid's REST API only requires a HTTP client library. For Java, the Apache HTTP Commons Client is recommended. Alternatively, the java.net API can be used.

Report a bug

## 11.1. RUBY CLIENT CODE

The following code is an example of interacting with Red Hat JBoss Data Grid REST API using ruby. The provided code does not require any special libraries and standard net/HTTP libraries are sufficient.

**Example 11.1. Using the REST API with Ruby**

```
require 'net/http'

http = Net::HTTP.new('localhost', 8080)

#An example of how to create a new entry

http.post('/rest/MyData/MyKey', 'DATA_HERE', {"Content-Type" =>
"text/plain"})

#An example of using a GET operation to retrieve the key

puts http.get('/rest/MyData/MyKey').body

#An Example of using a PUT operation to overwrite the key

http.put('/rest/MyData/MyKey', 'MORE DATA', {"Content-Type" =>
"text/plain"})

#An example of Removing the remote copy of the key

http.delete('/rest/MyData/MyKey')

#An example of creating binary data

http.put('/rest/MyImages/Image.png',
File.read('/Users/michaelneale/logo.png'), {"Content-Type" =>
"image/png"})
```

Report a bug

## 11.2. USING JSON WITH RUBY EXAMPLE

**Prerequisites**

To use JavaScript Object Notation (JSON) with ruby to interact with Red Hat JBoss Data Grid's REST Interface, install the JSON Ruby library (see your platform's package manager or the Ruby documentation) and declare the requirement using the following code:

```
require 'json'
```

**Using JSON with Ruby**

The following code is an example of how to use JavaScript Object Notation (JSON) in conjunction with Ruby to send specific data, in this case the name and age of an individual, using the **PUT** function.

```
data = {:name => "michael", :age => 42 }
http.put('/infinispan/rest/Users/data/0', data.to_json, {"Content-Type" =>
"application/json"})
```

Report a bug

## 11.3. PYTHON CLIENT CODE

The following code is an example of interacting with the Red Hat JBoss Data Grid REST API using Python. The provided code requires only the standard HTTP library.

**Example 11.2. Using the REST API with Python**

```
import httplib

#How to insert data

conn = httplib.HTTPConnection("localhost:8080")
data = "SOME DATA HERE \!" #could be string, or a file...
conn.request("POST", "/rest/Bucket/0", data, {"Content-Type":
"text/plain"})
response = conn.getresponse()
print response.status

#How to retrieve data

import httplib
conn = httplib.HTTPConnection("localhost:8080")
conn.request("GET", "/rest/Bucket/0")
response = conn.getresponse()
print response.status
print response.read()
```

Report a bug

## 11.4. JAVA CLIENT CODE

The following code is an example of interacting with Red Hat JBoss Data Grid REST API using Java.

**Example 11.3. Defining Imports**

```
import java.io.BufferedReader;import java.io.IOException;
import java.io.InputStreamReader;import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;import java.net.URL;
```

**Example 11.4. Adding a String Value to a Cache**

```
public class RestExample {

   /**
    * Method that puts a String value in cache.
    * @param urlServerAddress
    * @param value
    * @throws IOException
    */

   public void putMethod(String urlServerAddress, String value) throws
IOException {
      System.out.println("----------------------------------------");
      System.out.println("Executing PUT");
      System.out.println("----------------------------------------");
      URL address = new URL(urlServerAddress);
      System.out.println("executing request " + urlServerAddress);
      HttpURLConnection connection = (HttpURLConnection)
address.openConnection();
      System.out.println("Executing put method of value: " + value);
      connection.setRequestMethod("PUT");
      connection.setRequestProperty("Content-Type", "text/plain");
      connection.setDoOutput(true);

      OutputStreamWriter outputStreamWriter = new
OutputStreamWriter(connection.getOutputStream());
      outputStreamWriter.write(value);

      connection.connect();
      outputStreamWriter.flush();

      System.out.println("----------------------------------------");
      System.out.println(connection.getResponseCode() + " " +
connection.getResponseMessage());
      System.out.println("----------------------------------------");

      connection.disconnect();
   }
```

The following code is an example of a method used that reads a value specified in a URL using Java to interact with the JBoss Data Grid REST Interface.

**Example 11.5. Get a String Value from a Cache**

```
   /**
    * Method that gets an value by a key in url as param value.
    * @param urlServerAddress
```

```
     * @return String value
     * @throws IOException
     */
    public String getMethod(String urlServerAddress) throws IOException
{
        String line = new String();
        StringBuilder stringBuilder = new StringBuilder();

        System.out.println("----------------------------------------");
        System.out.println("Executing GET");
        System.out.println("----------------------------------------");

        URL address = new URL(urlServerAddress);
        System.out.println("executing request " + urlServerAddress);

        HttpURLConnection connection = (HttpURLConnection)
address.openConnection();
        connection.setRequestMethod("GET");
        connection.setRequestProperty("Content-Type", "text/plain");
        connection.setDoOutput(true);

        BufferedReader bufferedReader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));

        connection.connect();

        while ((line = bufferedReader.readLine()) != null) {
           stringBuilder.append(line + '\n');
        }

        System.out.println("Executing get method of value: " +
stringBuilder.toString());

        System.out.println("----------------------------------------");
        System.out.println(connection.getResponseCode() + " " +
connection.getResponseMessage());
        System.out.println("----------------------------------------");

        connection.disconnect();

        return stringBuilder.toString();
    }
```

**Example 11.6. Using a Java Main Method**

```
/**
     * Main method example.
     * @param args
     * @throws IOException
     */
public static void main(String[] args) throws IOException {
//Note that the cache name is "cacheX"
RestExample restExample = new RestExample();
restExample.putMethod("http://localhost:8080/rest/cacheX/1", "Infinispan
REST Test");
restExample.getMethod("http://localhost:8080/rest/cacheX/1");
   }
}
}
```

Report a bug

## 11.5. THE REST INTERFACE CONNECTOR

Red Hat JBoss Data Grid supports three connector types, namely:

- The **`hotrod-connector`** element, which defines the configuration for a Hot Rod based connector.
- The **`memcached-connector`** element, which defines the configuration for a memcached based connector.
- The **`rest-connector`** element, which defines the configuration for a REST interface based connector.

The connector declaration enables a Hot Rod, Memcached, or REST server using a socket binding, which is declared within a *`<socket-binding-group />`*, and exposing the caches declared in the **`local`** container, using defaults for all other settings. The following examples show how to connect to Hot Rod, Memcached, and REST servers.

The REST connector differs from Hot Rod and Memcached because it requires a web subsystem. Therefore configurations such as socket-binding, worker threads, timeouts, etc, must be performed on the web subsystem. The following enables a REST server.

```
<rest-connector virtual-server="default-host" cache-container="local"
security-domain="other" auth-method="BASIC"/>
```

See Section 11.6, “Using the REST Interface” for more information.

Report a bug

### 11.5.1. Configure REST Connectors

Use the following procedure to configure the **`rest-connector`** element in Red Hat JBoss Data Grid's Remote Client-Server mode.

**Procedure 11.1. Configuring REST Connectors for Remote Client-Server Mode**

The **`rest-connector`** element specifies the configuration information for the REST connector.

1. **The *`virtual-server`* Parameter**
   The *`virtual-server`* parameter specifies the virtual server used by the REST connector. The default value for this parameter is **`default-host`**. This is an optional parameter.

   ```
   <subsystem xmlns="urn:infinispan:server:endpoint:6.1">
      <rest-connector virtual-server="default-host" />
   </subsystem>
   ```

▪

2. **The *cache-container* Parameter**
   The ***cache-container*** parameter names the cache container used by the REST connector. This is a mandatory parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
   <rest-connector virtual-server="default-host"
                   cache-container="local" />
</subsystem>
```

3. **The *context-path* Parameter**
   The ***context-path*** parameter specifies the context path for the REST connector. The default value for this parameter is an empty string (**""**). This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
   <rest-connector virtual-server="default-host"
                   cache-container="local"
                   context-path="${CONTEXT_PATH}" />
</subsystem>
```

4. **The *security-domain* Parameter**
   The ***security-domain*** parameter specifies that the specified domain, declared in the security subsystem, should be used to authenticate access to the REST endpoint. This is an optional parameter. If this parameter is omitted, no authentication is performed.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
   <rest-connector virtual-server="default-host"
                   cache-container="local"
                   context-path="${CONTEXT_PATH}"
                   security-domain="${SECURITY_DOMAIN}" />
</subsystem>
```

5. **The *auth-method* Parameter**
   The ***auth-method*** parameter specifies the method used to retrieve credentials for the end point. The default value for this parameter is **BASIC**. Supported alternate values include **BASIC**, **DIGEST**, and **CLIENT-CERT**. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
   <rest-connector virtual-server="default-host"
                   cache-container="local"
                   context-path="${CONTEXT_PATH}"
                   security-domain="${SECURITY_DOMAIN}"
                   auth-method="${METHOD}"  />
</subsystem>
```

6. **The *security-mode* Parameter**
   The ***security-mode*** parameter specifies whether authentication is required only for write operations (such as PUT, POST and DELETE) or for read operations (such as GET and HEAD) as well. Valid values for this parameter are **WRITE** for authenticating write operations only, or **READ_WRITE** to authenticate read and write operations. The default value for this parameter is **READ_WRITE**.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
   <rest-connector virtual-server="default-host"
                   cache-container="local"
                   context-path="${CONTEXT_PATH}"
                   security-domain="${SECURITY_DOMAIN}"
                   auth-method="${METHOD}"
                   security-mode="${MODE}" />
</subsystem>
```

Report a bug

## 11.6. USING THE REST INTERFACE

The REST Interface can be used in Red Hat JBoss Data Grid's Remote Client-Server mode to perform the following operations:

- Adding data
- Retrieving data
- Removing data

Report a bug

### 11.6.1. Adding Data Using REST

In Red Hat JBoss Data Grid's REST Interface, use the following methods to add data to the cache:

- HTTP **PUT** method
- HTTP **POST** method

When the **PUT** and **POST** methods are used, the body of the request contains this data, which includes any information added by the user.

Both the **PUT** and **POST** methods require a Content-Type header.

Report a bug

#### 11.6.1.1. About PUT /{cacheName}/{cacheKey}

A **PUT** request from the provided URL form places the payload, from the request body in the targeted cache using the provided key. The targeted cache must exist on the server for this task to successfully complete.

As an example, in the following URL, the value **hr** is the cache name and **payRoll%2F3** is the key. The value **%2F** indicates that a **/** was used in the key.

```
http://someserver/rest/hr/payRoll%2F3
```

Any existing data is replaced and ***Time-To-Live*** and ***Last-Modified*** values are updated, if an update is required.

**NOTE**

A cache key that contains the value **%2F** to represent a / in the key (as in the provided example) can be successfully run if the server is started using the following argument:

```
-Dorg.apache.tomcat.util.buf.UDecoder.ALLOW_ENCODED_SLASH=true
```

Report a bug

#### 11.6.1.2. About POST /{cacheName}/{cacheKey}

The **POST** method from the provided URL form places the payload (from the request body) in the targeted cache using the provided key. However, in a **POST** method, if a value in a cache/key exists, a **HTTP CONFLICT** status is returned and the content is not updated.

Report a bug

### 11.6.2. Retrieving Data Using REST

In Red Hat JBoss Data Grid's REST Interface, use the following methods to retrieve data from the cache:

- HTTP **GET** method.
- HTTP **HEAD** method.

Report a bug

#### 11.6.2.1. About GET /{cacheName}/{cacheKey}

The **GET** method returns the data located in the supplied ***cacheName***, matched to the relevant key, as the body of the response. The Content-Type header provides the type of the data. A browser can directly access the cache.

A unique entity tag (ETag) is returned for each entry along with a Last-Modified header which indicates the state of the data at the requested URL. ETags allow browsers (and other clients) to ask for data only in the case where it has changed (to save on bandwidth). ETag is a part of the HTTP standard and is supported by Red Hat JBoss Data Grid.

The type of content stored is the type returned. As an example, if a String was stored, a String is returned. An object which was stored in a serialized form must be manually deserialized.

Report a bug

#### 11.6.2.2. About HEAD /{cacheName}/{cacheKey}

The **HEAD** method operates in a manner similar to the **GET** method, however returns no content (header fields are returned).

Report a bug

### 11.6.3. Removing Data Using REST

To remove data from Red Hat JBoss Data Grid using the REST interface, use the HTTP **DELETE** method to retrieve data from the cache. The **DELETE** method can:

- Remove a cache entry/value. (**DELETE /{cacheName}/{cacheKey}**)
- Remove all entries from a cache. (**DELETE /{cacheName}**)

Report a bug

### 11.6.3.1. About DELETE /{cacheName}/{cacheKey}

Used in this context (**DELETE /{cacheName}/{cacheKey}**), the **DELETE** method removes the key/value from the cache for the provided key.

Report a bug

### 11.6.3.2. About DELETE /{cacheName}

In this context (**DELETE /{cacheName}**), the **DELETE** method removes all entries in the named cache. After a successful **DELETE** operation, the HTTP status code **200** is returned.

Report a bug

### 11.6.3.3. Background Delete Operations

Set the value of the ***performAsync*** header to **true** to ensure an immediate return while the removal operation continues in the background.

Report a bug

## 11.6.4. REST Interface Operation Headers

The following table displays headers that are included in the Red Hat JBoss Data Grid REST Interface:

**Table 11.1. Header Types**

| Headers | Mandatory/Optional | Values | Default Value | Details |
|---|---|---|---|---|
| Content-Type | Mandatory | - | - | If the Content-Type is set to **application/x-java-serialized-object**, it is stored as a Java object. |

| Headers | Mandatory/Optional | Values | Default Value | Details |
|---|---|---|---|---|
| performAsync | Optional | True/False | - | If set to `true`, an immediate return occurs, followed by a replication of data to the cluster on its own. This feature is useful when dealing with bulk data inserts and large clusters. |
| timeToLiveSeconds | Optional | Numeric (positive and negative numbers) | `-1` (This value prevents expiration as a direct result of timeToLiveSeconds. Expiration values set elsewhere override this default value.) | Reflects the number of seconds before the entry in question is automatically deleted. Setting a negative value for timeToLiveSeconds provides the same result as the default value. |
| maxIdleTimeSeconds | Optional | Numeric (positive and negative numbers) | `-1` (This value prevents expiration as a direct result of maxIdleTimeSeconds. Expiration values set elsewhere override this default value.) | Contains the number of seconds after the last usage when the entry will be automatically deleted. Passing a negative value provides the same result as the default value. |

The following combinations can be set for the ***`timeToLiveSeconds`*** and ***`maxIdleTimeSeconds`*** headers:

- If both the ***`timeToLiveSeconds`*** and ***`maxIdleTimeSeconds`*** headers are assigned the value **`0`**, the cache uses the default ***`timeToLiveSeconds`*** and ***`maxIdleTimeSeconds`*** values configured either using XML or programatically.
- If only the ***`maxIdleTimeSeconds`*** header value is set to **`0`**, the ***`timeToLiveSeconds`*** value should be passed as the parameter (or the default **`-1`**, if the parameter is not present). Additionally, the ***`maxIdleTimeSeconds`*** parameter value defaults to the values configured either using XML or programatically.

- If only the ***`timeToLiveSeconds`*** header value is set to **`0`**, expiration occurs immediately and the ***`maxIdleTimeSeconds`*** value is set to the value passed as a parameter (or the default **`-1`** if no parameter was supplied).

**ETag Based Headers**

ETags (Entity Tags) are returned for each REST Interface entry, along with a ***`Last-Modified`*** header that indicates the state of the data at the supplied URL. ETags are used in HTTP operations to request data exclusively in cases where the data has changed to save bandwidth. The following headers support ETags (Entity Tags) based optimistic locking:

**Table 11.2. Entity Tag Related Headers**

| Header | Algorithm | Example | Details |
|---|---|---|---|
| If-Match | If-Match = "If-Match" ":" ( "*" \| 1#entity-tag ) | - | Used in conjunction with a list of associated entity tags to verify that a specified entity (that was previously obtained from a resource) remains current. |
| If-None-Match | | - | Used in conjunction with a list of associated entity tags to verify that none of the specified entities (that was previously obtained from a resource) are current. This feature facilitates efficient updates of cached information when required and with minimal transaction overhead. |
| If-Modified-Since | If-Modified-Since = "If-Modified-Since" ":" HTTP-date | If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT | Compares the requested variant's last modification time and date with a supplied time and date value. If the requested variant has not been modified since the specified time and date, a **`304`** (not modified) response is returned without a message-body instead of an entity. |

| Header | Algorithm | Example | Details |
|---|---|---|---|
| If-Unmodified-Since | If-Unmodified-Since = "If-Unmodified-Since" ":" HTTP-date | If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT | Compares the requested variant's last modification time and date with a supplied time and date value. If the requested resources has not been modified since the supplied date and time, the specified operation is performed. If the requested resource has been modified since the supplied date and time, the operation is not performed and a **412** (Precondition Failed) response is returned. |

Report a bug

## 11.7. REST INTERFACE SECURITY

### 11.7.1. Publish REST Endpoints as a Public Interface

Red Hat JBoss Data Grid's REST server operates as a management interface by default. To extend its operations to a public interface, alter the value of the ***interface*** parameter in the **socket-binding** element from **management** to **public** as follows:

```
<socket-binding name="http" interface="public" port="8080"/>
```

Report a bug

### 11.7.2. Enable Security for the REST Endpoint

Use the following procedure to enable security for the REST endpoint in Red Hat JBoss Data Grid.

**NOTE**

The REST endpoint supports any of the JBoss Enterprise Application Platform security subsystem providers.

**Procedure 11.2. Enable Security for the REST Endpoint**

To enable security for JBoss Data Grid when using the REST interface, make the following changes to **standalone.xml**:

1. **Specify Security Parameters**

Ensure that the rest endpoint specifies a valid value for the ***security-domain*** and ***auth-method*** parameters. Recommended settings for these parameters are as follows:

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
            <rest-connector virtual-server="default-host"
                            cache-container="local"
                            security-domain="other"
                            auth-method="BASIC"/>
</subsystem>
```

2. **Check Security Domain Declaration**
   Ensure that the security subsystem contains the corresponding security-domain declaration. For details about setting up security-domain declarations, see the JBoss Enterprise Application Platform 6 documentation.

3. **Add an Application User**
   Run the relevant script and enter the configuration settings to add an application user.

   a. Run the **adduser.sh** script (located in **$JDG_HOME/bin**).

      - On a Windows system, run the **adduser.bat** file (located in **$JDG_HOME/bin**) instead.

   b. When prompted about the type of user to add, select **Application User (application-users.properties)** by entering **b**.

   c. Accept the default value for realm (**ApplicationRealm**) by pressing the return key.

   d. Specify a username and password.

   e. When prompted for a role for the created user, enter **REST**.

   f. Ensure the username and application realm information is correct when prompted and enter "yes" to continue.

4. **Verify the Created Application User**
   Ensure that the created application user is correctly configured.

   a. Check the configuration listed in the **application-users.properties** file (located in **$JDG_HOME/standalone/configuration/**). The following is an example of what the correct configuration looks like in this file:

      ```
      user1=2dc3eacfed8cf95a4a31159167b936fc
      ```

   b. Check the configuration listed in the **application-roles.properties** file (located in **$JDG_HOME/standalone/configuration/**). The following is an example of what the correct configuration looks like in this file:

      ```
      user1=REST
      ```

5. **Test the Server**
   Start the server and enter the following link in a browser window to access the REST endpoint:

```
http://localhost:8080/rest/namedCache
```

–

**NOTE**

If testing using a GET request, a **405** response code is expected and indicates that the server was successfully authenticated.

Report a bug

# CHAPTER 12. THE MEMCACHED INTERFACE

Memcached is an in-memory caching system used to improve response and operation times for database-driven websites. The Memcached caching system defines a text based protocol called the Memcached protocol. The Memcached protocol uses in-memory objects or (as a last resort) passes to a persistent store such as a special memcached database.

Red Hat JBoss Data Grid offers a server that uses the Memcached protocol, removing the necessity to use Memcached separately with JBoss Data Grid. Additionally, due to JBoss Data Grid's clustering features, its data failover capabilities surpass those provided by Memcached.

Report a bug

## 12.1. ABOUT MEMCACHED SERVERS

Red Hat JBoss Data Grid contains a server module that implements the memcached protocol. This allows memcached clients to interact with one or multiple JBoss Data Grid based memcached servers.

The servers can be either:

- Standalone, where each server acts independently without communication with any other memcached servers.
- Clustered, where servers replicate and distribute data to other memcached servers.

Report a bug

## 12.2. MEMCACHED STATISTICS

The following table contains a list of valid statistics available using the memcached protocol in Red Hat JBoss Data Grid.

**Table 12.1. Memcached Statistics**

| Statistic | Data Type | Details |
|---|---|---|
| uptime | 32-bit unsigned integer. | Contains the time (in seconds) that the memcached instance has been available and running. |
| time | 32-bit unsigned integer. | Contains the current time. |
| version | String | Contains the current version. |
| curr_items | 32-bit unsigned integer. | Contains the number of items currently stored by the instance. |
| total_items | 32-bit unsigned integer. | Contains the total number of items stored by the instance during its lifetime. |

| Statistic | Data Type | Details |
|---|---|---|
| cmd_get | 64-bit unsigned integer | Contains the total number of get operation requests (requests to retrieve data). |
| cmd_set | 64-bit unsigned integer | Contains the total number of set operation requests (requests to store data). |
| get_hits | 64-bit unsigned integer | Contains the number of keys that are present from the keys requested. |
| get_misses | 64-bit unsigned integer | Contains the number of keys that were not found from the keys requested. |
| delete_hits | 64-bit unsigned integer | Contains the number of keys to be deleted that were located and successfully deleted. |
| delete_misses | 64-bit unsigned integer | Contains the number of keys to be deleted that were not located and therefore could not be deleted. |
| incr_hits | 64-bit unsigned integer | Contains the number of keys to be incremented that were located and successfully incremented |
| incr_misses | 64-bit unsigned integer | Contains the number of keys to be incremented that were not located and therefore could not be incremented. |
| decr_hits | 64-bit unsigned integer | Contains the number of keys to be decremented that were located and successfully decremented. |
| decr_misses | 64-bit unsigned integer | Contains the number of keys to be decremented that were not located and therefore could not be decremented. |
| cas_hits | 64-bit unsigned integer | Contains the number of keys to be compared and swapped that were found and successfully compared and swapped. |

| Statistic | Data Type | Details |
|---|---|---|
| cas_misses | 64-bit unsigned integer | Contains the number of keys to be compared and swapped that were not found and therefore not compared and swapped. |
| cas_badval | 64-bit unsigned integer | Contains the number of keys where a compare and swap occurred but the original value did not match the supplied value. |
| evictions | 64-bit unsigned integer | Contains the number of eviction calls performed. |
| bytes_read | 64-bit unsigned integer | Contains the total number of bytes read by the server from the network. |
| bytes_written | 64-bit unsigned integer | Contains the total number of bytes written by the server to the network. |

Report a bug

## 12.3. THE MEMCACHED INTERFACE CONNECTOR

Red Hat JBoss Data Grid supports three connector types, namely:

- The **`hotrod-connector`** element, which defines the configuration for a Hot Rod based connector.
- The **`memcached-connector`** element, which defines the configuration for a memcached based connector.
- The **`rest-connector`** element, which defines the configuration for a REST interface based connector.

The connector declaration enables a Hot Rod, Memcached, or REST server using a socket binding, which is declared within a `<`***`socket-binding-group`***` />`, and exposing the caches declared in the **`local`** container, using defaults for all other settings. The following examples show how to connect to Hot Rod, Memcached, and REST servers.

The following enables a Memcached server using the **`memcached`** socket binding, and exposes the **`memcachedCache`** cache declared in the **`local`** container, using defaults for all other settings.

```
<memcached-connector socket-binding="memcached" cache-container="local"/>
```

Due to the limitations in the Memcached protocol, only one cache can be exposed by a connector. To expose more than one cache, declare additional memcached-connectors on different socket-bindings. See Section 12.3.1, “Configure Memcached Connectors”.

Report a bug

### 12.3.1. Configure Memcached Connectors

The following procedure describes the attributes used to configure the memcached connector within the `connectors` element in Red Hat JBoss Data Grid's Remote Client-Server Mode.

**Procedure 12.1. Configuring the Memcached Connector in Remote Client-Server Mode**

The `memcached-connector` element defines the configuration elements for use with memcached.

1. **The *`socket-binding`* Parameter**
   The *`socket-binding`* parameter specifies the socket binding port used by the memcached connector. This is a mandatory parameter.

   ```
   <subsystem xmlns="urn:infinispan:server:endpoint:6.1">
   <memcached-connector socket-binding="memcached" />
   ```

2. **The *`cache-container`* Parameter**
   The *`cache-container`* parameter names the cache container used by the memcached connector. This is a mandatory parameter.

   ```
   <subsystem xmlns="urn:infinispan:server:endpoint:6.1">
   <memcached-connector socket-binding="memcached"
                        cache-container="local" />
   ```

3. **The *`worker-threads`* Parameter**
   The *`worker-threads`* parameter specifies the number of worker threads available for the memcached connector. The default value for this parameter is 160. This is an optional parameter.

   ```
   <subsystem xmlns="urn:infinispan:server:endpoint:6.1">
   <memcached-connector socket-binding="memcached"
                        cache-container="local"
                        worker-threads="${VALUE}" />
   ```

4. **The *`idle-timeout`* Parameter**
   The *`idle-timeout`* parameter specifies the time (in milliseconds) the connector can remain idle before the connection times out. The default value for this parameter is `-1`, which means that no timeout period is set. This is an optional parameter.

   ```
   <subsystem xmlns="urn:infinispan:server:endpoint:6.1">
   <memcached-connector socket-binding="memcached"
                        cache-container="local"
                        worker-threads="${VALUE}"
                        idle-timeout="${VALUE}" />
   ```

5. **The *`tcp-nodelay`* Parameter**
   The *`tcp-nodelay`* parameter specifies whether TCP packets will be delayed and sent out in batches. Valid values for this parameter are `true` and `false`. The default value for this parameter is `true`. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
<memcached-connector socket-binding="memcached"
                     cache-container="local"
                     worker-threads="${VALUE}"
                     idle-timeout="{VALUE}"
                     tcp-nodelay="{TRUE/FALSE}"/>
```

6. **The *send-buffer-size* Parameter**

   The ***send-buffer-size*** parameter indicates the size of the send buffer for the memcached connector. The default value for this parameter is the size of the TCP stack buffer. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
<memcached-connector socket-binding="memcached"
                     cache-container="local"
                     worker-threads="${VALUE}"
                     idle-timeout="{VALUE}"
                     tcp-nodelay="{TRUE/FALSE}"
                     send-buffer-size="{VALUE}" />
```

7. **The *receive-buffer-size* Parameter**

   The ***receive-buffer-size*** parameter indicates the size of the receive buffer for the memcached connector. The default value for this parameter is the size of the TCP stack buffer. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
<memcached-connector socket-binding="memcached"
                     cache-container="local"
                     worker-threads="${VALUE}"
                     idle-timeout="{VALUE}"
                     tcp-nodelay="{TRUE/FALSE}"
                     send-buffer-size="{VALUE}"
                     receive-buffer-size="${VALUE}" />
</subsystem>
```

Report a bug

## 12.4. MEMCACHED INTERFACE SECURITY

### 12.4.1. Publish Memcached Endpoints as a Public Interface

Red Hat JBoss Data Grid's memcached server operates as a management interface by default. To extend its operations to a public interface, alter the value of the ***interface*** parameter in the **socket-binding** element from **management** to **public** as follows:

```
<socket-binding name="memcached" interface="public" port="11211" />
```

Report a bug

# CHAPTER 13. THE HOT ROD INTERFACE

## 13.1. ABOUT HOT ROD

Hot Rod is a binary TCP client-server protocol used in Red Hat JBoss Data Grid. It was created to overcome deficiencies in other client/server protocols, such as Memcached.

Hot Rod will failover on a server cluster that undergoes a topology change. Hot Rod achieves this by providing regular updates to clients about the cluster topology.

Hot Rod enables clients to do smart routing of requests in partitioned or distributed JBoss Data Grid server clusters. To do this, Hot Rod allows clients to determine the partition that houses a key and then communicate directly with the server that has the key. This functionality relies on Hot Rod updating the cluster topology with clients, and that the clients use the same consistent hash algorithm as the servers.

JBoss Data Grid contains a server module that implements the Hot Rod protocol. The Hot Rod protocol facilitates faster client and server interactions in comparison to other text-based protocols and allows clients to make decisions about load balancing, failover and data location operations.

Report a bug

## 13.2. THE BENEFITS OF USING HOT ROD OVER MEMCACHED

Red Hat JBoss Data Grid offers a choice of protocols for allowing clients to interact with the server in a Remote Client-Server environment. When deciding between using memcached or Hot Rod, the following should be considered.

**Memcached**

The memcached protocol causes the server endpoint to use the **memcached text wire protocol**. The **memcached wire protocol** has the benefit of being commonly used, and is available for almost any platform. All of JBoss Data Grid's functions, including clustering, state sharing for scalability, and high availability, are available when using memcached.

However the memcached protocol lacks dynamicity, resulting in the need to manually update the list of server nodes on your clients in the event one of the nodes in a cluster fails. Also, memcached clients are not aware of the location of the data in the cluster. This means that they will request data from a non-owner node, incurring the penalty of an additional request from that node to the actual owner, before being able to return the data to the client. This is where the Hot Rod protocol is able to provide greater performance than memcached.

**Hot Rod**

JBoss Data Grid's Hot Rod protocol is a binary wire protocol that offers all the capabilities of memcached, while also providing better scaling, durability, and elasticity.

The Hot Rod protocol does not need the hostnames and ports of each node in the remote cache, whereas memcached requires these parameters to be specified. Hot Rod clients automatically detect changes in the topology of clustered Hot Rod servers; when new nodes join or leave the cluster, clients update their Hot Rod server topology view. Consequently, Hot Rod provides ease of configuration and maintenance, with the advantage of dynamic load balancing and failover.

Additionally, the Hot Rod wire protocol uses smart routing when connecting to a distributed cache. This involves sharing a consistent hash algorithm between the server nodes and clients, resulting in faster read and writing capabilities than memcached.

Report a bug

## 13.3. HOT ROD HASH FUNCTIONS

Hot Rod uses the same algorithm as on the server. The Hot Rod client always connects to the primary owner of the key, which is the first node in the list of owners. For more information about consistent hashing in Red Hat JBoss Data Grid, see Section 7.2, "Distribution Mode's Consistent Hash Algorithm" .

Report a bug

## 13.4. THE HOT ROD INTERFACE CONNECTOR

Red Hat JBoss Data Grid supports three connector types, namely:

- The **`hotrod-connector`** element, which defines the configuration for a Hot Rod based connector.
- The **`memcached-connector`** element, which defines the configuration for a memcached based connector.
- The **`rest-connector`** element, which defines the configuration for a REST interface based connector.

The connector declaration enables a Hot Rod, Memcached, or REST server using a socket binding, which is declared within a ***`<socket-binding-group />`***, and exposing the caches declared in the **`local`** container, using defaults for all other settings. The following examples show how to connect to Hot Rod, Memcached, and REST servers.

The following enables a Hot Rod server using the **`hotrod`** socket binding.

```
<hotrod-connector socket-binding="hotrod" cache-container="local" />
```

The connector creates a supporting topology cache with default settings. These settings can be tuned by adding the ***`<topology-state-transfer />`*** child element to the connector as follows:

```
<hotrod-connector socket-binding="hotrod" cache-container="local">
   <topology-state-transfer lazy-retrieval="false" lock-timeout="1000"
replication-timeout="5000" />
</hotrod-connector>
```

The Hot Rod connector can be tuned with additional settings. See Section 13.4.1, "Configure Hot Rod Connectors" for more information on how to configure the Hot Rod connector.

**NOTE**

The Hot Rod connector can be secured using SSL. See the *Hot Rod Authentication Using SASL* section of the *Developer Guide* for more information.

Report a bug

### 13.4.1. Configure Hot Rod Connectors

The following procedure describes the attributes used to configure the Hot Rod connector in Red Hat JBoss Data Grid's Remote Client-Server Mode. Both the **`hotrod-connector`** and **`topology-state-transfer`** elements must be configured based on the following procedure.

**Procedure 13.1. Configuring Hot Rod Connectors for Remote Client-Server Mode**

1. **The `hotrod-connector` Element**
   The **`hotrod-connector`** element defines the configuration elements for use with Hot Rod.

   a. **The *`socket-binding`* Parameter**
      The ***`socket-binding`*** parameter specifies the socket binding port used by the Hot Rod connector. This is a mandatory parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
      <hotrod-connector socket-binding="hotrod" />
```

   b. **The *`cache-container`* Parameter**
      The ***`cache-container`*** parameter names the cache container used by the Hot Rod connector. This is a mandatory parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
      <hotrod-connector socket-binding="hotrod"
                        cache-container="local" />
```

   c. **The *`worker-threads`* Parameter**
      The ***`worker-threads`*** parameter specifies the number of worker threads available for the Hot Rod connector. The default value for this parameter is **`160`**. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
      <hotrod-connector socket-binding="hotrod"
                        cache-container="local"
                        worker-threads="${VALUE}" />
```

   d. **The *`idle-timeout`* Parameter**
      The ***`idle-timeout`*** parameter specifies the time (in milliseconds) the connector can remain idle before the connection times out. The default value for this parameter is **`-1`**, which means that no timeout period is set. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
      <hotrod-connector socket-binding="hotrod"
                        cache-container="local"
                        worker-threads="${VALUE}"
                        idle-timeout="${VALUE}"/>
```

   e. **The *`tcp-nodelay`* Parameter**
      The ***`tcp-nodelay`*** parameter specifies whether TCP packets will be delayed and sent out in batches. Valid values for this parameter are **`true`** and **`false`**. The default value for this parameter is **`true`**. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
      <hotrod-connector socket-binding="hotrod"
```

```
                              cache-container="local"
                              worker-threads="${VALUE}"
                              idle-timeout="${VALUE}"
                              tcp-nodelay="${TRUE/FALSE}" />
```

f. **The *send-buffer-size* Parameter**

The ***send-buffer-size*** parameter indicates the size of the send buffer for the Hot Rod connector. The default value for this parameter is the size of the TCP stack buffer. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
      <hotrod-connector socket-binding="hotrod"
                        cache-container="local"
                        worker-threads="${VALUE}"
                        idle-timeout="${VALUE}"
                        tcp-nodelay="${TRUE/FALSE}"
                        send-buffer-size="${VALUE}"/>
```

g. **The *receive-buffer-size* Parameter**

The ***receive-buffer-size*** parameter indicates the size of the receive buffer for the Hot Rod connector. The default value for this parameter is the size of the TCP stack buffer. This is an optional parameter.

```
subsystem xmlns="urn:infinispan:server:endpoint:6.1">
      <hotrod-connector socket-binding="hotrod"
                        cache-container="local"
                        worker-threads="${VALUE}"
                        idle-timeout="${VALUE}"
                        tcp-nodelay="${TRUE/FALSE}"
                        send-buffer-size="${VALUE}"
                        receive-buffer-size="${VALUE}"   />
```

2. **The `topology-state-transfer` Element**

The **`topology-state-transfer`** element specifies the topology state transfer configurations for the Hot Rod connector. This element can only occur once within a **`hotrod-connector`** element.

a. **The *lock-timeout* Parameter**

The ***lock-timeout*** parameter specifies the time (in milliseconds) after which the operation attempting to obtain a lock times out. The default value for this parameter is **`10`** seconds. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
      <hotrod-connector socket-binding="hotrod"
                        cache-container="local"
                        worker-threads="${VALUE}"
                        idle-timeout="${VALUE}"
                        tcp-nodelay="${TRUE/FALSE}"
                        send-buffer-size="${VALUE}"
                        receive-buffer-size="${VALUE}"   />
      <topology-state-transfer lock-timeout"="${MILLISECONDS}" />
```

b. **The *replication-timeout* Parameter**

The ***replication-timeout*** parameter specifies the time (in milliseconds) after which the replication operation times out. The default value for this parameter is **10** seconds. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
      <hotrod-connector socket-binding="hotrod"
                       cache-container="local"
                       worker-threads="${VALUE}"
                       idle-timeout="${VALUE}"
                       tcp-nodelay="${TRUE/FALSE}"
                       send-buffer-size="${VALUE}"
                       receive-buffer-size="${VALUE}"   />
      <topology-state-transfer lock-timeout"="${MILLISECONDS}"
                              replication-
timeout="${MILLISECONDS}" />
```

c. **The *external-host* Parameter**

The ***external-host*** parameter specifies the hostname sent by the Hot Rod server to clients listed in the topology information. The default value for this parameter is the host address. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
      <hotrod-connector socket-binding="hotrod"
                       cache-container="local"
                       worker-threads="${VALUE}"
                       idle-timeout="${VALUE}"
                       tcp-nodelay="${TRUE/FALSE}"
                       send-buffer-size="${VALUE}"
                       receive-buffer-size="${VALUE}"   />
      <topology-state-transfer lock-timeout"="${MILLISECONDS}"
                              replication-
timeout="${MILLISECONDS}"
                              external-host="${HOSTNAME}" />
```

d. **The *external-port* Parameter**

The ***external-port*** parameter specifies the port sent by the Hot Rod server to clients listed in the topology information. The default value for this parameter is the configured port. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
      <hotrod-connector socket-binding="hotrod"
                       cache-container="local"
                       worker-threads="${VALUE}"
                       idle-timeout="${VALUE}"
                       tcp-nodelay="${TRUE/FALSE}"
                       send-buffer-size="${VALUE}"
                       receive-buffer-size="${VALUE}"   />
      <topology-state-transfer lock-timeout"="${MILLISECONDS}"
                              replication-
timeout="${MILLISECONDS}"
                              external-host="${HOSTNAME}"
                              external-port="${PORT}" />
```

e. **The *lazy-retrieval* Parameter**

The ***lazy-retrieval*** parameter indicates whether the Hot Rod connector will carry out retrieval operations lazily. The default value for this parameter is **true**. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
 <hotrod-connector socket-binding="hotrod"
     cache-container="local"
     worker-threads="${VALUE}"
     idle-timeout="${VALUE}"
     tcp-nodelay="${TRUE/FALSE}"
     send-buffer-size="${VALUE}"
     receive-buffer-size="${VALUE}" />
 <topology-state-transfer lock-timeout"="${MILLISECONDS}"
     replication-timeout="${MILLISECONDS}"
     external-host="${HOSTNAME}"
     external-port="${PORT}"
     lazy-retrieval="${TRUE/FALSE}" />
</subsystem>
```

f. **The *await-initial-transfer* Parameter**

The ***await-initial-transfer*** parameter specifies whether the initial state retrieval happens immediately at startup. This parameter only applies when ***lazy-retrieval*** is set to **false**. This default value for this parameter is **true**.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.1">
 <hotrod-connector socket-binding="hotrod"
     cache-container="local"
     worker-threads="${VALUE}"
     idle-timeout="${VALUE}"
     tcp-nodelay="${TRUE/FALSE}"
     send-buffer-size="${VALUE}"
     receive-buffer-size="${VALUE}" />
 <topology-state-transfer lock-timeout"="${MILLISECONDS}"
     replication-timeout="${MILLISECONDS}"
     external-host="${HOSTNAME}"
     external-port="${PORT}"
     lazy-retrieval="${TRUE/FALSE}"
     await-initial-transfer="${TRUE/FALSE}" />
</subsystem>
```

Report a bug

## 13.5. HOT ROD HEADERS

### 13.5.1. Hot Rod Header Data Types

All keys and values used for Hot Rod in Red Hat JBoss Data Grid are stored as byte arrays. Certain header values, such as those for REST and Memcached, are stored using the following data types instead:

**Table 13.1. Header Data Types**

| Data Type | Size | Details |
|---|---|---|
| vInt | Between 1-5 bytes. | Unsigned variable length integer values. |
| vLong | Between 1-9 bytes. | Unsigned variable length long values. |
| string | - | Strings are always represented using UTF-8 encoding. |

Report a bug

### 13.5.2. Request Header

When using Hot Rod to access Red Hat JBoss Data Grid, the contents of the request header consist of the following:

**Table 13.2. Request Header Fields**

| Field Name | Data Type/Size | Details |
|---|---|---|
| Magic | 1 byte | Indicates whether the header is a request header or response header. |
| Message ID | vLong | Contains the message ID. Responses use this unique ID when responding to a request. This allows Hot Rod clients to implement the protocol in an asynchronous manner. |
| Version | 1 byte | Contains the Hot Rod server version. |
| Opcode | 1 byte | Contains the relevant operation code. In a request header, opcode can only contain the request operation codes. |
| Cache Name Length | vInt | Stores the length of the cache name. If Cache Name Length is set to `0` and no value is supplied for Cache Name, the operation interacts with the default cache. |

| Field Name | Data Type/Size | Details |
|---|---|---|
| Cache Name | string | Stores the name of the target cache for the specified operation. This name must match the name of a predefined cache in the cache configuration file. |
| Flags | vInt | Contains a numeric value of variable length that represents flags passed to the system. Each bit represents a flag, except the most significant bit, which is used to determine whether more bytes must be read. Using a bit to represent each flag facilitates the representation of flag combinations in a condensed manner. |
| Client Intelligence | 1 byte | Contains a value that indicates the client capabilities to the server. |
| Topology ID | vInt | Contains the last known view ID in the client. Basic clients supply the value `0` for this field. Clients that support topology or hash information supply the value `0` until the server responds with the current view ID, which is subsequently used until a new view ID is returned by the server to replace the current view ID. |
| Transaction Type | 1 byte | Contains a value that represents one of two known transaction types. Currently, the only supported value is `0`. |
| Transaction ID | byte-array | Contains a byte array that uniquely identifies the transaction associated with the call. The transaction type determines the length of this byte array. If the value for ***`Transaction Type`*** was set to `0`, no Transaction ID is present. |

Report a bug

### 13.5.3. Response Header

When using Hot Rod to access Red Hat JBoss Data Grid, the contents of the response header consist of the following:

**Table 13.3. Response Header Fields**

| Field Name | Data Type | Details |
|---|---|---|
| Magic | 1 byte | Indicates whether the header is a request or response header. |
| Message ID | vLong | Contains the message ID. This unique ID is used to pair the response with the original request. This allows Hot Rod clients to implement the protocol in an asynchronous manner. |
| Opcode | 1 byte | Contains the relevant operation code. In a response header, opcode can only contain the response operation codes. |
| Status | 1 byte | Contains a code that represents the status of the response. |
| Topology Change Marker | 1 byte | Contains a marker byte that indicates whether the response is included in the topology change information. |

Report a bug

## 13.5.4. Topology Change Headers

When using Hot Rod to access Red Hat JBoss Data Grid, response headers respond to changes in the cluster or view formation by looking for clients that can distinguish between different topologies or hash distributions. The Hot Rod server compares the current ***topology ID*** and the ***topology ID*** sent by the client and, if the two differ, it returns a new ***topology ID***.

Report a bug

### 13.5.4.1. Topology Change Marker Values

The following is a list of valid values for the ***Topology Change Marker*** field in a response header:

**Table 13.4. Topology Change Marker Field Values**

| Value | Details |
|---|---|
| 0 | No topology change information is added. |

| Value | Details |
|---|---|
| 1 | Topology change information is added. |

Report a bug

### 13.5.4.2. Topology Change Headers for Topology-Aware Clients

The response header sent to topology-aware clients when a topology change is returned by the server includes the following elements:

**Table 13.5. Topology Change Header Fields**

| Response Header Fields | Data Type/Size | Details |
|---|---|---|
| Response Header with Topology Change Marker | - | - |
| Topology ID | vInt | - |
| Num Servers in Topology | vInt | Contains the number of Hot Rod servers running in the cluster. This value can be a subset of the entire cluster if only some nodes are running Hot Rod servers. |
| mX: Host/IP Length | vInt | Contains the length of the hostname or IP address of an individual cluster member. Variable length allows this element to include hostnames, IPv4 and IPv6 addresses. |
| mX: Host/IP Address | string | Contains the hostname or IP address of an individual cluster member. The Hot Rod client uses this information to access the individual cluster member. |
| mX: Port | Unsigned Short. 2 bytes | Contains the port used by Hot Rod clients to communicate with the cluster member. |

The three entries with the prefix **`mX`**, are repeated for each server in the topology. The first server in the topology's information fields will be prefixed with **`m1`** and the numerical value is incremented by one for each additional server till the value of **`X`** equals the number of servers specified in the ***`num servers in topology`*** field.

Report a bug

### 13.5.4.3. Topology Change Headers for Hash Distribution-Aware Clients

The response header sent to clients when a topology change is returned by the server includes the following elements:

**Table 13.6. Topology Change Header Fields**

| Field | Data Type/Size | Details |
|---|---|---|
| Response Header with Topology Change Marker | - | - |
| Topology ID | vInt | - |
| Number Key Owners | Unsigned short. 2 bytes. | Contains the number of globally configured copies for each distributed key. Contains the value `0` if distribution is not configured on the cache. |
| Hash Function Version | 1 byte | Contains a pointer to the hash function in use. Contains the value `0` if distribution is not configured on the cache. |
| Hash Space Size | vInt | Contains the modulus used by JBoss Data Grid for all module arithmetic related to hash code generation. Clients use this information to apply the correct hash calculations to the keys. Contains the value `0` if distribution is not configured on the cache. |
| Number servers in topology | vInt | Contains the number of **`Hot Rod`** servers running in the cluster. This value can be a subset of the entire cluster if only some nodes are running **`Hot Rod`** servers. This value also represents the number of host to port pairings included in the header. |
| Number Virtual Nodes Owners | vInt | Contains the number of configured virtual nodes. Contains the value `0` if no virtual nodes are configured or if distribution is not configured on the cache. |

| Field | Data Type/Size | Details |
|---|---|---|
| mX: Host/IP Length | vInt | Contains the length of the hostname or **IP** address of an individual cluster member. Variable length allows this element to include hostnames, **IPv4** and **IPv6** addresses. |
| mX: Host/IP Address | string | Contains the hostname or **IP** address of an individual cluster member. The **Hot Rod** client uses this information to access the individual cluster member. |
| mX: Port | Unsigned short. 2 bytes. | Contains the port used by **Hot Rod** clients to communicate with the cluster member. |
| mX: Hashcode | 4 bytes. | |

The three entries with the prefix `mX`, are repeated for each server in the topology. The first server in the topology's information fields will be prefixed with `m1` and the numerical value is incremented by one for each additional server till the value of `X` equals the number of servers specified in the ***num servers in topology*** field.

Report a bug

## 13.6. HOT ROD OPERATIONS

The following are valid operations when using Hot Rod protocol 1.3 to interact with Red Hat JBoss Data Grid:

- BulkGetKeys
- BulkGet
- Clear
- ContainsKey
- Get
- GetWithMetadata
- Ping
- PutIfAbsent
- Put
- Query

- RemoveIfUnmodified
- Remove
- ReplaceIfUnmodified
- Replace
- Stats

**IMPORTANT**

When using the RemoteCache API to call the Hot Rod client's **`Put`**, **`PutIfAbsent`**, **`Replace`**, and **`ReplaceWithVersion`** operations, if lifespan is set to a value greater than 30 days, the value is treated as UNIX time and represents the number of seconds since the date 1/1/1970.

Report a bug

### 13.6.1. Hot Rod BulkGet Operation

A Hot Rod **`BulkGet`** operation uses the following request format:

**Table 13.7. BulkGet Operation Request Format**

| Field | Data Type | Details |
|---|---|---|
| Header | - | - |
| Entry Count | vInt | Contains the maximum number of Red Hat JBoss Data Grid entries to be returned by the server. The entry is the key and value pair. |

The response header for this operation contains one of the following response statuses:

**Table 13.8. BulkGet Operation Response Format**

| Field | Data Type | Details |
|---|---|---|
| Header | - | - |
| More | vInt | Represents if more entries must be read from the stream. While ***`More`*** is set to**`1`**, additional entries follow until the value of More is set to **`0`**, which indicates the end of the stream. |
| Key Size | - | Contains the size of the key. |

| Field | Data Type | Details |
|---|---|---|
| Key | - | Contains the key value. |
| Value Size | - | Contains the size of the value. |
| Value | - | Contains the value. |

For each entry that was requested, a ***More***, ***Key Size***, ***Key***, ***Value Size*** and ***Value*** entry is appended to the response.

Report a bug

### 13.6.2. Hot Rod BulkGetKeys Operation

A Hot Rod **BulkGetKeys** operation uses the following request format:

**Table 13.9. BulkGetKeys Operation Request Format**

| Field | Data Type | Details |
|---|---|---|
| Header | variable | Request header. |

| Field | Data Type | Details |
|---|---|---|
| Scope | vInt | <ul><li>`0` = Default Scope - This scope is used by `RemoteCache.keySet()` method. If the remote cache is a distributed cache, the server launches a map/reduce operation to retrieve all keys from all of the nodes (A topology-aware Hot Rod Client could be load balancing the request to any one node in the cluster). Otherwise, it will get keys from the cache instance local to the server receiving the request, as the keys must be the same across all nodes in a replicated cache.</li><li>`1` = Global Scope - This scope behaves the same to Default Scope.</li><li>`2` = Local Scope - In situations where the remote cache is a distributed cache, the server will not launch a map/reduce operation to retrieve keys from all nodes. Instead, it will only get keys local from the cache instance local to the server receiving the request.</li></ul> |

The response header for this operation contains one of the following response statuses:

**Table 13.10. BulkGetKeys Operation Response Format**

| Field | Data Type | Details |
|---|---|---|
| Header | variable | Response header |
| Response status | 1 byte | `0x00` = success, data follows. |

| Field | Data Type | Details |
|---|---|---|
| More | 1 byte | One byte representing whether more keys need to be read from the stream. When set to **1** an entry follows, when set to **0**, it is the end of stream and no more entries are left to read. |
| Key 1 Length | vInt | Length of key |
| Key 1 | Byte array | Retrieved key. |
| More | 1 byte | - |
| Key 2 Length | vInt | - |
| Key 2 | byte array | - |
| ...etc | | |

Report a bug

### 13.6.3. Hot Rod Clear Operation

The `clear` operation format includes only a header.

Valid response statuses for this operation are as follows:

**Table 13.11. Clear Operation Response**

| Response Status | Details |
|---|---|
| 0x00 | Red Hat JBoss Data Grid was successfully cleared. |

Report a bug

### 13.6.4. Hot Rod ContainsKey Operation

A Hot Rod `ContainsKey` operation uses the following request format:

**Table 13.12. ContainsKey Operation Request Format**

| Field | Data Type | Details |
|---|---|---|
| Header | - | - |

| Field | Data Type | Details |
|---|---|---|
| Key Length | vInt | Contains the length of the key. The `vInt` data type is used because of its size (up to **5** bytes), which is larger than the size of ***`Integer.MAX_VALUE`***. However, Java disallows single array sizes to exceed the size of ***`Integer.MAX_VALUE`***. As a result, this `vInt` is also limited to the maximum size of `Integer.MAX_VALUE`. |
| Key | Byte array | Contains a key, the corresponding value of which is requested. |

The response header for this operation contains one of the following response statuses:

**Table 13.13. ContainsKey Operation Response Format**

| Response Status | Details |
|---|---|
| 0x00 | Successful operation. |
| 0x02 | The key does not exist. |

The response for this operation is empty.

Report a bug

### 13.6.5. Hot Rod Get Operation

A Hot Rod `Get` operation uses the following request format:

**Table 13.14. Get Operation Request Format**

| Field | Data Type | Details |
|---|---|---|
| Header | - | - |

| Field | Data Type | Details |
|---|---|---|
| Key Length | vInt | Contains the length of the key. The **`vInt`** data type is used because of its size (up to **5** bytes), which is larger than the size of ***`Integer.MAX_VALUE`***. However, Java disallows single array sizes to exceed the size of ***`Integer.MAX_VALUE`***. As a result, this vInt is also limited to the maximum size of ***`Integer.MAX_VALUE`***. |
| Key | Byte array | Contains a key, the corresponding value of which is requested. |

The response header for this operation contains one of the following response statuses:

**Table 13.15. Get Operation Response Format**

| Response Status | Details |
|---|---|
| 0x00 | Successful operation. |
| 0x02 | The key does not exist. |

The format of the `get` operation's response when the key is found is as follows:

**Table 13.16. Get Operation Response Format**

| Field | Data Type | Details |
|---|---|---|
| Header | - | - |
| Value Length | vInt | Contains the length of the value. |
| Value | Byte array | Contains the requested value. |

Report a bug

### 13.6.6. Hot Rod GetWithMetadata Operation

A Hot Rod **`GetWithMetadata`** operation uses the following request format:

**Table 13.17. GetWithMetadata Operation Request Format**

| Field | Data Type | Details |
|---|---|---|
| Header | variable | Request header. |
| Key Length | vInt | Length of key. Note that the size of a vInt can be up to five bytes, which theoretically can produce bigger numbers than `Integer.MAX_VALUE`. However, Java cannot create a single array that is bigger than `Integer.MAX_VALUE`, hence the protocol limits vInt array lengths to `Integer.MAX_VALUE`. |
| Key | byte array | Byte array containing the key whose value is being requested. |

The response header for this operation contains one of the following response statuses:

**Table 13.18. GetWithMetadata Operation Response Format**

| Field | Data Type | Details |
|---|---|---|
| Header | variable | Response header. |
| Response status | 1 byte | `0x00` = success, if key retrieved.<br><br>`0x02` = if key does not exist. |
| Flag | 1 byte | A flag indicating whether the response contains expiration information. The value of the flag is obtained as a bitwise OR operation between `INFINITE_LIFESPAN (0x01)` and `INFINITE_MAXIDLE (0x02)`. |
| Created | Long | (optional) a Long representing the timestamp when the entry was created on the server. This value is returned only if the flag's `INFINITE_LIFESPAN` bit is not set. |

| Field | Data Type | Details |
|---|---|---|
| Lifespan | vInt | (optional) a vInt representing the lifespan of the entry in seconds. This value is returned only if the flag's **`INFINITE_LIFESPAN`** bit is not set. |
| LastUsed | Long | (optional) a Long representing the timestamp when the entry was last accessed on the server. This value is returned only if the flag's **`INFINITE_MAXIDLE`** bit is not set. |
| MaxIdle | vInt | (optional) a vInt representing the maxIdle of the entry in seconds. This value is returned only if the flag's **`INFINITE_MAXIDLE`** bit is not set. |
| Entry Version | 8 bytes | Unique value of an existing entry modification. The protocol does not mandate that entry_version values are sequential, however they need to be unique per update at the key level. |
| Value Length | vInt | If success, length of value. |
| Value | byte array | If success, the requested value. |

Report a bug

### 13.6.7. Hot Rod Ping Operation

The `ping` is an application level request to check for server availability.

Valid response statuses for this operation are as follows:

**Table 13.19. Ping Operation Response**

| Response Status | Details |
|---|---|
| 0x00 | Successful ping without any errors. |

Report a bug

### 13.6.8. Hot Rod Put Operation

The **put** operation request format includes the following:

**Table 13.20.**

| Field | Data Type | Details |
|---|---|---|
| Header | - | - |
| Key Length | - | Contains the length of the key. |
| Key | Byte array | Contains the key value. |
| Lifespan | vInt | Contains the number of seconds before the entry expires. If the number of seconds exceeds thirty days, the value is treated as UNIX time (i.e. the number of seconds since the date **1/1/1970**) as the entry lifespan. When set to the value **0**, the entry will never expire. |
| Max Idle | vInt | Contains the number of seconds an entry is allowed to remain idle before it is evicted from the cache. If this entry is set to **0**, the entry is allowed to remain idle indefinitely without being evicted due to the ***max idle*** value. |
| Value Length | vInt | Contains the length of the value. |
| Value | Byte array | The requested value. |

The following are the value response values returned from this operation:

**Table 13.21.**

| Response Status | Details |
|---|---|
| 0x00 | The value was successfully stored. |

An empty response is the default response for this operation. However, if ***ForceReturnPreviousValue*** is passed, the previous value and key are returned. If the previous key and value do not exist, the value length would contain the value **0**.

**IMPORTANT**

When using the RemoteCache API to call the Hot Rod client's `Put`, `PutIfAbsent`, `Replace`, and `ReplaceWithVersion` operations, if lifespan is set to a value greater than 30 days, the value is treated as UNIX time and represents the number of seconds since the date 1/1/1970.

Report a bug

### 13.6.9. Hot Rod PutIfAbsent Operation

The `putIfAbsent` operation request format includes the following:

**Table 13.22. PutIfAbsent Operation Request Fields**

| Field | Data Type | Details |
|---|---|---|
| Header | - | - |
| Key Length | vInt | Contains the length of the key. |
| Key | Byte array | Contains the key value. |
| Lifespan | vInt | Contains the number of seconds before the entry expires. If the number of seconds exceeds thirty days, the value is treated as UNIX time (i.e. the number of seconds since the date `1/1/1970`) as the entry lifespan. When set to the value `0`, the entry will never expire. |
| Max Idle | vInt | Contains the number of seconds an entry is allowed to remain idle before it is evicted from the cache. If this entry is set to `0`, the entry is allowed to remain idle indefinitely without being evicted due to the max idle value. |
| Value Length | vInt | Contains the length of the value. |
| Value | Byte array | Contains the requested value. |

The following are the value response values returned from this operation:

**Table 13.23.**

| Response Status | Details |
|---|---|
| 0x00 | The value was successfully stored. |
| 0x01 | The key was present, therefore the value was not stored. The current value of the key is returned. |

An empty response is the default response for this operation. However, if ***ForceReturnPreviousValue*** is passed, the previous value and key are returned. If the previous key and value do not exist, the value length would contain the value **0**.

**IMPORTANT**

When using the RemoteCache API to call the Hot Rod client's **Put**, **PutIfAbsent**, **Replace**, and **ReplaceWithVersion** operations, if lifespan is set to a value greater than 30 days, the value is treated as UNIX time and represents the number of seconds since the date 1/1/1970.

Report a bug

### 13.6.10. Hot Rod Query Operation

The **Query** operation request format includes the following:

**Table 13.24. Query Operation Request Fields**

| Field | Data Type | Details |
|---|---|---|
| Header | variable | Request header. |
| Query Length | vInt | The length of the Protobuf encoded query object. |
| Query | Byte array | Byte array containing the Protobuf encoded query object, having a length specified by previous field. |

The following are the value response values returned from this operation:

**Table 13.25. Query Operation Response**

| Response Status | Data | Details |
|---|---|---|
| Header | variable | Response header. |
| Response payload Length | vInt | The length of the Protobuf encoded response object. |

| Response Status | Data | Details |
|---|---|---|
| Response payload | Byte array | Byte array containing the Protobuf encoded response object, having a length specified by previous field. |

Report a bug

### 13.6.11. Hot Rod Remove Operation

A **`Hot Rod Remove`** operation uses the following request format:

**Table 13.26. Remove Operation Request Format**

| Field | Data Type | Details |
|---|---|---|
| Header | - | - |
| Key Length | vInt | Contains the length of the key. The **`vInt`** data type is used because of its size (up to **5** bytes), which is larger than the size of ***`Integer.MAX_VALUE`***. However, Java disallows single array sizes to exceed the size of ***`Integer.MAX_VALUE`***. As a result, this **`vInt`** is also limited to the maximum size of **`Integer.MAX_VALUE`**. |
| Key | Byte array | Contains a key, the corresponding value of which is requested. |

The response header for this operation contains one of the following response statuses:

**Table 13.27. Remove Operation Response Format**

| Response Status | Details |
|---|---|
| 0x00 | Successful operation. |
| 0x02 | The key does not exist. |

Normally, the response header for this operation is empty. However, if ***`ForceReturnPreviousValue`*** is passed, the response header contains either:

- The value and length of the previous key.

- The value length `0` and the response status `0x02` to indicate that the key does not exist.

The remove operation's response header contains the previous value and the length of the previous value for the provided key if ***`ForceReturnPreviousValue`*** is passed. If the key does not exist or the previous value was null, the value length is `0`.

Report a bug

### 13.6.12. Hot Rod RemoveIfUnmodified Operation

The **`RemoveIfUnmodified`** operation request format includes the following:

**Table 13.28. RemoveIfUnmodified Operation Request Fields**

| Field | Data Type | Details |
|---|---|---|
| Header | - | - |
| Key Length | vInt | Contains the length of the key. |
| Key | Byte array | Contains the key value. |
| Entry Version | 8 bytes | The version number for the entry. |

The following are the value response values returned from this operation:

**Table 13.29. RemoveIfUnmodified Operation Response**

| Response Status | Details |
|---|---|
| 0x00 | Returned status if the entry was replaced or removed. |
| 0x01 | Returns status if the entry replace or remove was unsuccessful because the key was modified. |
| 0x02 | Returns status if the key does not exist. |

An empty response is the default response for this operation. However, if ***`ForceReturnPreviousValue`*** is passed, the previous value and key are returned. If the previous key and value do not exist, the value length would contain the value `0`.

Report a bug

### 13.6.13. Hot Rod Replace Operation

The **`replace`** operation request format includes the following:

**Table 13.30. Replace Operation Request Fields**

| Field | Data Type | Details |
|---|---|---|
| Header | - | - |
| Key Length | vInt | Contains the length of the key. |
| Key | Byte array | Contains the key value. |
| Lifespan | vInt | Contains the number of seconds before the entry expires. If the number of seconds exceeds thirty days, the value is treated as UNIX time (i.e. the number of seconds since the date **`1/1/1970`**) as the entry lifespan. When set to the value **`0`**, the entry will never expire. |
| Max Idle | vInt | Contains the number of seconds an entry is allowed to remain idle before it is evicted from the cache. If this entry is set to **`0`**, the entry is allowed to remain idle indefinitely without being evicted due to the max idle value. |
| Value Length | vInt | Contains the length of the value. |
| Value | Byte array | Contains the requested value. |

The following are the value response values returned from this operation:

**Table 13.31. Replace Operation Response**

| Response Status | Details |
|---|---|
| 0x00 | The value was successfully stored. |
| 0x01 | The value was not stored because the key does not exist. |

An empty response is the default response for this operation. However, if ***`ForceReturnPreviousValue`*** is passed, the previous value and key are returned. If the previous key and value do not exist, the value length would contain the value **`0`**.

**IMPORTANT**

When using the RemoteCache API to call the Hot Rod client's **`Put`**, **`PutIfAbsent`**, **`Replace`**, and **`ReplaceWithVersion`** operations, if lifespan is set to a value greater than 30 days, the value is treated as UNIX time and represents the number of seconds since the date 1/1/1970.

Report a bug

### 13.6.14. Hot Rod ReplaceWithVersion Operation

The **`ReplaceWithVersion`** operation request format includes the following:

**NOTE**

In the RemoteCache API, the Hot Rod **`ReplaceWithVersion`** operation uses the **`ReplaceIfUnmodified`** operation. As a result, these two operations are exactly the same in JBoss Data Grid.

**Table 13.32. ReplaceWithVersion Operation Request Fields**

| Field | Data Type | Details |
|---|---|---|
| Header | - | - |
| Key Length | vInt | Contains the length of the key. |
| Key | Byte array | Contains the key value. |
| Lifespan | vInt | Contains the number of seconds before the entry expires. If the number of seconds exceeds thirty days, the value is treated as UNIX time (i.e. the number of seconds since the date **`1/1/1970`**) as the entry lifespan. When set to the value **`0`**, the entry will never expire. |
| Max Idle | vInt | Contains the number of seconds an entry is allowed to remain idle before it is evicted from the cache. If this entry is set to **`0`**, the entry is allowed to remain idle indefinitely without being evicted due to the max idle value. |
| Entry Version | 8 bytes | The version number for the entry. |
| Value Length | vInt | Contains the length of the value. |

| Field | Data Type | Details |
|---|---|---|
| Value | Byte array | Contains the requested value. |

The following are the value response values returned from this operation:

**Table 13.33. ReplaceWithVersion Operation Response**

| Response Status | Details |
|---|---|
| 0x00 | Returned status if the entry was replaced or removed. |
| 0x01 | Returns status if the entry replace or remove was unsuccessful because the key was modified. |
| 0x02 | Returns status if the key does not exist. |

An empty response is the default response for this operation. However, if ***`ForceReturnPreviousValue`*** is passed, the previous value and key are returned. If the previous key and value do not exist, the value length would contain the value **`0`**.

Report a bug

## 13.6.15. Hot Rod Stats Operation

This operation returns a summary of all available statistics. For each returned statistic, a name and value is returned in both string and UTF-8 formats.

The following are supported statistics for this operation:

**Table 13.34. Stats Operation Request Fields**

| Name | Details |
|---|---|
| timeSinceStart | Contains the number of seconds since Hot Rod started. |
| currentNumberOfEntries | Contains the number of entries that currently exist in the Hot Rod server. |
| totalNumberOfEntries | Contains the total number of entries stored in the Hot Rod server. |
| stores | Contains the number of put operations attempted. |
| retrievals | Contains the number of get operations attempted. |
| hits | Contains the number of get hits. |

| Name | Details |
|---|---|
| misses | Contains the number of get misses. |
| removeHits | Contains the number of remove hits. |
| removeMisses | Contains the number of removal misses. |

The response header for this operation contains the following:

**Table 13.35. Stats Operation Response**

| Name | Data Type | Details |
|---|---|---|
| Header | - | - |
| Number of Stats | vInt | Contains the number of individual statistics returned. |
| Name Length | vInt | Contains the length of the named statistic. |
| Name | string | Contains the name of the statistic. |
| Value Length | vInt | Contains the length of the value. |
| Value | string | Contains the statistic value. |

The values ***`Name Length`***, ***`Name`***, ***`Value Length`*** and ***`Value`*** recur for each statistic requested.

Report a bug

## 13.7. HOT ROD OPERATION VALUES

The following is a list of valid ***opcode*** values for a request header and their corresponding response header values:

**Table 13.36. Opcode Request and Response Header Values**

| Operation | Request Operation Code | Response Operation Code |
|---|---|---|
| put | 0x01 | 0x02 |
| get | 0x03 | 0x04 |
| putIfAbsent | 0x05 | 0x06 |

| Operation | Request Operation Code | Response Operation Code |
|---|---|---|
| replace | 0x07 | 0x08 |
| replaceIfUnmodified | 0x09 | 0x0A |
| remove | 0x0B | 0x0C |
| removeIfUnmodified | 0x0D | 0x0E |
| containsKey | 0x0F | 0x10 |
| clear | 0x13 | 0x14 |
| stats | 0x15 | 0x16 |
| ping | 0x17 | 0x18 |
| bulkGet | 0x19 | 0x1A |
| getWithMetadata | 0x1B | 0x1C |
| bulkKeysGet | 0x1D | 0x1E |
| query | 0x1F | 0x20 |

Additionally, if the response header ***opcode*** value is **`0x50`**, it indicates an error response.

Report a bug

### 13.7.1. Magic Values

The following is a list of valid values for the ***Magic*** field in request and response headers:

**Table 13.37. Magic Field Values**

| Value | Details |
|---|---|
| 0xA0 | Cache request marker. |
| 0xA1 | Cache response marker. |

Report a bug

### 13.7.2. Status Values

The following is a table that contains all valid values for the ***Status*** field in a response header:

**Table 13.38. Status Values**

| Value | Details |
|---|---|
| 0x00 | No error. |
| 0x01 | Not put/removed/replaced. |
| 0x02 | Key does not exist. |
| 0x81 | Invalid Magic value or Message ID. |
| 0x82 | Unknown command. |
| 0x83 | Unknown version. |
| 0x84 | Request parsing error. |
| 0x85 | Server error. |
| 0x86 | Command timed out. |

Report a bug

### 13.7.3. Transaction Type Values

The following is a list of valid values for ***`Transaction Type`*** in a request header:

**Table 13.39. Transaction Type Field Values**

| Value | Details |
|---|---|
| 0 | Indicates a non-transactional call or that the client does not support transactions. If used, the ***`TX_ID`*** field is omitted. |
| 1 | Indicates X/Open XA transaction ID (XID). This value is currently not supported. |

Report a bug

### 13.7.4. Client Intelligence Values

The following is a list of valid values for ***`Client Intelligence`*** in a request header:

**Table 13.40. Client Intelligence Field Values**

| Value | Details |
|---|---|
| 0x01 | Indicates a basic client that does not require any cluster or hash information. |
| 0x02 | Indicates a client that is aware of topology and requires cluster information. |
| 0x03 | Indicates a client that is aware of hash and distribution and requires both the cluster and hash information. |

Report a bug

### 13.7.5. Flag Values

The following is a list of valid ***`flag`*** values in the request header:

**Table 13.41. Flag Field Values**

| Value | Details |
|---|---|
| 0x0001 | ForceReturnPreviousValue |

Report a bug

### 13.7.6. Hot Rod Error Handling

**Table 13.42. Hot Rod Error Handling using Response Header Fields**

| Field | Data Type | Details |
|---|---|---|
| Error Opcode | - | Contains the error operation code. |
| Error Status Number | - | Contains a status number that corresponds to the ***`error opcode`***. |
| Error Message Length | vInt | Contains the length of the error message. |
| Error Message | string | Contains the actual error message. If an **`0x84`** error code returns, which indicates that there was an error in parsing the request, this field contains the latest version supported by the **`Hot Rod`** server. |

Report a bug

## 13.8. PUT REQUEST EXAMPLE

The following is the coded request from a sample **put** request using Hot Rod:

**Table 13.43. Put Request Example**

| Byte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 8 | 0xA0 | 0x09 | 0x41 | 0x01 | 0x07 | 0x4D ('M') | 0x79 ('y') | 0x43 ('C') |
| 16 | 0x61 ('a') | 0x63 ('c') | 0x68 ('h') | 0x65 ('e') | 0x00 | 0x03 | 0x00 | 0x00 |
| 24 | 0x00 | 0x05 | 0x48 ('H') | 0x65 ('e') | 0x6C ('l') | 0x6C ('l') | 0x6F ('o') | 0x00 |
| 32 | 0x00 | 0x05 | 0x57 ('W') | 0x6F ('o') | 0x72 ('r') | 0x6C ('l') | 0x64 ('d') | - |

The following table contains all header fields and their values for the example request:

**Table 13.44. Example Request Field Names and Values**

| Field Name | Byte | Value |
|---|---|---|
| Magic | 0 | 0xA0 |
| Version | 2 | 0x41 |
| Cache Name Length | 4 | 0x07 |
| Flag | 12 | 0x00 |
| Topology ID | 14 | 0x00 |
| Transaction ID | 16 | 0x00 |
| Key | 18-22 | 'Hello' |
| Max Idle | 24 | 0x00 |
| Value | 26-30 | 'World' |
| Message ID | 1 | 0x09 |

| Field Name | Byte | Value |
|---|---|---|
| Opcode | 3 | 0x01 |
| Cache Name | 5-11 | 'MyCache' |
| Client Intelligence | 13 | 0x03 |
| Transaction Type | 15 | 0x00 |
| Key Field Length | 17 | 0x05 |
| Lifespan | 23 | 0x00 |
| Value Field Length | 25 | 0x05 |

The following is a coded response for the sample `put` request:

**Table 13.45. Coded Response for the Sample Put Request**

| Byte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 8 | 0xA1 | 0x09 | 0x01 | 0x00 | 0x00 | - | - | - |

The following table contains all header fields and their values for the example response:

**Table 13.46. Example Response Field Names and Values**

| Field Name | Byte | Value |
|---|---|---|
| Magic | 0 | 0xA1 |
| Opcode | 2 | 0x01 |
| Topology Change Marker | 4 | 0x00 |
| Message ID | 1 | 0x09 |
| Status | 3 | 0x00 |

Report a bug

## 13.9. HOT ROD JAVA CLIENT

Hot Rod is a binary, language neutral protocol. A Java client is able to interact with a server via the Hot Rod protocol using the Hot Rod Java Client API.

Report a bug

### 13.9.1. Hot Rod Java Client Download

Use the following steps to download the JBoss Data Grid Hot Rod Java Client:

**Procedure 13.2. Download Hot Rod Java Client**

1. Log into the Customer Portal at https://access.redhat.com.
2. Click the **Downloads** button near the top of the page.
3. In the **`Product Downloads`** page, click **Red Hat JBoss Data Grid**.
4. Select the appropriate JBoss Data Grid version from the **`Version:`** drop down menu.
5. Locate the **`Red Hat JBoss Data Grid ${VERSION} Hot Rod Java Client`** entry and click the corresponding **Download** link.

Report a bug

### 13.9.2. Hot Rod Java Client Configuration

The Hot Rod Java client is configured both programmatically and externally using a configuration file or a properties file. The following example illustrate creation of a client instance using the available Java fluent API:

**Example 13.1. Client Instance Creation**

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
= new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.tcpNoDelay(true)
  .connectionPool()
      .numTestsPerEvictionRun(3)
      .testOnBorrow(false)
      .testOnReturn(false)
      .testWhileIdle(true)
  .addServer()
      .host("localhost")
      .port(11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

**Configuring the Hot Rod Java client using a properties file**

To configure the Hot Rod Java client, edit the **`hotrod-client.properties`** file on the classpath.

The following example shows the possible content of the **`hotrod-client.properties`** file.

**Example 13.2. Configuration**

```
infinispan.client.hotrod.transport_factory =
org.infinispan.client.hotrod.impl.transport.tcp.TcpTransportFactory

infinispan.client.hotrod.server_list = 127.0.0.1:11222
infinispan.client.hotrod.marshaller =
org.infinispan.commons.marshall.jboss.GenericJBossMarshaller

infinispan.client.hotrod.async_executor_factory =
org.infinispan.client.hotrod.impl.async.DefaultAsyncExecutorFactory

infinispan.client.hotrod.default_executor_factory.pool_size = 1

infinispan.client.hotrod.default_executor_factory.queue_size = 10000

infinispan.client.hotrod.hash_function_impl.1 =
org.infinispan.client.hotrod.impl.consistenthash.ConsistentHashV1

infinispan.client.hotrod.tcp_no_delay = true

infinispan.client.hotrod.ping_on_startup = true

infinispan.client.hotrod.request_balancing_strategy =
org.infinispan.client.hotrod.impl.transport.tcp.RoundRobinBalancingStrat
egy

infinispan.client.hotrod.key_size_estimate = 64

infinispan.client.hotrod.value_size_estimate = 512

infinispan.client.hotrod.force_return_values = false

infinispan.client.hotrod.tcp_keep_alive = true

## below is connection pooling config

maxActive=-1

maxTotal = -1

maxIdle = -1

whenExhaustedAction = 1

timeBetweenEvictionRunsMillis=120000

minEvictableIdleTimeMillis=300000

testWhileIdle = true

minIdle = 1
```

**NOTE**

The **TCP KEEPALIVE** configuration is enabled/disabled on the Hot Rod Java client either through a config property as seen in the example (**infinispan.client.hotrod.tcp_keep_alive = true/false** or programmatically through the **org.infinispan.client.hotrod.ConfigurationBuilder.tcpKeepAlive()** method.

Either of the following two constructors must be used in order for the properties file to be consumed by Red Hat JBoss Data Grid:

1. **new RemoteCacheManager(boolean start)**
2. **new RemoteCacheManager()**

Report a bug

### 13.9.3. Hot Rod Java Client Basic API

The following code shows how the client API can be used to store or retrieve information from a Hot Rod server using the Hot Rod Java client. This example assumes that a Hot Rod server has been started bound to the default location, **localhost:11222**.

**Example 13.3. Basic API**

```
//API entry point, by default it connects to localhost:11222
CacheContainer cacheContainer = new RemoteCacheManager();
//obtain a handle to the remote default cache
Cache<String, String> cache = cacheContainer.getCache();
//now add something to the cache and make sure it is there
cache.put("car", "ferrari");]
assert cache.get("car").equals("ferrari");
//remove the data
cache.remove("car");
assert !cache.containsKey("car") : "Value must have been removed!";
```

The **RemoteCacheManager** corresponds to **DefaultCacheManager**, and both implement **CacheContainer**.

This API facilitates migration from local calls to remote calls via Hot Rod. This can be done by switching between **DefaultCacheManager** and **RemoteCacheManager**, which is simplified by the common **CacheContainer** interface.

All keys can be retrieved from the remote cache using the **keySet()** method. If the remote cache is a distributed cache, the server will start a Map/Reduce job to retrieve all keys from clustered nodes and return all keys to the client.

Use this method with caution if there are a large number of keys.

```
Set keys = remoteCache.keySet();
```

Report a bug

### 13.9.4. Hot Rod Java Client Versioned API

To ensure data consistency, Hot Rod stores a version number that uniquely identifies each modification. Using ***getVersioned***, clients can retrieve the value associated with the key as well as the current version.

When using the Hot Rod Java client, a **RemoteCacheManager** provides instances of the **RemoteCache** interface that accesses the named or default cache on the remote cluster. This extends the **Cache** interface to which it adds new methods, including the versioned API.

**Example 13.4. Using Versioned Methods**

```
// To use the versioned API, remote classes are specifically needed
RemoteCacheManager remoteCacheManager = new RemoteCacheManager();
RemoteCache<String, String> cache = remoteCacheManager.getCache();
remoteCache.put("car", "ferrari");
VersionedValue valueBinary = remoteCache.getVersioned("car");
// removal only takes place only if the version has not been changed
// in between. (a new version is associated with 'car' key on each
change)
assert remoteCache.remove("car", valueBinary.getVersion());
assert !cache.containsKey("car");
```

**Example 13.5. Using Replace**

```
remoteCache.put("car", "ferrari");
VersionedValue valueBinary = remoteCache.getVersioned("car");
assert remoteCache.replace("car", "lamborghini",
valueBinary.getVersion());
```

Report a bug

## 13.10. HOT ROD C ++ CLIENT

The Hot Rod C++ client is a new addition to the Hot Rod client family which includes the Hot Rod Java client. It enables C++ runtime applications to connect and interact with Red Hat JBoss Data Grid remote servers.

The Hot Rod C++ client allows applications developed in C++ to read or write data to remote caches. The Hot Rod C++ client supports all three levels of client intelligence and is supported on the following platforms:

- Red Hat Enterprise Linux 5, 64-bit
- Red Hat Enterprise Linux 6, 64-bit

The C++ Hot Rod client is available as a Technology Preview on Windows with Visual Studio 2010.

Report a bug

### 13.10.1. Hot Rod C ++ Client Formats

The Hot Rod C++ client is available in the following two library formats:

- Static library
- Shared/Dynamic library

**Static Library**

The static library is statically linked to an application. This increases the size of the final executable. The application is self-contained and it does not need to ship a separate library.

**Shared/Dynamic Library**

Shared/Dynamic libraries are dynamically linked to an application at runtime. The library is stored in a separate file and can be upgraded separately from the application, without recompiling the application.

**NOTE**

This can only happen if the library's major version is equal to the one against which the application was linked at compile time, indicating that it is binary compatible.

Report a bug

### 13.10.2. Hot Rod C ++ Client Prerequisites

In order to use the Hot Rod C++ Client, the following are needed:

- C++ 03 compiler with support for shared_ptr TR1 (GCC 4.0+, Visual Studio C++ 2010).
- Red Hat JBoss Data Grid Server 6.1.0 or higher version.

Report a bug

### 13.10.3. Hot Rod C ++ Client Download

The Hot Rod C++ client is included in a separate zip file **`jboss-datagrid-<version>-hotrod-cpp-client-<platform>.zip`** under Red Hat JBoss Data Grid binaries on the Red Hat Customer Portal at https://access.redhat.com. Download the appropriate Hot Rod C++ client which applies to your operating system.

Report a bug

### 13.10.4. Hot Rod C ++ Client Configuration

The Hot Rod C++ client interacts with a remote Hot Rod server using the RemoteCache API. To initiate communication with a particular Hot Rod server, configure RemoteCache and choose the specific cache on the Hot Rod server.

Use the ConfigurationBuilder API to configure:

- The initial set of servers to connect to.
- Connection pooling attributes.
- Connection/Socket timeouts and TCP nodelay.

- Hot Rod protocol version.

**Sample C++ main executable file configuration**

The following example shows how to use the ConfigurationBuilder to configure a `RemoteCacheManager` and how to obtain the default remote cache:

**Example 13.6. SimpleMain.cpp**

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include <stdlib.h>
using namespace infinispan::hotrod;
int main(int argc, char** argv) {
    ConfigurationBuilder b;
    b.addServer().host("127.0.0.1").port(11222);
    RemoteCacheManager cm(builder.build());
    RemoteCache<std::string, std::string> cache =
cm.getCache<std::string, std::string>();
    return 0;
}
```

Report a bug

## 13.10.5. Hot Rod C ++ Client API

The RemoteCacheManager is a starting point to obtain a reference to a RemoteCache. The RemoteCache API can interact with a remote Hot Rod server and the specific cache on that server.

Using the RemoteCache reference obtained in the previous example, it is possible to put, get, replace and remove values in a remote cache. It is also possible to perform bulk operations, such as retrieving all of the keys, and clearing the cache.

When a RemoteCacheManager is stopped, all resources in use are released.

**Example 13.7. SimpleMain.cpp**

```
RemoteCache<std::string, std::string> rc = cm.getCache<std::string,
std::string>();
    std::string k1("key13");
    std::string v1("boron");
    // put
    rc.put(k1, v1);
    std::auto_ptr<std::string> rv(rc.get(k1));
    rc.putIfAbsent(k1, v1);
    std::auto_ptr<std::string> rv2(rc.get(k1));
    std::map<HR_SHARED_PTR<std::string>,HR_SHARED_PTR<std::string> > map
= rc.getBulk(0);
    std::cout << "getBulk size" << map.size() << std::endl;
    ..
    .
    cm.stop();
```

–

Report a bug

## 13.11. HOT ROD C# CLIENT

The Hot Rod C# client is a new addition to the list of Hot Rod clients that includes Hot Rod Java and Hot Rod C++ clients. Hot Rod C# client allows .NET runtime applications to connect and interact with Red Hat JBoss Data Grid servers.

The Hot Rod C# client is aware of the cluster topology and hashing scheme, and can access an entry on the server in a single hop similar to the Hot Rod Java and Hot Rod C++ clients.

The Hot Rod C# client is compatible with 32-bit and 64-bit operating systems on which the .NET Framework is supported by Microsoft. The .NET Framework 4.0 is a prerequisite along with the supported operating systems to use the Hot Rod C# client.

**WARNING**

The Hot Rod C# client is a Technology Preview feature and is not supported in JBoss Data Grid 6.3.

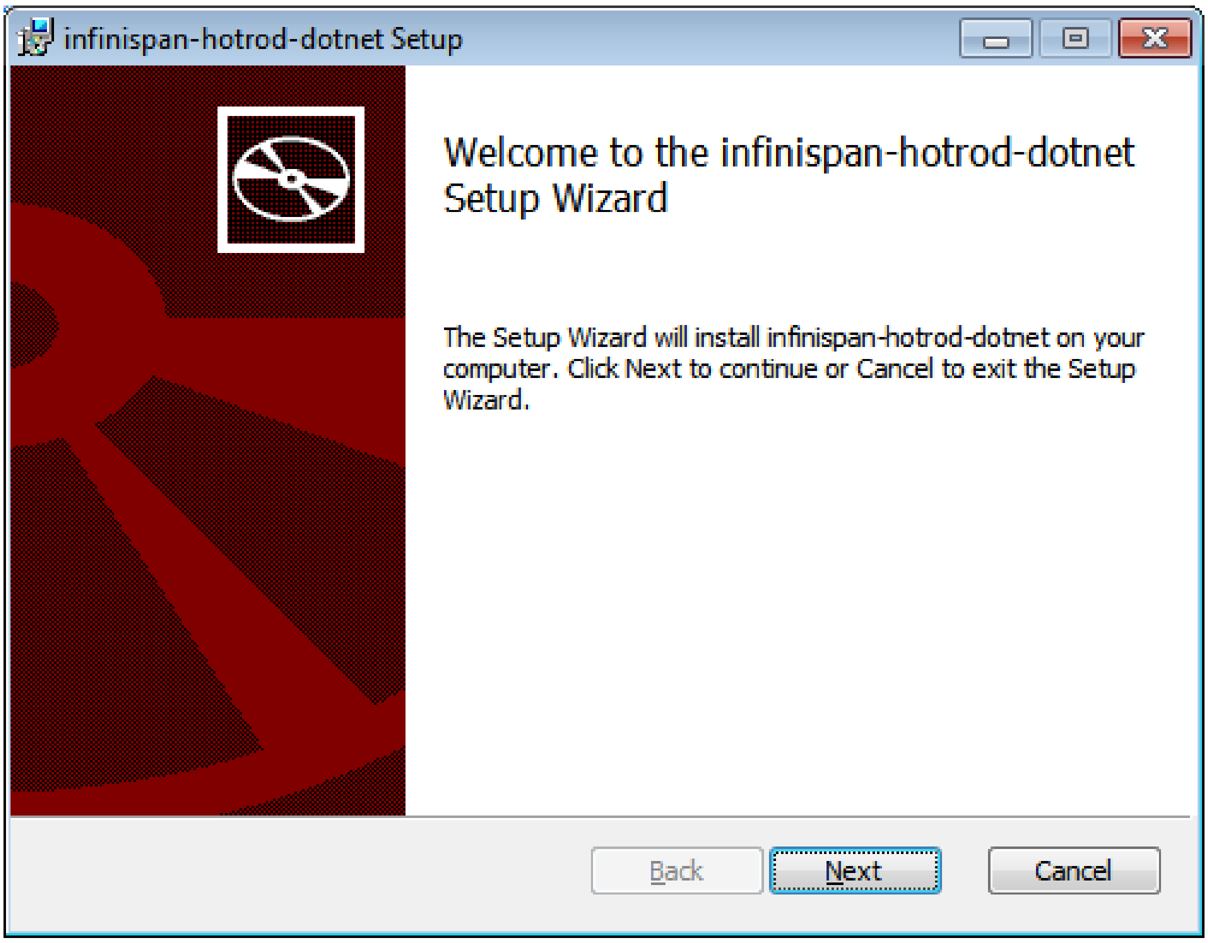
Report a bug

### 13.11.1. Hot Rod C# Client Download and Installation

The Hot Rod C# client is included in a .msi file `jboss-datagrid-<version>-hotrod-dotnet-client.msi` packed for download with Red Hat JBoss Data Grid . To install the Hot Rod C# client, execute the following instructions.

**Procedure 13.3. Installing the Hot Rod C# Client**

1. As an administrator, navigate to the location where the Hot Rod C# .msi file is downloaded. Run the .msi file to launch the windows installer and then click **Next**.

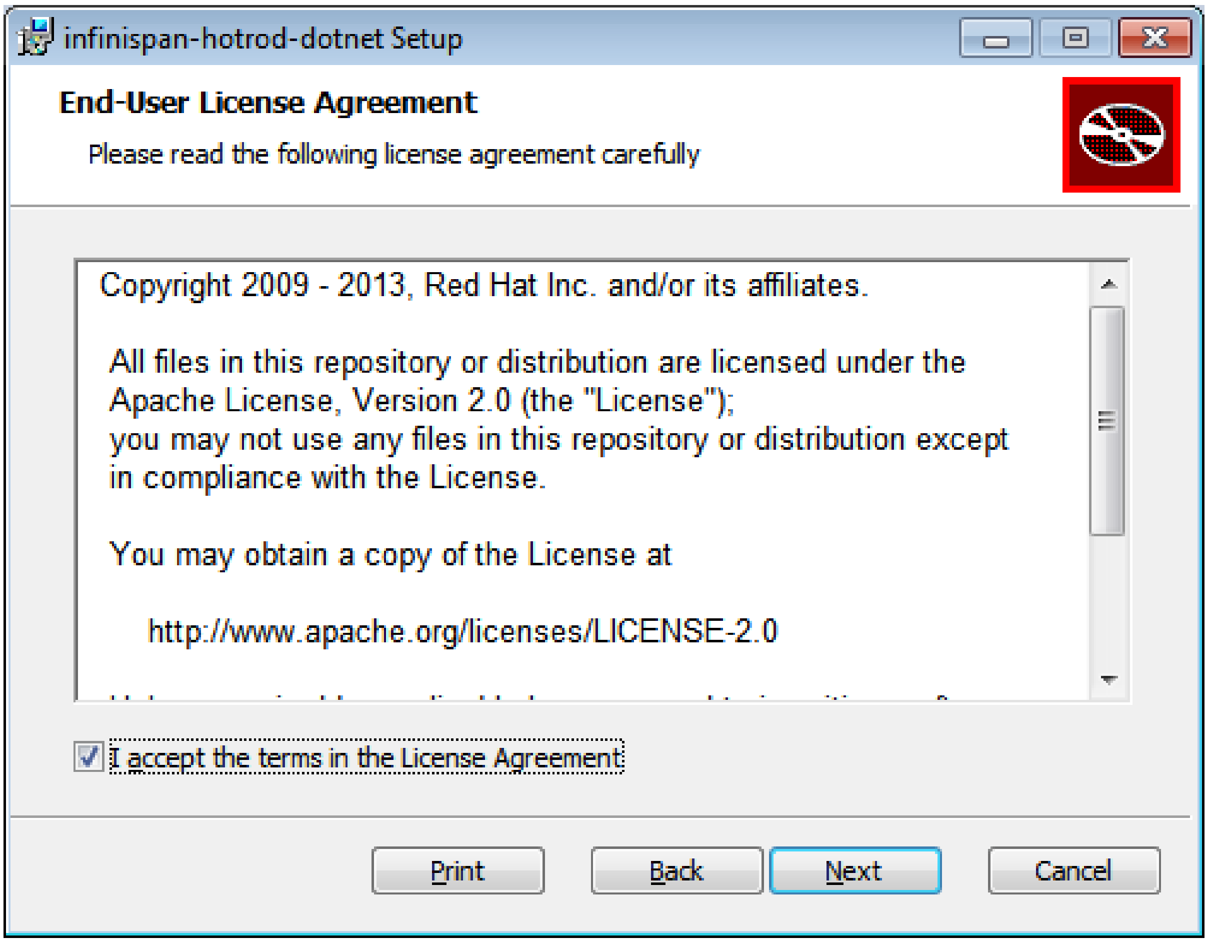

infinispan-hotrod-dotnet Setup

Welcome to the infinispan-hotrod-dotnet Setup Wizard

The Setup Wizard will install infinispan-hotrod-dotnet on your computer. Click Next to continue or Cancel to exit the Setup Wizard.

**Figure 13.1. Hot Rod C# Client Setup Welcome**

2. Review the end-user license agreement. Select the `I accept the terms in the License Agreement` check box and then click **Next**.


infinispan-hotrod-dotnet Setup
End-User License Agreement
Please read the following license agreement carefully

Copyright 2009 - 2013, Red Hat Inc. and/or its affiliates.

All files in this repository or distribution are licensed under the Apache License, Version 2.0 (the "License");
you may not use any files in this repository or distribution except in compliance with the License.

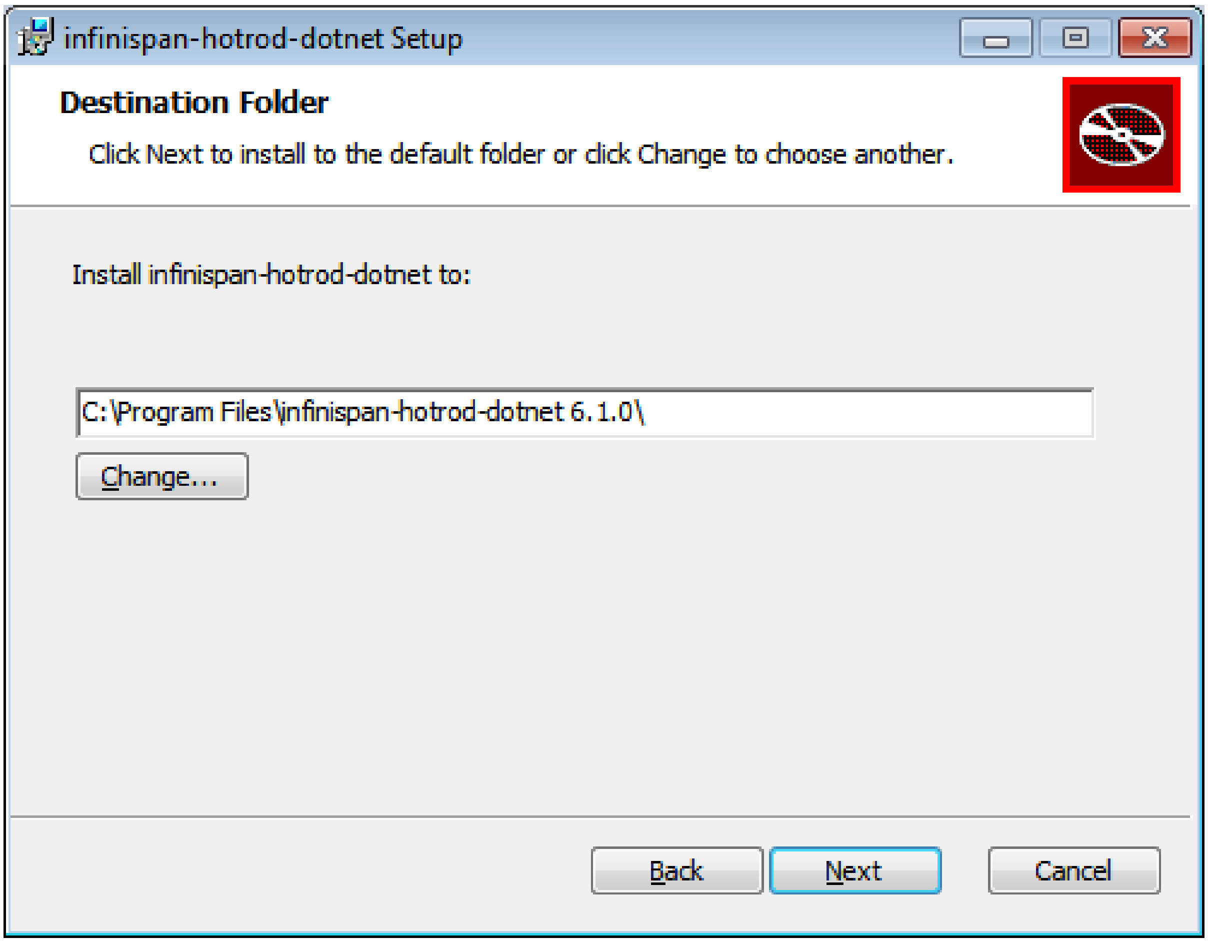
You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

☑ I accept the terms in the License Agreement

**Figure 13.2. Hot Rod C# Client End-User License Agreement**

3. To change the default directory, click **Change...** or click **Next** to install in the default directory.


infinispan-hotrod-dotnet Setup

Destination Folder

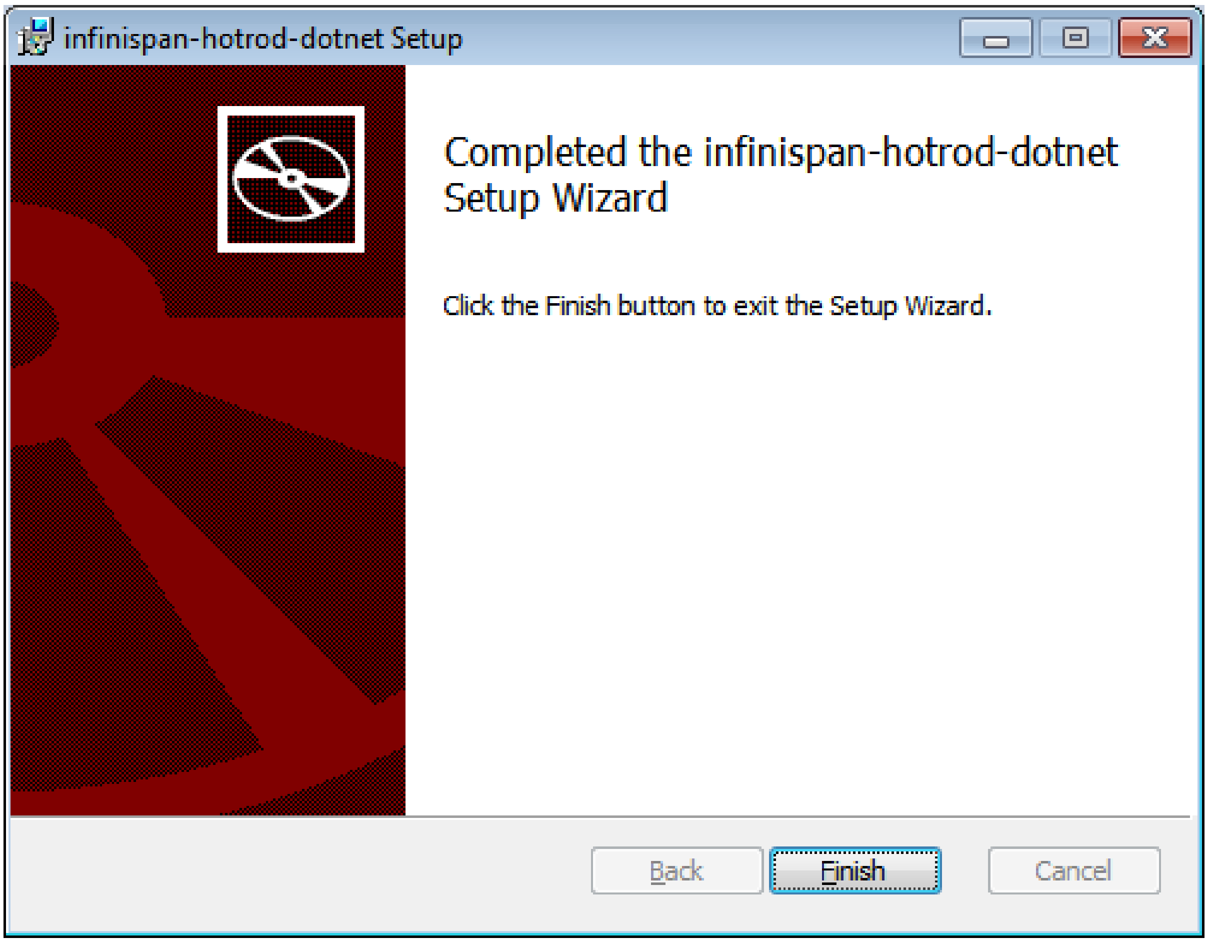
Click Next to install to the default folder or click Change to choose another.

Install infinispan-hotrod-dotnet to:

C:\Program Files\infinispan-hotrod-dotnet 6.1.0\

**Figure 13.3. Hot Rod C# Client Destination Folder**

4. Click **Finish** to complete the Hot Rod C# client installation.



Completed the infinispan-hotrod-dotnet Setup Wizard

Click the Finish button to exit the Setup Wizard.

**Figure 13.4. Hot Rod C# Client Setup Completion**

Report a bug

### 13.11.2. Hot Rod C# Client Configuration

The Hot Rod C# client is configured programmatically using the ConfigurationBuilder. Configure the host and the port to which the client should connect.

#### Sample C# file configuration

The following example shows how to use the ConfigurationBuilder to configure a **RemoteCacheManager**.

**Example 13.8. C# configuration**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;
namespace simpleapp
{
    class Program
    {
        static void Main(string[] args)
        {
            ConfigurationBuilder builder = new ConfigurationBuilder();
            builder.AddServer()
                .Host(args.Length > 1 ? args[0] : "127.0.0.1")
                .Port(args.Length > 2 ? int.Parse(args[1]) : 11222);
            Configuration config = builder.Build();
            RemoteCacheManager cacheManager = new
RemoteCacheManager(config);
            [...]
        }
    }
}
```

Report a bug

### 13.11.3. Hot Rod C# Client API

The **`RemoteCacheManager`** is a starting point to obtain a reference to a RemoteCache.

The following example shows retrieval of a default cache from the server and a few basic operations.

**Example 13.9.**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;
namespace simpleapp
{
    class Program
    {
        static void Main(string[] args)
        {
            ConfigurationBuilder builder = new ConfigurationBuilder();
            builder.AddServer()
                .Host(args.Length > 1 ? args[0] : "127.0.0.1")
                .Port(args.Length > 2 ? int.Parse(args[1]) : 11222);
            Configuration config = builder.Build();
            RemoteCacheManager cacheManager = new
RemoteCacheManager(config);
            cacheManager.Start();
            // Retrieve a reference to the default cache.
            IRemoteCache<String, String> cache =
cacheManager.GetCache<String, String>();
            // Add entries.
            cache.Put("key1", "value1");
            cache.PutIfAbsent("key1", "anotherValue1");
            cache.PutIfAbsent("key2", "value2");
            cache.PutIfAbsent("key3", "value3");
            // Retrive entries.
            Console.WriteLine("key1 -> " + cache.Get("key1"));
            // Bulk retrieve key/value pairs.
```

```
                int limit = 10;
                IDictionary<String, String> result = cache.GetBulk(limit);
                foreach (KeyValuePair<String, String> kv in result)
                {
                    Console.WriteLine(kv.Key + " -> " + kv.Value);
                }
                // Remove entries.
                cache.Remove("key2");
                Console.WriteLine("key2 -> " + cache.Get("key2"));
                cacheManager.Stop();
            }
        }
    }
```

Report a bug

## 13.12. INTEROPERABILITY BETWEEN HOT ROD C++ AND HOT ROD JAVA CLIENT

Red Hat JBoss Data Grid provides interoperability between Hot Rod Java and Hot Rod C++ clients to access structured data. This is made possible by structuring and serializing data using Google's Protobuf format.

For example, using interoperability between languages would allow a Hot Rod C++ client to write the following **Person** object structured and serialized using Protobuf, and the Java C++ client can read the same **Person** object structured as Protobuf.

**Example 13.10. Using Interoperability Between Languages**

```
package sample;
message Person {
    required int32 age = 1;
    required string name = 2;
}
```

Although it is neither enforced or supported, it is recommended that the Hot Rod Java client use the shipped Protostream library for serializing Protobuf objects. Protobuf library is a recommended serialization solution for C++ (https://code.google.com/p/protobuf/).

Interoperability between C++ and Hot Rod Java Client is fully supported for primitive data types, strings, and byte arrays, as Protobuf and Protostream are not required for these types of interoperability.

For information about how the C++ client can read Protobuf encoded data, see https://github.com/jboss-developer/jboss-jdg-quickstarts/tree/master/remote-query/src/main/cpp.

Report a bug

# PART VI. SET UP LOCKING FOR THE CACHE

# CHAPTER 14. LOCKING

Red Hat JBoss Data Grid provides locking mechanisms to prevent dirty reads (where a transaction reads an outdated value before another transaction has applied changes to it) and non-repeatable reads.

Report a bug

## 14.1. CONFIGURE LOCKING (REMOTE CLIENT-SERVER MODE)

In Remote Client-Server mode, locking is configured using the **`locking`** element within the cache tags (for example, **`invalidation-cache`**, **`distributed-cache`**, **`replicated-cache`** or **`local-cache`**).

**NOTE**

The default isolation mode for the Remote Client-Server mode configuration is **READ_COMMITTED**. If the ***`isolation`*** attribute is included to explicitly specify an isolation mode, it is ignored, a warning is thrown, and the default value is used instead.

The following is a sample procedure of a basic locking configuration for a default cache in Red Hat JBoss Data Grid's Remote Client-Server mode.

**Procedure 14.1. Configure Locking (Remote Client-Server Mode)**

1. **Set the *`acquire-timeout`* Parameter**
   The ***`acquire-timeout`*** parameter specifies the number of milliseconds after which lock acquisition will time out.

```
<distributed-cache>
 <locking acquire-timeout="30000" />
```

2. **Set Number of Lock Stripes**
   The ***`concurrency-level`*** parameter defines the number of lock stripes used by the LockManager.

```
<distributed-cache>
 <locking acquire-timeout="30000"
          concurrency-level="1000" />
```

3. **Set Lock Striping**
   The ***`striping`*** parameter specifies whether lock striping will be used for the local cache.

```
<distributed-cache>
 <locking acquire-timeout="30000"
          concurrency-level="1000"
          striping="false" />
          ...
</distributed-cache>
```

Report a bug

## 14.2. CONFIGURE LOCKING (LIBRARY MODE)

For Library mode, the `locking` element and its parameters are set within the optional `configuration` element on a per cache basis. For example, for the default cache, the `configuration` element occurs within the `default` element and for each named cache, it occurs within the **namedCache** element. The following is an example of this configuration:

**Procedure 14.2. Configure Locking (Library Mode)**

1. **Set the Concurrency Level**
   The ***concurrencyLevel*** parameter specifies the concurrency level for the lock container. Set this value according to the number of concurrent threads interacting with the data grid.

   ```
   <infinispan>
    ...
    <default>
     <locking concurrencyLevel="${VALUE}" />
   ```

2. **Specify the Cache Isolation Level**
   The ***isolationLevel*** parameter specifies the cache's isolation level. Valid isolation levels are **READ_COMMITTED** and **REPEATABLE_READ**. For details about isolation levels, see Section 16.1, “About Isolation Levels”

   ```
   <infinispan>
    ...
    <default>
     <locking concurrencyLevel="${VALUE}"
       isolationLevel="${LEVEL}" />
   ```

3. **Set the Lock Acquisition Timeout**
   The ***lockAcquisitionTimeout*** parameter specifies time (in milliseconds) after which a lock acquisition attempt times out.

   ```
   <infinispan>
    ...
    <default>
     <locking concurrencyLevel="${VALUE}"
      isolationLevel="${LEVEL}"
      lockAcquisitionTimeout="${TIME}" />
   ```

4. **Configure Lock Striping**
   The ***useLockStriping*** parameter specifies whether a pool of shared locks are maintained for all entries that require locks. If set to **FALSE**, locks are created for each entry in the cache. For details, see Section 15.1, “About Lock Striping”

   ```
   <infinispan>
    ...
    <default>
     <locking concurrencyLevel="${VALUE}"
      isolationLevel="${LEVEL}"
      lockAcquisitionTimeout="${TIME}"
      useLockStriping="${TRUE/FALSE}" />
   ```

- **Set *writeSkewCheck* Parameter**
  The *writeSkewCheck* parameter is only valid if the *isolationLevel* is set to **REPEATABLE_READ**. If this parameter is set to **FALSE**, a disparity between a working entry and the underlying entry at write time results in the working entry overwriting the underlying entry. If the parameter is set to **TRUE**, such conflicts (namely write skews) throw an exception.

```
<infinispan>
 ...
 <default>
  <locking concurrencyLevel="${VALUE}"
   isolationLevel="${LEVEL}"
   lockAcquisitionTimeout="${TIME}"
   useLockStriping="${TRUE/FALSE}"
   writeSkewCheck="${TRUE/FALSE}" />
```

Report a bug

## 14.3. LOCKING TYPES

### 14.3.1. 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

### 14.3.2. 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

### 14.3.3. Pessimistic Locking Types

Red Hat 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

### 14.3.4. 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:

**Procedure 14.3. Transaction with Explicit Pessimistic Locking**

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

   ```
   tx.begin()
   cache.lock(K)
   ```

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

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

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

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

Report a bug

### 14.3.5. Implicit Pessimistic Locking Example

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

**Procedure 14.4. Transaction with Implicit Pessimistic locking**

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

   ```
   tx.begin()
   cache.put(K,V)
   ```

2. When the line **`cache.put(K2,V2)`** executes, a cluster-wide lock is acquired on **`K2`**.

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

3. 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.

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

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

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

Report a bug

### 14.3.6. Configure Locking Mode (Remote Client-Server Mode)

To configure a locking mode in Red Hat JBoss Data Grid's Remote Client-Server mode, use the ***`transaction`*** element as follows:

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

Report a bug

### 14.3.7. Configure Locking Mode (Library Mode)

In Red Hat JBoss Data Grid's Library mode, the locking mode is set within the **`transaction`** element as follows:

```
<transaction transactionManagerLookupClass="
{TransactionManagerLookupClass}"
      transactionMode="{TRANSACTIONAL,NON_TRANSACTIONAL}"
      lockingMode="{OPTIMISTIC,PESSIMISTIC}"
      useSynchronization="true">
</transaction>
```

Set the ***`lockingMode`*** value to **`OPTIMISTIC`** or **`PESSIMISTIC`** to configure the locking mode used for the transactional cache.

Report a bug

## 14.4. LOCKING OPERATIONS

### 14.4.1. 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:**

- Chapter 15, *Set Up Lock Striping*

Report a bug

### 14.4.2. About Lock Acquisition

Red Hat 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

### 14.4.3. About Concurrency Levels

Concurrency refers to the number of threads simultaneously interacting with the data grid. In Red Hat JBoss Data Grid, concurrency levels refer to the number of concurrent threads used within a lock container.

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

# CHAPTER 15. SET UP LOCK STRIPING

## 15.1. 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 Red Hat JBoss Data Grid. If 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

## 15.2. CONFIGURE LOCK STRIPING (REMOTE CLIENT-SERVER MODE)

Lock striping in Red Hat JBoss Data Grid's Remote Client-Server mode is enabled using the ***striping*** element to **true**.

**Example 15.1. Lock Striping (Remote Client-Server Mode)**

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

**NOTE**

The default isolation mode for the Remote Client-Server mode configuration is **READ_COMMITTED**. If the ***isolation*** attribute is included to explicitly specify an isolation mode, it is ignored, a warning is thrown, and the default value is used instead.

The ***locking*** element uses the following attributes:

- The ***acquire-timeout*** attribute specifies the maximum time to attempt a lock acquisition. The default value for this attribute is **15000** milliseconds.
- The ***concurrency-level*** attribute specifies the concurrency level for lock containers. Adjust this value according to the number of concurrent threads interacting with JBoss Data Grid. The default value for this attribute is **1000**.
- The ***striping*** attribute specifies whether a shared pool of locks is maintained for all entries that require locking (**true**). If set to **false**, a lock is created for each entry. Lock striping controls the memory footprint but can reduce concurrency in the system. The default value for this attribute is **false**.

Report a bug

## 15.3. CONFIGURE LOCK STRIPING (LIBRARY MODE)

Lock striping is disabled by default in Red Hat JBoss Data Grid. Configure lock striping in JBoss Data Grid's Library mode using the ***`useLockStriping`*** parameter as demonstrated in the following procedure.

**Procedure 15.1. Configure Lock Striping (Library Mode)**

1. **Set the Concurrency Level**
   The ***`concurrencyLevel`*** is used to specify the size of the shared lock collection use when lock striping is enabled.

   ```
   <infinispan>
    ...
    <default>

     <locking concurrencyLevel="${VALUE}" />
   ```

2. **Set Isolation Level**
   The ***`isolationLevel`*** parameter specifies the cache's isolation level. Valid isolation levels are **`READ_COMMITTED`** and **`REPEATABLE_READ`**.

   ```
   <infinispan>
    ...
    <default>

     <locking concurrencyLevel="${VALUE}"
       isolationLevel="${LEVEL}"/>
   ```

3. **Specify Lock Acquisition Timeout**
   The ***`lockAcquisitionTimeout`*** parameter specifies time (in milliseconds) after which a lock acquisition attempt times out.

   ```
   <infinispan>
    ...
    <default>

     <locking concurrencyLevel="${VALUE}"
       isolationLevel="${LEVEL}"
       lockAcquisitionTimeout="${TIME}"/>
   ```

4. **Configure Lock Striping**
   The ***`useLockStriping`*** parameter specifies whether a pool of shared locks are maintained for all entries that require locks. If set to **FALSE**, locks are created for each entry in the cache. If set to **TRUE**, lock striping is enabled and shared locks are used as required from the pool.

   ```
   <infinispan>
    ...
    <default>

     <locking concurrencyLevel="${VALUE}"
       isolationLevel="${LEVEL}"
       lockAcquisitionTimeout="${TIME}"
       useLockStriping="${TRUE/FALSE}"/>
   ```

5. **Configure Write Skew Check**
   The ***writeSkewCheck*** check determines if a modification to the entry from a different transaction should roll back the transaction. Write skew set to true requires ***isolation_level*** set to **REPEATABLE_READ**. The default value for ***writeSkewCheck*** and ***isolation_level*** are **FALSE** and **READ_COMMITTED** respectively.

```
<infinispan>
 ...
 <default>

  <locking concurrencyLevel="${VALUE}"
    isolationLevel="${LEVEL}"
    lockAcquisitionTimeout="${TIME}"
    useLockStriping="${TRUE/FALSE}"
    writeSkewCheck="${TRUE/FALSE}" />
  ...

 </default>
</infinispan>
```

Report a bug

# CHAPTER 16. SET UP ISOLATION LEVELS

## 16.1. 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 Red Hat JBoss Data Grid.

- **READ_COMMITTED**. This isolation level is applicable to a wide variety of requirements. This is the default value in Remote Client-Server and Library modes.
- **REPEATABLE_READ**.

**IMPORTANT**

The only valid value for locks in Remote Client-Server mode is the default **READ_COMMITTED** value. The value explicitly specified with the ***isolation*** value is ignored.

If the **locking** element is not present in the configuration, the default isolation value is **READ_COMMITTED**.

For isolation mode configuration examples in JBoss Data Grid, see the lock striping configuration samples:

- See Section 15.2, “Configure Lock Striping (Remote Client-Server Mode)” for a Remote Client-Server mode configuration sample.
- See Section 15.3, “Configure Lock Striping (Library Mode)” for a Library mode configuration sample.

Report a bug

## 16.2. ABOUT READ_COMMITTED

***READ_COMMITTED*** is one of two isolation modes available in Red Hat 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

## 16.3. ABOUT REPEATABLE_READ

***REPEATABLE_READ*** is one of two isolation modes available in Red Hat 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 an entry before a modification occurs. As a result, the "non-repeatable read" phenomenon is avoided because a second read operation on the same entry 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

# PART VII. SET UP AND CONFIGURE A CACHE STORE

# CHAPTER 17. CACHE STORES

The cache store connects Red Hat JBoss Data Grid to the persistent data store. Cache stores are associated with individual caches. Different caches attached to the same cache manager can have different cache store configurations.

Report a bug

## 17.1. CACHE LOADERS AND CACHE WRITERS

Integration with the persistent store is done through the following SPIs located in **`org.infinispan.persistence.spi`**:

- **`CacheLoader`**
- **`CacheWriter`**
- **`AdvancedCacheLoader`**
- **`AdvancedCacheWriter`**

**`CacheLoader`** and **`CacheWriter`** provide basic methods for reading and writing to a store. **`CacheLoader`** retrieves data from a data store when the required data is not present in the cache.

**`AdvancedCacheLoader`** and **`AdvancedCacheWriter`** provide operations to manipulate the underlaying storage in bulk: parallel iteration and purging of expired entries, clear and size.

**`org.infinispan.persistence.file.SingleFileStore`** can be used as a starting point when writing a custom interface.

**NOTE**

Previously, JBoss Data Grid used the old API (**`CacheLoader`**, extended by **`CacheStore`**), which is also still available.

Report a bug

## 17.2. CACHE STORE CONFIGURATION

### 17.2.1. Configuring the Cache Store

Cache stores can be configured in a chain. Cache read operations checks each cache store in the order configured until a valid non-null element of data has been located. Write operations affect all cache stores unless the ***`ignoreModifications`*** element has been set to **`"true"`** for a specific cache store.

Report a bug

### 17.2.2. Configure the Cache Store using XML (Library Mode)

The following example demonstrates cache store configuration using XML in JBoss Data Grid's Library mode:

```
<persistence passivation="false">
   <singleFile
           shared="false"
           preload="true"
           fetchPersistentState="true"
           ignoreModifications="false"
           purgeOnStartup="false"
           location="${java.io.tmpdir}" >
      <async
           enabled="true"
           flushLockTimeout="15000"
           threadPoolSize="5" />
      <singleton
           enabled="true"
           pushStateWhenCoordinator="true"
           pushStateTimeout="20000" />
   </singleFile>
</persistence>
```

For details about the elements and parameters used in this sample configuration, see Section 18.2, "Cache Store Configuration Details (Library Mode)".

Report a bug

### 17.2.3. Configure the Cache Store Programmatically

The following example demonstrates how to configure the cache store programmatically:

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
    .passivation(false)
    .addSingleFileStore()
        .shared(false)
        .preload(true)
        .fetchPersistentState(true)
        .ignoreModifications(false)
        .purgeOnStartup(false)
        .location(System.getProperty("java.io.tmpdir"))
        .async()
           .enabled(true)
           .flushLockTimeout(15000)
           .threadPoolSize(5)
        .singleton()
           .enabled(true)
           .pushStateWhenCoordinator(true)
           .pushStateTimeout(20000);
```

**Procedure 17.1. Configure the Cache store Programatically**

1. **Create a New Configuration Builder**
   Use the ***`ConfigurationBuilder`*** to create a new configuration object.

2. **Set Passivation**

***passivation*** affects the way Red Hat JBoss Data Grid interacts with stores. Passivation removes an object from in-memory cache and writes it to a secondary data store, such as a system or database. Passivation is **false** by default.

3. **Configure the Cache Store**
   ***addSingleFileStore()*** adds the SingleFileStore as the cache store for this configuration. It is possible to create other stores, such as a JDBC Cache Store, which can be added using the ***addStore*** method.

4. **Set Up Sharing**
   ***shared*** indicates that the cache store is shared by different cache instances. For example, where all instances in a cluster use the same JDBC settings to talk to the same remote, shared database. ***shared*** is **false** by default. When set to **true**, it prevents duplicate data being written to the cache store by different cache instances.

5. **Set Up Preloading**
   ***preload*** is set to **false** by default. When set to **true** the data stored in the cache store is preloaded into the memory when the cache starts. This allows data in the cache store to be available immediately after startup and avoids cache operations delays as a result of loading data lazily. Preloaded data is only stored locally on the node, and there is no replication or distribution of the preloaded data. JBoss Data Grid will only preload up to the maximum configured number of entries in eviction.

6. **Set Up Persistence**
   ***fetchPersistentState*** determines whether or not to fetch the persistent state of a cache and apply it to the local cache store when joining the cluster. If the cache store is shared the fetch persistent state is ignored, as caches access the same cache store. A configuration exception will be thrown when starting the cache service if more than one cache store has this property set to **true**. The ***fetchPersistentState*** property is **false** by default.

7. **Set Modifications**
   ***ignoreModifications*** determines whether write methods are pushed to the specific cache store by allowing write operations to the local file cache store, but not the shared cache store. In some cases, transient application data should only reside in a file-based cache store on the same server as the in-memory cache. For example, this would apply with a further JDBC based cache store used by all servers in the network. ***ignoreModifications*** is **false** by default.

8. **Set Up Purging**
   ***purgeOnStartup*** controls whether cache store is purged when it starts up.

9. **Set Up Location**
   ***location*** configuration element sets a location on disk where the store can write.

10. **Asynchronous Settings**
    These attributes configure aspects specific to each cache store. For example, the ***location*** attribute points to where the SingleFileStore will keep files containing data. Other stores may require more complex configuration.

11. **Configure Singletons**
    ***singleton*** enables modifications to be stored by only one node in the cluster. This node is called the coordinator. The coordinator pushes the caches in-memory states to disk. This function is activated by setting the ***enabled*** attribute to **true** in all nodes. The ***shared*** parameter cannot be defined with ***singleton*** enabled at the same time. The ***enabled*** attribute is **false** by default.

12. **Set Up Push States**
    ***pushStateWhenCoordinator*** is set to `true` by default. If `true`, this property will cause a node that has become the coordinator to transfer in-memory state to the underlying cache store. This parameter is useful where the coordinator has crashed and a new coordinator is elected.

Report a bug

### 17.2.4. About SKIP_CACHE_LOAD Flag

In Red Hat JBoss Data Grid's Remote Client-Server mode, when the cache is preloaded from a cache store and eviction is disabled, read requests go to the memory. If the entry is not found in a memory during a read request, it accesses the cache store which may impact the read performance.

To avoid referring to the cache store when a key is not found in the memory, use the **`SKIP_CACHE_LOAD`** flag.

Report a bug

## 17.3. SHARED CACHE STORES

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

A cache store 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 store prevents the unnecessary repeated write operations that occur when various cache instances attempt to write the same data to the cache store.

Report a bug

### 17.3.1. Invalidation Mode and Shared Cache Stores

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

The benefits of using invalidation mode in conjunction with shared cache stores 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 store lazily and only when required to do so, resulting in further reduced network traffic.

Report a bug

### 17.3.2. The Cache Store and Cache Passivation

In Red Hat JBoss Data Grid, a cache store 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 store 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 store steps in to persist the changes in the store.

Report a bug

### 17.3.3. Application Cachestore Registration

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

Report a bug

## 17.4. CONNECTION FACTORIES

In Red Hat JBoss Data Grid, all JDBC cache stores 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

### 17.4.1. 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

### 17.4.2. 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 18. CACHE STORE IMPLEMENTATIONS

The cache store connects Red Hat JBoss Data Grid to the persistent data store. Cache stores are associated with individual caches. Different caches attached to the same cache manager can have different cache store configurations.

Report a bug

## 18.1. CACHE STORE COMPARISON

Select a cache store based on your requirements. The following is a summary of high level differences between the cache stores available in Red Hat JBoss Data Grid:

- The Single File Cache Store is a local file cache store. It persists data locally for each node of the clustered cache. The Single File Cache Store provides superior read and write performance, but keeps keys in memory which limits its use when persisting large data sets at each node. See Section 18.4, “Single File Cache Store” for details.
- The LevelDB file cache store is a local file cache store which provides high read and write performance. It does not have the limitation of Single File Cache Store of keeping keys in memory. See Section 18.5, “LevelDB Cache Store” for details.
- The JDBC cache store is a cache store that may be shared, if required. When using it, all nodes of a clustered cache persist to a single database or a local JDBC database for every node in the cluster. The shared cache store lacks the scalability and performance of a local cache store such as the LevelDB cache store, but it provides a single location for persisted data. The JDBC cache store persists entries as binary blobs, which are not readable outside JBoss Data Grid. See Section 18.6, “JDBC Based Cache Stores” for details.
- The JPA Cache Store (supported in Library mode only) is a shared cache store like JDBC cache store, but preserves schema information when persisting to the database. Therefore, the persisted entries can be read outside JBoss Data Grid. See Section 18.8, “JPA Cache Store” for details.

Report a bug

## 18.2. CACHE STORE CONFIGURATION DETAILS (LIBRARY MODE)

The following tables contain details about the configuration elements and parameters for cache store elements in JBoss Data Grid's Library mode:

**The namedCache Element**

- Add the name value to the ***name*** parameter to set the name of the cache store.

**The persistence Element**

- The ***passivation*** parameter affects the way in which Red Hat JBoss Data Grid interacts with stores. Passivation removes an object from in-memory cache and writes it to a secondary data store, such as a system or database. Valid values for this parameter are `true` and `false` but ***passivation*** is set to `false` by default.

**The singleFile Element**

- The ***shared*** parameter indicates that the cache store is shared by different cache instances.

For example, where all instances in a cluster use the same JDBC settings to talk to the same remote, shared database. ***`shared`*** is **`false`** by default. When set to **`true`**, it prevents duplicate data being written to the cache store by different cache instances. For the LevelDB cache stores, this parameter must be excluded from the configuration, or set to **`false`** because sharing this cache store is not supported.

- The ***`preload`*** parameter is set to **`false`** by default. When set to **`true`** the data stored in the cache store is preloaded into the memory when the cache starts. This allows data in the cache store to be available immediately after startup and avoids cache operations delays as a result of loading data lazily. Preloaded data is only stored locally on the node, and there is no replication or distribution of the preloaded data. Red Hat JBoss Data Grid will only preload up to the maximum configured number of entries in eviction.

- The ***`fetchPersistentState`*** parameter determines whether or not to fetch the persistent state of a cache and apply it to the local cache store when joining the cluster. If the cache store is shared the fetch persistent state is ignored, as caches access the same cache store. A configuration exception will be thrown when starting the cache service if more than one cache store has this property set to **`true`**. The ***`fetchPersistentState`*** property is **`false`** by default.

- The ***`ignoreModifications`*** parameter determines whether write methods are pushed to the specific cache store by allowing write operations to the local file cache store, but not the shared cache store. In some cases, transient application data should only reside in a file-based cache store on the same server as the in-memory cache. For example, this would apply with a further JDBC based cache store used by all servers in the network. ***`ignoreModifications`*** is **`false`** by default.

- The ***`maxEntries`*** parameter provides maximum number of entries allowed. The default value is -1 for unlimited entries.

- The ***`maxKeysInMemory`*** parameter is used to speed up data lookup. The single file store keeps an index of keys and their positions in the file using the ***`maxKeysInMemory`*** parameter. The default value for this parameter is -1.

- The ***`purgeOnStartup`*** parameter controls whether cache store is purged when it starts up.

- The ***`location`*** configuration element sets a location on disk where the store can write.

**The async Element**

The **`async`** element contains parameters that configure various aspects of the cache store.

- The ***`enabled`*** parameter determines whether the file store is asynchronous.

- The ***`threadPoolSize`*** parameter specifies the number of threads that concurrently apply modifications to the store. The default value for this parameter is **`5`**.

- The ***`flushLockTimeout`*** parameter specifies the time to acquire the lock which guards the state to be flushed to the cache store periodically. The default value for this parameter is **`1`**.

- The ***`modificationQueueSize`*** parameter specifies the size of the modification queue for the asynchronous store. If updates are made at a rate that is faster than the underlying cache store can process this queue, then the asynchronous store behaves like a synchronous store for that period, blocking until the queue can accept more elements. The default value for this parameter is **`1024`** elements.

- The ***shutdownTimeout*** parameter specifies the time to stop the cache store. Default value for this parameter is **25** seconds.

**The singleton Element**

The ***singleton*** element enables modifications to be stored by only one node in the cluster. This node is called the coordinator. The coordinator pushes the caches in-memory states to disk. The ***shared*** element cannot be defined with ***singleton*** enabled at the same time.

- The ***enabled*** attribute determines whether this feature is enabled. Valid values for this parameter are **true** and **false**. The ***enabled*** attribute is set to **false** by default.

- The ***pushStateWhenCoordinator*** parameter is set to **true** by default. If **true**, this property causes a node that has become the coordinator to transfer in-memory state to the underlying cache store. This parameter is useful where the coordinator has crashed and a new coordinator is elected.

- When ***pushStateWhenCoordinator*** is set to **true**, the ***pushStateTimeout*** parameter sets the maximum number of milliseconds that the process pushing the in-memory state to the underlying cache loader can take. The default time for this parameter is 10 seconds.

**The remoteStore Element**

- The ***remoteCacheName*** attribute specifies the name of the remote cache to which it intends to connect in the remote Infinispan cluster. The default cache will be used if the remote cache name is unspecified.

- The ***fetchPersistentState*** attribute, when set to **true**, ensures that the persistent state is fetched when the remote cache joins the cluster. If multiple cache stores are chained, only one cache store can have this property set to **true** . The default for this value is **false**.

- The ***shared*** attribute is set to **true** when multiple cache instances share a cache store, which prevents multiple cache instances writing the same modification individually. The default for this attribute is **false**.

- The ***preload*** attribute ensures that the cache store data is pre-loaded into memory and is immediately accessible after starting up. The disadvantage of setting this to **true** is that the start up time increases. The default value for this attribute is **false**.

- The ***ignoreModifications*** attribute prevents cache modification operations such as put, remove, clear, store, etc. from affecting the cache store. As a result, the cache store can become out of sync with the cache. The default value for this attribute is **false**.

- The ***purgeOnStartup*** attribute ensures that the cache store is purged during the start up process. The default value for this attribute is **false**.

- The ***tcpNoDelay*** attribute triggers the **TCP** ***NODELAY*** stack. The default value for this attribute is **true**.

- The ***pingOnStartup*** attribute sends a ping request to a back end server to fetch the cluster topology. The default value for this attribute is **true**.

- The ***keySizeEstimate*** attribute specifies the class name of the driver used to connect to the database. The default value for this attribute is **64**.

- The ***valueSizeEstimate*** attribute specifies the size of the byte buffers when serializing and deserializing values. The default value for this attribute is **`512`**.
- The ***forceReturnValues*** attribute sets whether ***FORCE_RETURN_VALUE*** is enabled for all calls. The default value for this attribute is **`false`**.

**The servers and server Elements**

Create a **`servers`** element within the **`remoteStore`** element to set up the server information.for multiple servers. Add a **`server`** element within the general **`servers`** element to add the information for a single server.

- The ***host*** attribute configures the host address.
- The ***port*** attribute configures the port used by the Remote Cache Store.

**The connectionPool Element**

- The ***maxActive*** attribute indicates the maximum number of active connections for each server at a time. The default value for this attribute is **`-1`** which indicates an infinite number of active connections.
- The ***maxIdle*** attribute indicates the maximum number of idle connections for each server at a time. The default value for this attribute is **`-1`** which indicates an infinite number of idle connections.
- The ***maxTotal*** attribute indicates the maximum number of persistent connections within the combined set of servers. The default setting for this attribute is **`-1`** which indicates an infinite number of connections.
- The ***connectionUrl*** parameter specifies the JDBC driver-specific connection URL.
- The ***username*** parameter contains the username used to connect via the ***connectionUrl***.
- The ***driverClass*** parameter specifies the class name of the driver used to connect to the database.

**The leveldbStore Element**

- The ***location*** parameter specifies the location to store the primary cache store. The directory is automatically created if it does not exist.
- The ***expiredLocation*** parameter specifies the location for expired data using. The directory stores expired data before it is purged. The directory is automatically created if it does not exist.
- The ***shared*** parameter specifies whether the cache store is shared. The only supported value for this parameter in the LevelDB cache store is **`false`**.
- The ***preload*** parameter specifies whether the cache store will be pre-loaded. Valid values are **`true`** and **`false`**.

**The jpaStore Element**

- The ***persistenceUnitName*** attribute specifies the name of the JPA cache store.

- The ***entityClassName*** attribute specifies the fully qualified class name of the JPA entity used to store the cache entry value.
- The ***batchSize*** (optional) attribute specifies the batch size for cache store streaming. The default value for this attribute is **100**.
- The ***storeMetadata*** (optional) attribute specifies whether the cache store keeps the metadata (for example expiration and versioning information) with the entries. The default value for this attribute is **true**.

**The binaryKeyedJdbcStore, stringKeyedJdbcStore, and mixedKeyedJdbcStore Elements**

- The ***fetchPersistentState*** parameter determines whether the persistent state is fetched when joining a cluster. Set this to **true** if using a replication and invalidation in a clustered environment. Additionally, if multiple cache stores are chained, only one cache store can have this property enabled. If a shared cache store is used, the cache does not allow a persistent state transfer despite this property being set to **true**. The ***fetchPersistentState*** parameter is **false** by default.
- The **ignoreModifications** parameter determines whether operations that modify the cache (e.g. put, remove, clear, store, etc.) do not affect the cache store. As a result, the cache store can become out of sync with the cache.
- The **purgeOnStartup** parameter specifies whether the cache store is purged when initially started.
- The ***key2StringMapper*** parameter specifies the class name of the Key2StringMapper used to map keys to strings for the database tables.

**The binaryKeyedTable and stringKeyedTable Elements**

- The ***dropOnExit*** parameter specifies whether the database tables are dropped upon shutdown.
- The ***createOnStart*** parameter specifies whether the database tables are created by the store on startup.
- The ***prefix*** parameter defines the string prepended to name of the target cache when composing the name of the cache bucket table.

**The idColumn, dataColumn, and timestampColumn Elements**

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

**The store Element**

- The ***class*** parameter specifies the class name of the cache store implementation.
- The ***preload*** parameter specifies whether to load entries into the cache during start up. Valid values for this parameter are **true** and **false**.
- The ***shared*** parameter specifies whether the cache store is shared. This is used when multiple cache instances share a cache store. Valid values for this parameter are **true** and **false**.

**The property Element**

- The ***name*** parameter specifies the name of the property.
- The ***value*** parameter specifies the value of the property.

Report a bug

## 18.3. CACHE STORE CONFIGURATION DETAILS (REMOTE CLIENT-SERVER MODE)

The following tables contain details about the configuration elements and parameters for cache store elements in JBoss Data Grid's Remote Client-Server mode:

**The local-cache Element**

- The ***name*** parameter of the `local-cache` attribute is used to specify a name for the cache.
- The ***statistics*** parameter specifies whether statistics are enabled at the container level. Enable or disable statistics on a per-cache basis by setting the ***statistics*** attribute to `false`.

**The file-store Element**

- The ***name*** parameter of the `file-store` element is used to specify a name for the file store.
- 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 ***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 `true` and `false`. However, the ***shared*** parameter is not recommended for the LevelDB cache store because this cache store cannot be shared.
- The ***relative-to*** property is the directory where the `file-store` stores the data. It is used to define a named path.
- The ***path*** property is the name of the file where the data is stored. It is a relative path name that is appended to the value of the ***relative-to*** property to determine the complete path.
- The ***maxEntries*** parameter provides maximum number of entries allowed. The default value is -1 for unlimited entries.
- The ***fetch-state*** parameter when set to true fetches the persistent state when joining a cluster. If multiple cache stores are chained, only one of them can have this property enabled. Persistent state transfer with a shared cache store does not make sense, as the same persistent store that provides the data will just end up receiving it. Therefore, if a shared cache store is used, the cache does not allow a persistent state transfer even if a cache store has this property set to `true`. It is recommended to set this property to true only in a clustered environment. The default value for this parameter is false.

- The ***preload*** parameter when set to true, loads the data stored in the cache store into memory when the cache starts. However, setting this parameter to true affects the performance as the startup time is increased. The default value for this parameter is false.
- The ***singleton*** parameter enables a singleton store cache store. SingletonStore is a delegating cache store used when only one instance in a cluster can interact with the underlying store. However, ***singleton*** parameter is not recommended for **`file-store`**.

**The store Element**

- The ***class*** parameter specifies the class name of the cache store implementation.

**The property Element**

- The ***name*** parameter specifies the name of the property.
- The ***value*** parameter specifies the value assigned to the property.

**The remote-store Element**

- The ***cache*** parameter defines the name for the remote cache. If left undefined, the default cache is used instead.
- The ***socket-timeout*** parameter sets whether the value defined in ***SO_TIMEOUT*** (in milliseconds) applies to remote Hot Rod servers on the specified timeout. A timeout value of **`0`** indicates an infinite timeout.
- The ***tcp-no-delay*** sets whether **`TCP_NODELAY`** applies on socket connections to remote Hot Rod servers.
- The ***hotrod-wrapping*** sets whether a wrapper is required for Hot Rod on the remote store.

**The remote-server Element**

- The ***outbound-socket-binding*** parameter sets the outbound socket binding for the remote server.

**The binary-keyed-jdbc-store, string-keyed-jdbc-store, and mixed-keyed-jdbc-store Elements**

- 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 **`true`** and **`false`**.

- The ***`singleton`*** parameter enables a singleton store cache store. SingletonStore is a delegating cache store used when only one instance in a cluster can interact with the underlying store

**The binary-keyed-table and string-keyed-table Elements**

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

**The id-column, data-column, and timestamp-column Elements**

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

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

**The leveldb-store Element**

- The ***`path`*** parameter The directory within the path specified in the ***`relative-to`*** parameter where the cache state is stored. If undefined, the path defaults to the cache container name.

- The ***`passivation`*** parameter specifies whether passivation is enabled for the LevelDB cache store. Valid values are `true` and `false`.

- The ***`purge`*** parameter specifies whether the cache store is purged when it starts up. Valid values are `true` and `false`.

Report a bug

## 18.4. SINGLE FILE CACHE STORE

Red Hat JBoss Data Grid includes one file system based cache store: the **`SingleFileCacheStore`**.

The **`SingleFileCacheStore`** is a simple, file system based implementation and a replacement to the older file system based cache store: the **`FileCacheStore`**.

**`SingleFileCacheStore`** stores all key/value pairs and their corresponding metadata information in a single file. To speed up data location, it also keeps all keys and the positions of their values and metadata in memory. Hence, using the single file cache store slightly increases the memory required, depending on the key size and the amount of keys stored. Hence **`SingleFileCacheStore`** is not recommended for use cases where the keys are too big.

To reduce memory consumption, the size of the cache store can be set to a fixed number of entries to store in the file. However, this works only when Infinispan is used as a cache. When Infinispan used this way, data which is not present in Infinispan can be recomputed or re-retrieved from the authoritative data store and stored in Infinispan cache. The reason for this limitation is because once the maximum number of entries is reached, older data in the cache store is removed, so if Infinispan was used as an authoritative data store, it would lead to data loss which is undesirable in this use case

Due to its limitations, **`SingleFileCacheStore`** can be used in a limited capacity in production environments. It can 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.

Report a bug

### 18.4.1. Single File Store Configuration (Remote Client-Server Mode)

The following is an example of a Single File Store configuration for Red Hat JBoss Data Grid's Remote Client-Server mode:

```
<local-cache name="default" statistics="true">
    <file-store name="myFileStore"
                passivation="true"
                purge="true"
                relative-to="{PATH}"
                path="{DIRECTORY}"
                max-entries="10000"
                fetch-state="true"
                preload="false" />
</local-cache>
```

For details about the elements and parameters used in this sample configuration, see Section 18.3, “Cache Store Configuration Details (Remote Client-Server Mode)”.

Report a bug

### 18.4.2. Single File Store Configuration (Library Mode)

In Red Hat JBoss Grid's Library mode, configure a Single File Cache Store as follows:.

```
<namedCache name="writeThroughToFile">
      <persistence passivation="false">
         <singleFile fetchPersistentState="true"
                     ignoreModifications="false"
                     purgeOnStartup="false"
                     shared="false"
                     preload="false"
                     location="/tmp/Another-FileCacheStore-Location"
                     maxEntries="100"
                     maxKeysInMemory="100">
            <async enabled="true"
                threadPoolSize="500"
                flushLockTimeout="1"
            modificationQueueSize="1024"
            shutdownTimeout="25000"/>
        </singleFile>
      </persistence>
 </namedCache>
```

For details about the elements and parameters used in this sample configuration, see Section 18.2, “Cache Store Configuration Details (Library Mode)”.

Report a bug

### 18.4.3. Migrating data from FileCacheStore to SingleFileCacheStore

Red Hat JBoss Data Grid stores data in a different format than previous versions of JBoss Data Grid. As a result, the newer version of JBoss Data Grid cannot read data stored by older versions. Use rolling upgrades to upgrade persisted data from the format used by the old JBoss Data Grid to the new format. Additionally, the newer version of JBoss Data Grid also stores persistence configuration information in a different location.

Rolling upgrades is the process by which a JBoss Data Grid installation is upgraded without a service shutdown. In Library mode, it refers to a node installation where JBoss Data Grid is running in Library mode. For JBoss Data Grid servers, it refers to the server side components. The upgrade can be due to either hardware or software change such as upgrading JBoss Data Grid.

Rolling upgrades are only available in JBoss Data Grid's Remote Client-Server mode.

Report a bug

## 18.5. LEVELDB CACHE STORE

LevelDB is a key-value storage engine that provides an ordered mapping from string keys to string values.

The LevelDB Cache Store uses two filesystem directories. Each directory is configured for a LevelDB database. One directory stores the non-expired data and the second directory stores the expired keys before a purge.

Report a bug

### 18.5.1. Configuring LevelDB Cache Store (Remote Client-Server Mode)

**Procedure 18.1. To configure LevelDB Cache Store:**

- Add the following elements to a cache definition in `standalone.xml` to configure the database:

  ```
  <leveldb-store path="/path/to/leveldb/data"
      passivation="false"
      purge="false" >
      <expiration path="/path/to/leveldb/expires/data" />
      <implementation type="JNI" />
  </leveldb-store>
  ```

For details about the elements and parameters used in this sample configuration, see Section 18.3, “Cache Store Configuration Details (Remote Client-Server Mode)”.

Report a bug

### 18.5.2. LevelDB Cache Store Programmatic Configuration

The following is a sample programmatic configuration of LevelDB Cache Store:

```
Configuration cacheConfig = new ConfigurationBuilder().persistence()
                .addStore(LevelDBStoreConfigurationBuilder.class)
                .location("/tmp/leveldb/data")
                .expiredLocation("/tmp/leveldb/expired").build();
```

**Procedure 18.2. LevelDB Cache Store programmatic configuration**

1. Use the ***ConfigurationBuilder*** to create a new configuration object.
2. Add the store to the ***LevelDBCacheStoreConfigurationBuilder*** instance to build its configuration.
3. Set the LevelDB Cache Store location path. The specified path stores the primary cache store data. The directory is automatically created if it does not exist.
4. Specify the location for expired data using the ***expiredLocation*** parameter for the LevelDB Store. The specified path stores expired data before it is purged. The directory is automatically created if it does not exist.

**NOTE**

Programmatic configurations can only be used with Red Hat JBoss Data Grid Library mode.

Report a bug

### 18.5.3. LevelDB Cache Store Sample XML Configuration (Library Mode)

The following is a sample XML configuration of LevelDB Cache Store:

```
<namedCache name="vehicleCache">
      <persistence passivation="false">
          <leveldbStore location="/path/to/leveldb/data"
                        expiredLocation="/path/to/expired/data"
                        shared="false"
                        preload="true"/>
      </persistence>
   </namedCache>
```

For details about the elements and parameters used in this sample configuration, see Section 18.2, “Cache Store Configuration Details (Library Mode)”.

Report a bug

### 18.5.4. Configure a LevelDB Cache Store Using JBoss Operations Network

Use the following procedure to set up a new LevelDB cache store using the JBoss Operations Network.

**Procedure 18.3.**

1. Ensure that Red Hat JBoss Operations Network 3.2 or better is installed and started.
2. Install the Red Hat JBoss Data Grid Plugin Pack for JBoss Operations Network 3.2.0.
3. Ensure that JBoss Data Grid is installed and started.
4. Import JBoss Data Grid server into the inventory.
5. Configure the JBoss Data Grid connection settings.
6. Create a new LevelDB cache store as follows:

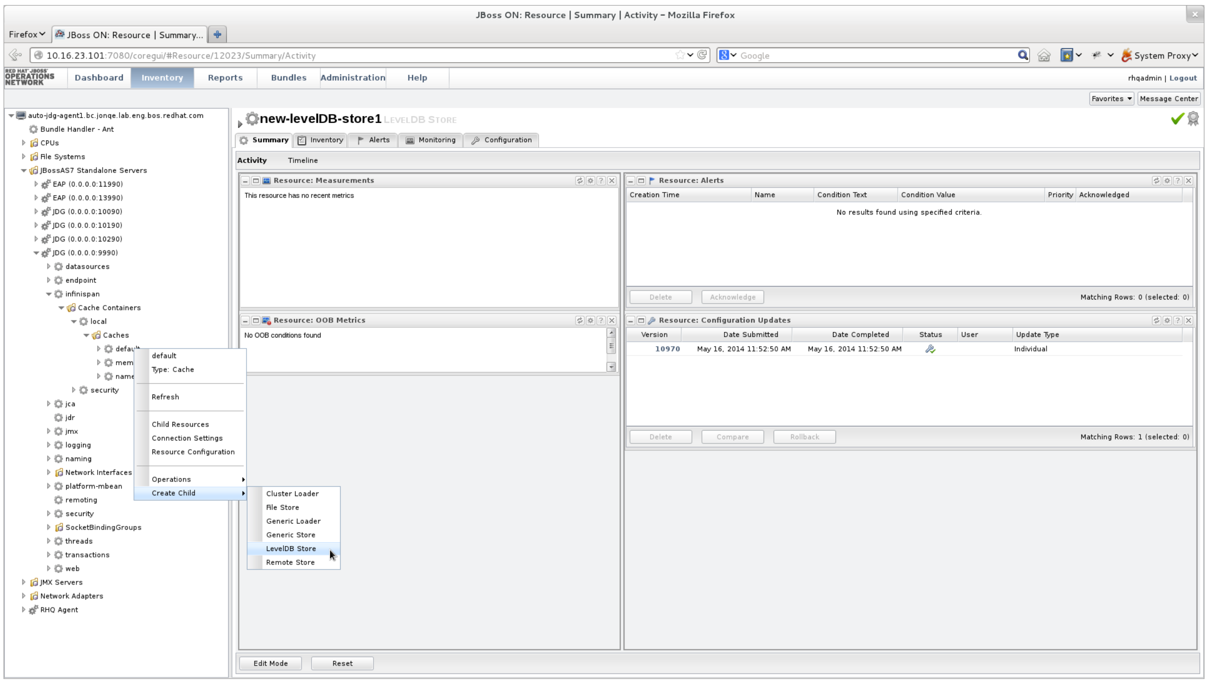


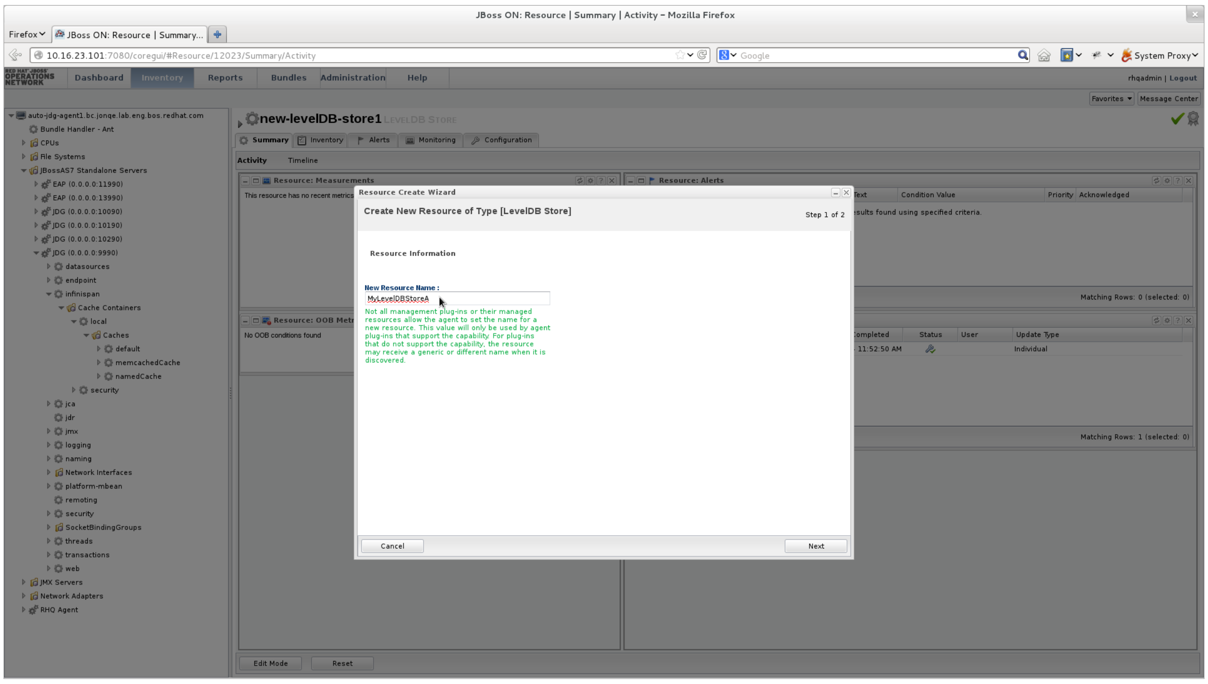
**Figure 18.1. Create a new LevelDB Cache Store**

a. Right-click the **`default`** cache.

b. In the menu, mouse over the **Create Child** option.

c. In the submenu, click **LevelDB Store**.

7. Name the new LevelDB cache store as follows:


Resource Create Wizard

Create New Resource of Type [LevelDB Store] Step 1 of 2

Resource Information

New Resource Name : MyLevelDBStoreA

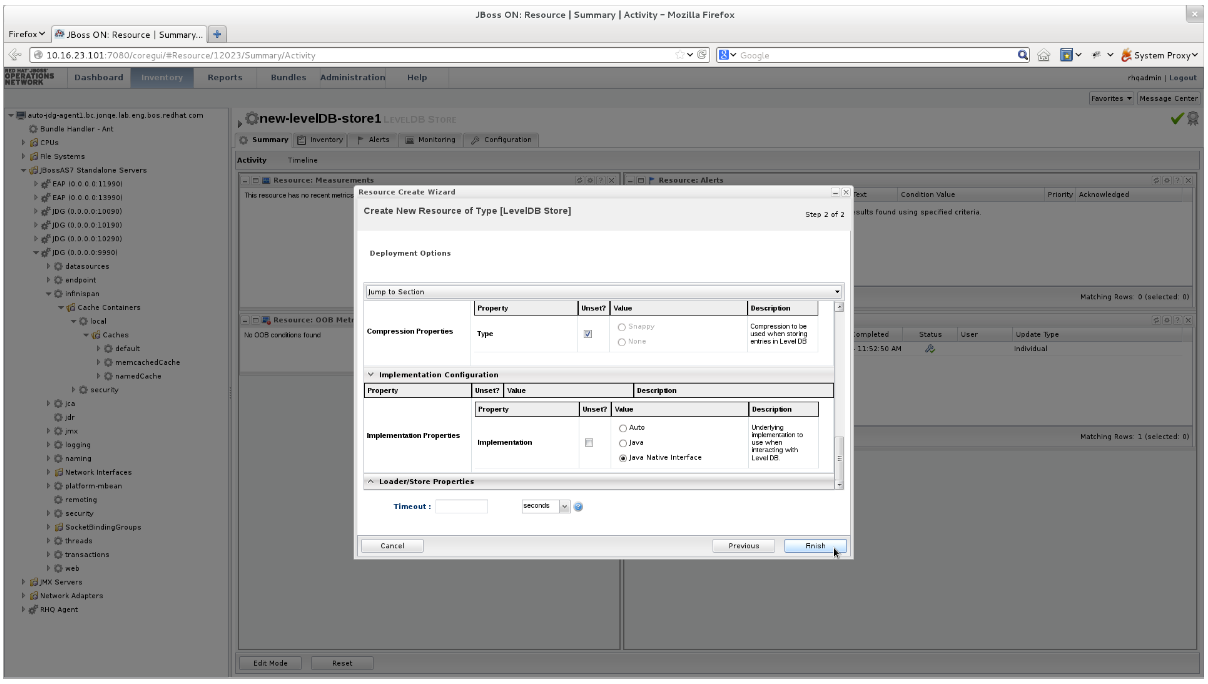
Not all management plug-ins or their managed resources allow the agent to set the name for a new resource. This value will only be used by agent plug-ins that support the capability. For plug-ins that do not support the capability, the resource may receive a generic or different name when it is discovered.

**Figure 18.2. Name the new LevelDB Cache Store**

a. In the **`Resource Create Wizard`** that appears, add a name for the new LevelDB Cache Store.

b. Click **`Next`** to continue.

8. Configure the LevelDB Cache Store settings as follows:



Resource Create Wizard

Create New Resource of Type [LevelDB Store] — Step 2 of 2

Deployment Options

| Compression Properties | Property | Unset? | Value | Description |
|---|---|---|---|---|
| | Type | ✓ | Snappy / None | Compression to be used when storing entries in Level DB |

Implementation Configuration

| Implementation Properties | Property | Unset? | Value | Description |
|---|---|---|---|---|
| | Implementation | | Auto / Java / [✓ Java Native Interface] | Underlying implementation to use when interacting with Level DB. |

Loader/Store Properties

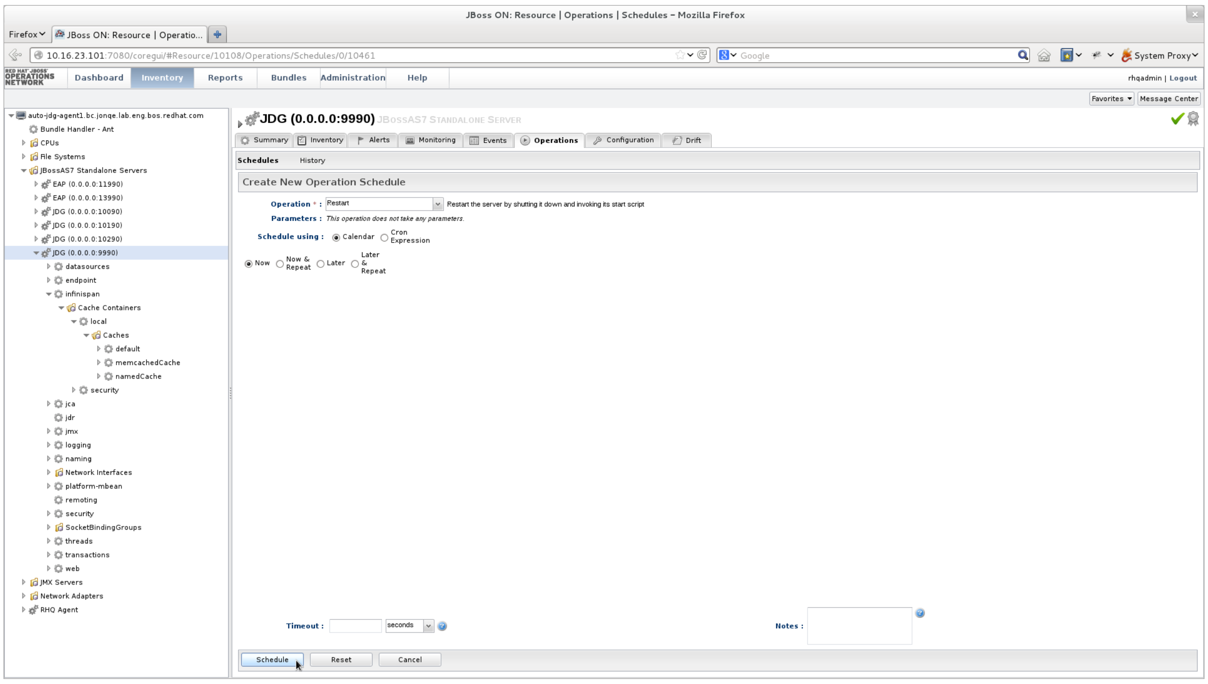
Timeout : seconds

**Figure 18.3. Configure the LevelDB Cache Store Settings**

a. Use the options in the configuration window to configure a new LevelDB cache store.

b. Click **Finish** to complete the configuration.

9. Schedule a restart operation as follows:



JDG (0.0.0.0:9990) JBossAS7 Standalone Server

Create New Operation Schedule

Operation * : Restart — Restart the server by shutting it down and invoking its start script

Parameters : *This operation does not take any parameters.*

Schedule using : [✓ Calendar] Cron Expression

[✓ Now] Now & Repeat Later Later & Repeat

Timeout : seconds

Notes :

**Figure 18.4. Schedule a Restart Operation**

a. In the screen's left panel, expand the `JBossAS7 Standalone Servers` entry, if it is not currently expanded.

b. Click `JDG (0.0.0.0:9990)` from the expanded menu items.

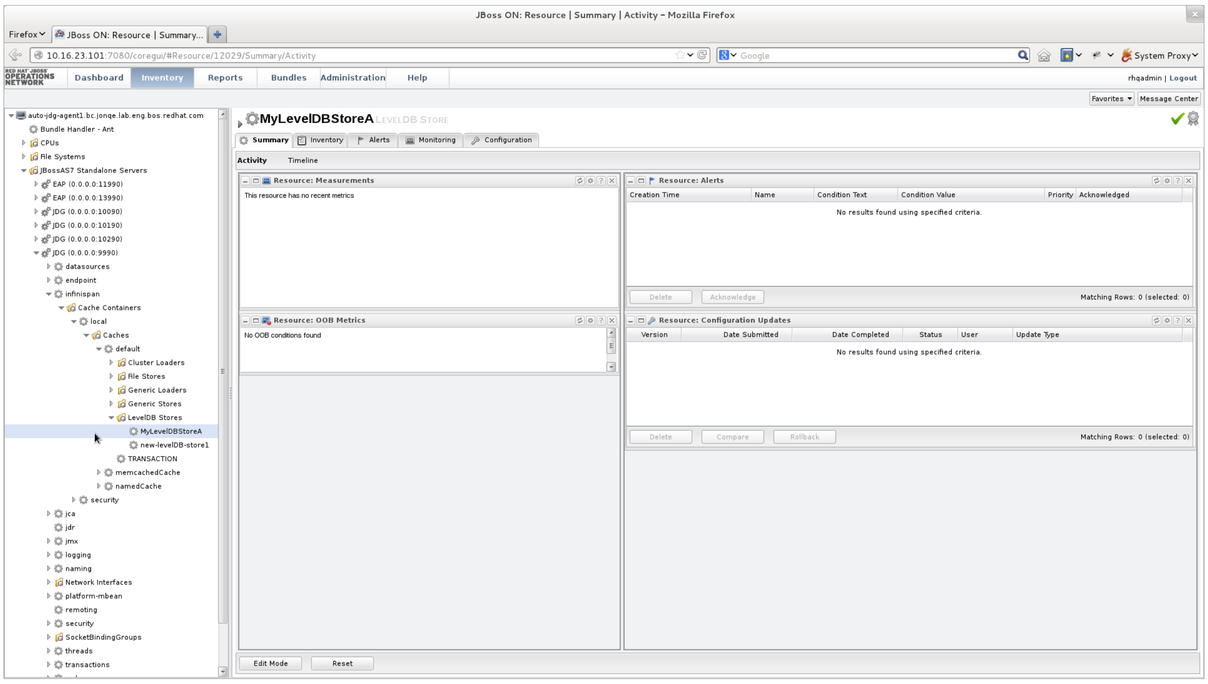
c. In the screen's right panel, details about the selected server display. Click the **Operations** tab.

d. In the `Operation` drop-down box, select the `Restart` operation.

e. Select the radio button for the **Now** entry.

f. Click **Schedule** to restart the server immediately.

10. Discover the new LevelDB cache store as follows:



This resource has no recent metrics

| Creation Time | Name | Condition Text | Condition Value | Priority | Acknowledged |
|---|---|---|---|---|---|

No results found using specified criteria.

Matching Rows: 0 (selected: 0)

No OOB conditions found

| Version | Date Submitted | Date Completed | Status | User | Update Type |
|---|---|---|---|---|---|

No results found using specified criteria.

Matching Rows: 0 (selected: 0)

**Figure 18.5. Discover the New LevelDB Cache Store**

a. In the screen's left panel, select each of the following items in the specified order to expand them: **JBossAS7 Standalong Servers** → **JDG (0.0.0.0:9990)** → **infinispan** → **Cache Containers** → **local** → **Caches** → **default** → **LevelDB Stores**

b. Click the name of your new LevelDB Cache Store to view its configuration information in the right panel.

Report a bug

## 18.6. JDBC BASED CACHE STORES

Red Hat 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:

- `JdbcBinaryStore`.
- `JdbcStringBasedStore`.
- `JdbcMixedStore`.

Report a bug

### 18.6.1. JdbcBinaryStores

The **`JdbcBinaryStore`** 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, **`JdbcBinaryStore`** 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

#### 18.6.1.1. JdbcBinaryStore Configuration (Remote Client-Server Mode)

The following is a configuration for **`JdbcBinaryStore`** using Red Hat JBoss Data Grid's Remote Client-Server mode with Passivation enabled:

```
<local-cache>
 ...
 <binary-keyed-jdbc-store datasource="java:jboss/datasources/JdbcDS"
     passivation="${true/false}"
     preload="${true/false}"
     purge="${true/false}">
                <binary-keyed-table prefix="JDG">
                 <id-column name="id"
       type="${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>
```

For details about the elements and parameters used in this sample configuration, see Section 18.3, “Cache Store Configuration Details (Remote Client-Server Mode)”.

Report a bug

#### 18.6.1.2. JdbcBinaryStore Configuration (Library Mode)

The following is a sample configuration for the **`JdbcBinaryStore`**:

```
<infinispan
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:infinispan:config:6.0
http://www.infinispan.org/schemas/infinispan-config-6.0.xsd
            urn:infinispan:config:jdbc:6.0
http://www.infinispan.org/schemas/infinispan-cachestore-jdbc-config-
6.0.xsd"
        xmlns="urn:infinispan:config:6.0">
        ...
 <persistence>
```

```
  <binaryKeyedJdbcStore xmlns="urn:infinispan:config:jdbc:6.0"
                              fetchPersistentState="false"
          ignoreModifications="false"
          purgeOnStartup="false">
   <connectionPool
connectionUrl="jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1"
    username="sa"
    driverClass="org.h2.Driver"/>
   <binaryKeyedTable dropOnExit="true"
      createOnStart="true"
      prefix="ISPN_BUCKET_TABLE">
    <idColumn name="ID_COLUMN"
      type="VARCHAR(255)" />
    <dataColumn name="DATA_COLUMN"
        type="BINARY" />
    <timestampColumn name="TIMESTAMP_COLUMN"
      type="BIGINT" />
   </binaryKeyedTable>
  </binaryKeyedJdbcStore>
</persistence>
```

For details about the elements and parameters used in this sample configuration, see Section 18.2, “Cache Store Configuration Details (Library Mode)”.

Report a bug

#### 18.6.1.3. JdbcBinaryStore Programmatic Configuration

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

```
ConfigurationBuilder builder = new ConfigurationBuilder();
  builder.persistence()
     .addStore(JdbcBinaryStoreConfigurationBuilder.class)
     .fetchPersistentState(false)
     .ignoreModifications(false)
     .purgeOnStartup(false)
     .table()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_BUCKET_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")

.timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
     .connectionPool()

.connectionUrl("jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1")
        .username("sa")
        .driverClass("org.h2.Driver");
```

**Procedure 18.4. JdbcBinaryStore Programmatic Configuration (Library Mode)**

1. **Create a New Configuration Builder**
   Use the ***ConfigurationBuilder*** to create a new configuration object.

2. **Add the `JdbcBinaryStoreConfigurationBuilder`**
   Add the `JdbcBinaryStore` configuration builder to build a specific configuration related to this store.

3. **Set Up Persistence**
   ***`fetchPersistentState`*** determines whether or not to fetch the persistent state of a cache and apply it to the local cache store when joining the cluster. If the cache store is shared the fetch persistent state is ignored, as caches access the same cache store. A configuration exception will be thrown when starting the cache service if more than one cache loader has this property set to `true`. The ***`fetchPersistentState`*** property is `false` by default.

4. **Set Modifications**
   ***`ignoreModifications`*** determines whether write methods are pushed to the specific cache loader by allowing write operations to the local file cache loader, but not the shared cache loader. In some cases, transient application data should only reside in a file-based cache loader on the same server as the in-memory cache. For example, this would apply with a further JDBC based cache loader used by all servers in the network. ***`ignoreModifications`*** is `false` by default.

5. **Configure Purging**
   ***`purgeOnStartup`*** specifies whether the cache is purged when initially started.

6. **Configure the Table**
   Configure the table as follows:

   a. ***`dropOnExit`*** determines if the table will be dropped when the cache store is stopped. This is set to `false` by default.

   b. ***`createOnStart`*** creates the table when starting the cache store if no table currently exists. This method is `true` by default.

   c. ***`tableNamePrefix`*** sets the prefix for the name of the table in which the data will be stored.

   d. The ***`idColumnName`*** property defines the column where the cache key or bucket ID is stored.

   e. The ***`dataColumnName`*** property specifies the column where the cache entry or bucket is stored.

   f. The ***`timestampColumnName`*** element specifies the column where the time stamp of the cache entry or bucket is stored.

7. **The connectionPool Element**
   The `connectionPool` element specifies a connection pool for the JDBC driver using the following parameters:

   a. The ***`connectionUrl`*** parameter specifies the JDBC driver-specific connection URL.

   b. The ***`username`*** parameter contains the user name used to connect via the ***`connectionUrl`***.

   c. The ***`driverClass`*** parameter specifies the class name of the driver used to connect to the database.

**NOTE**

Programmatic configurations can only be used with Red Hat JBoss Data Grid Library mode.

Report a bug

### 18.6.2. JdbcStringBasedStores

The `JdbcStringBasedStore` stores each entry in 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.

Red Hat JBoss Data Grid includes a default implementation called `DefaultTwoWayKey2StringMapper` that handles primitive types.

Report a bug

#### 18.6.2.1. JdbcStringBasedStore Configuration (Remote Client-Server Mode)

The following is a sample `JdbcStringBasedStore` for Red Hat JBoss Data Grid's Remote Client-Server mode:

```
<local-cache>
 ...
 <string-keyed-jdbc-store datasource="java:jboss/datasources/JdbcDS"
     passivation="true"
     preload="false"
     purge="false"
     shared="false"
     singleton="true">
                <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>
```

For details about the elements and parameters used in this sample configuration, see Section 18.3, “Cache Store Configuration Details (Remote Client-Server Mode)”.

Report a bug

#### 18.6.2.2. JdbcStringBasedStore Configuration (Library Mode)

The following is a sample configuration for the `JdbcStringBasedStore`:

```
<infinispan
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:infinispan:config:6.0
http://www.infinispan.org/schemas/infinispan-config-6.0.xsd
            urn:infinispan:config:jdbc:6.0
http://www.infinispan.org/schemas/infinispan-cachestore-jdbc-config-
6.0.xsd"
        xmlns="urn:infinispan:config:6.0">
        ...
 <persistence>
 <stringKeyedJdbcStore xmlns="urn:infinispan:config:jdbc:6.0"
                      fetchPersistentState="false"
         ignoreModifications="false"
         purgeOnStartup="false"

key2StringMapper="org.infinispan.loaders.keymappers.DefaultTwoWayKey2Strin
gMapper">
  <connectionPool
connectionUrl="jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1"
    username="sa"
    driverClass="org.h2.Driver"/>
  <stringKeyedTable dropOnExit="true"
      createOnStart="true"
      prefix="ISPN_STRING_TABLE">
   <idColumn name="ID_COLUMN"
      type="VARCHAR(255)" />
   <dataColumn name="DATA_COLUMN"
        type="BINARY" />
   <timestampColumn name="TIMESTAMP_COLUMN"
      type="BIGINT" />
  </stringKeyedTable>
 </stringKeyedJdbcStore>
</persistence>
```

For details about the elements and parameters used in this sample configuration, see Section 18.2, “Cache Store Configuration Details (Library Mode)”.

Report a bug

#### 18.6.2.3. JdbcStringBasedStore Multiple Node Configuration (Remote Client-Server Mode)

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

```
<subsystem xmlns="urn:infinispan:server:core:6.1" default-cache-
container="default">
 <cache-container ... >
  ...
      <replicated-cache>
   ...
       <string-keyed-jdbc-store datasource="java:jboss/datasources/JdbcDS"
                      fetch-state="true"
                      passivation="false"
                      preload="false"
                      purge="false"
                      shared="false"
                      singleton="true">
          <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>
```

For details about the elements and parameters used in this sample configuration, see Section 18.3, “Cache Store Configuration Details (Remote Client-Server Mode)”.

Report a bug

### 18.6.2.4. JdbcStringBasedStore Programmatic Configuration

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

```
ConfigurationBuilder builder = new ConfigurationBuilder();

builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.cl
ass)
     .fetchPersistentState(false)
     .ignoreModifications(false)
     .purgeOnStartup(false)
     .table()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_STRING_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")

.timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
     .connectionPool()

.connectionUrl("jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1")
        .username("sa")
        .driverClass("org.h2.Driver");
```

**Procedure 18.5. Configure the JdbcStringBasedStore Programmatically**

1. **Create a New Configuration Builder**
   Use the ***ConfigurationBuilder*** to create a new configuration object.

2. **Add JdbcStringBasedStoreConfigurationBuilder**
   Add the **JdbcStringBasedStore** configuration builder to build a specific configuration related to this store.

3. **Set Up Persistence**
   ***fetchPersistentState*** determines whether or not to fetch the persistent state of a cache and apply it to the local cache store when joining the cluster. If the cache store is shared the fetch persistent state is ignored, as caches access the same cache store. A configuration

exception will be thrown when starting the cache service if more than one cache loader has this property set to `true`. The ***fetchPersistentState*** property is `false` by default.

4. **Set Modifications**
   ***ignoreModifications*** determines whether write methods are pushed to the specific cache loader by allowing write operations to the local file cache loader, but not the shared cache loader. In some cases, transient application data should only reside in a file-based cache loader on the same server as the in-memory cache. For example, this would apply with a further JDBC based cache loader used by all servers in the network. ***ignoreModifications*** is `false` by default.

5. **Configure Purging**
   ***purgeOnStartup*** specifies whether the cache is purged when initially started.

6. **Configure the Table**

   a. ***dropOnExit*** determines if the table will be dropped when the cache store is stopped. This is set to `false` by default.

   b. ***createOnStart*** creates the table when starting the cache store if no table currently exists. This method is `true` by default.

   c. ***tableNamePrefix*** sets the prefix for the name of the table in which the data will be stored.

   d. ***idColumnName***
      The ***idColumnName*** property defines the column where the cache key or bucket ID is stored.

   e. ***dataColumnName***
      The ***dataColumnName*** property specifies the column where the cache entry or bucket is stored.

   f. ***timestampColumnName***
      The ***timestampColumnName*** element specifies the column where the time stamp of the cache entry or bucket is stored.

7. **The connectionPool Element**
   The `connectionPool` element specifies a connection pool for the JDBC driver using the following parameters:

   a. The ***connectionUrl*** parameter specifies the JDBC driver-specific connection URL.

   b. The ***username*** parameter contains the username used to connect via the ***connectionUrl***.

   c. The ***driverClass*** parameter specifies the class name of the driver used to connect to the database.

**NOTE**

Programmatic configurations can only be used with Red Hat JBoss Data Grid Library mode.

Report a bug

### 18.6.3. JdbcMixedStores

The **`JdbcMixedStore`** is a hybrid implementation that delegates keys based on their type to either the **`JdbcBinaryStore`** or **`JdbcStringBasedStore`**.

Report a bug

#### 18.6.3.1. JdbcMixedStore Configuration (Remote Client-Server Mode)

The following is a configuration for a **`JdbcMixedStore`** for Red Hat JBoss Data Grid's Remote Client-Server mode:

```
<local-cache>
 <mixed-keyed-jdbc-store datasource="java:jboss/datasources/JdbcDS"
    passivation="true"
    preload="false"
    purge="false">
  <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>
```

For details about the elements and parameters used in this sample configuration, see Section 18.3, “Cache Store Configuration Details (Remote Client-Server Mode)”.

Report a bug

#### 18.6.3.2. JdbcMixedStore Configuration (Library Mode)

The following is a sample configuration for the **`mixedKeyedJdbcStore`**:

```
<infinispan
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:infinispan:config:6.0
http://www.infinispan.org/schemas/infinispan-config-6.0.xsd
            urn:infinispan:config:jdbc:6.0
http://www.infinispan.org/schemas/infinispan-cachestore-jdbc-config-
6.0.xsd"
        xmlns="urn:infinispan:config:6.0">
        ...
 <persistence>
 <mixedKeyedJdbcStore xmlns="urn:infinispan:config:jdbc:6.0"
                      fetchPersistentState="false"
        ignoreModifications="false"
        purgeOnStartup="false"

key2StringMapper="org.infinispan.persistence.keymappers.DefaultTwoWayKey2S
tringMapper">
  <connectionPool
connectionUrl="jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1"
    username="sa"
    driverClass="org.h2.Driver"/>
  <binaryKeyedTable dropOnExit="true"
      createOnStart="true"
      prefix="ISPN_BUCKET_TABLE_BINARY">
   <idColumn name="ID_COLUMN"
      type="VARCHAR(255)" />
   <dataColumn name="DATA_COLUMN"
        type="BINARY" />
   <timestampColumn name="TIMESTAMP_COLUMN"
      type="BIGINT" />
  </binaryKeyedTable>
  <stringKeyedTable dropOnExit="true"
      createOnStart="true"
      prefix="ISPN_BUCKET_TABLE_STRING">
   <idColumn name="ID_COLUMN"
      type="VARCHAR(255)" />
   <dataColumn name="DATA_COLUMN"
        type="BINARY" />
   <timestampColumn name="TIMESTAMP_COLUMN"
      type="BIGINT" />
  </stringKeyedTable>
 </mixedKeyedJdbcStore>
</persistence>
```

For details about the elements and parameters used in this sample configuration, see Section 18.2, “Cache Store Configuration Details (Library Mode)”.

Report a bug

#### 18.6.3.3. JdbcMixedStore Programmatic Configuration

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

```
ConfigurationBuilder builder = new ConfigurationBuilder();
  builder.persistence().addStore(JdbcMixedStoreConfigurationBuilder.class)
     .fetchPersistentState(false)
     .ignoreModifications(false)
     .purgeOnStartup(false)
     .stringTable()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_MIXED_STR_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
```

```
.timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
     .binaryTable()
        .dropOnExit(true)
        .createOnStart(true)
        .tableNamePrefix("ISPN_MIXED_BINARY_TABLE")
        .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
        .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")

.timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
     .connectionPool()

.connectionUrl("jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1")
        .username("sa")
        .driverClass("org.h2.Driver");
```

**Procedure 18.6. Configure JdbcMixedStore Programmatically**

1. **Create a New Configuration Builder**
   Use the ***`ConfigurationBuilder`*** to create a new configuration object.

2. **Add `JdbcMixedStoreConfigurationBuilder`**
   Add the **`JdbcMixedStore`** configuration builder to build a specific configuration related to this store.

3. **Set Up Persistence**
   ***`fetchPersistentState`*** determines whether or not to fetch the persistent state of a cache and apply it to the local cache store when joining the cluster. If the cache store is shared the state is ignored, as caches access the same cache store. A configuration exception will be thrown when starting the cache service if more than one cache loader has this property set to **`true`**. The ***`fetchPersistentState`*** property is **`false`** by default.

4. **Set Modifications**
   ***`ignoreModifications`*** determines whether write methods are pushed to the specific cache loader by allowing write operations to the local file cache loader, but not the shared cache loader. In some cases, transient application data should only reside in a file-based cache loader on the same server as the in-memory cache. For example, this would apply with a further JDBC based cache loader used by all servers in the network. ***`ignoreModifications`*** is **`false`** by default.

5. **Configure Purging**
   ***`purgeOnStartup`*** specifies whether the cache is purged when initially started.

6. **Configure the Table**
   Configure the table as follows:

   a. ***`dropOnExit`*** determines if the table will be dropped when the cache store is stopped. This is set to **`false`** by default.

   b. ***`createOnStart`*** creates the table when starting the cache store if no table currently exists. This method is **`true`** by default.

   c. ***`tableNamePrefix`*** sets the prefix for the name of the table in which the data will be stored.

   d. The ***idColumnName*** property defines the column where the cache key or bucket ID is stored.

   e. The ***dataColumnName*** property specifies the column where the cache entry or bucket is stored.

   f. The ***timestampColumnName*** element specifies the column where the time stamp of the cache entry or bucket is stored.

7. **The connectionPool Element**
   The **connectionPool** element specifies a connection pool for the JDBC driver using the following parameters:

   a. The ***connectionUrl*** parameter specifies the JDBC driver-specific connection URL.

   b. The ***username*** parameter contains the username used to connect via the ***connectionUrl***.

   c. The ***driverClass*** parameter specifies the class name of the driver used to connect to the database.

**NOTE**

Programmatic configurations can only be used with Red Hat JBoss Data Grid Library mode.

Report a bug

### 18.6.4. Cache Store Troubleshooting

#### 18.6.4.1. IOExceptions with JdbcStringBasedStore

An IOException **Unsupported protocol version 48** error when using **JdbcStringBasedStore** 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, **JdbcStringBasedStore** 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

## 18.7. THE REMOTE CACHE STORE

The **RemoteCacheStore** is an implementation of the cache loader that stores data in a remote Red Hat 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

### 18.7.1. Remote Cache Store Configuration (Remote Client-Server Mode)

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

```
<remote-store cache="default"
              socket-timeout="60000"
              tcp-no-delay="true"
              hotrod-wrapping="true">
 <remote-server outbound-socket-binding="remote-store-hotrod-server" />
</remote-store>
```

For details about the elements and parameters used in this sample configuration, see Section 18.3, “Cache Store Configuration Details (Remote Client-Server Mode)”.

Report a bug

### 18.7.2. Remote Cache Store Configuration (Library Mode)

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

```
<persistence passivation="false">
 <remoteStore xmlns="urn:infinispan:config:remote:6.0"
              remoteCacheName="default"
       fetchPersistentState="false"
       shared="true"
       preload="false"
       ignoreModifications="false"
       purgeOnStartup="false"
       tcpNoDelay="true"
       pingOnStartup="true"
       keySizeEstimate="62"
       valueSizeEstimate="512"
       forceReturnValues="false">
  <servers>
   <server host="127.0.0.1"
    port="19711"/>
  </servers>
  <connectionPool maxActive="99"
    maxIdle="97"
    maxTotal="98" />
 </remoteStore>
</persistence>
```

For details about the elements and parameters used in this sample configuration, see Section 18.2, “Cache Store Configuration Details (Library Mode)”.

Report a bug

### 18.7.3. 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:

**Example 18.1. Define the Outbound Socket**

```
<server>
    ...
    <socket-binding-group name="standard-sockets"
         default-interface="public"
         port-offset="${jboss.socket.binding.port-offset:0}">
        ...
        <outbound-socket-binding name="remote-store-hotrod-server">
            <remote-destination host="remote-host"
                                port="11222"/>
        </outbound-socket-binding>
    </socket-binding-group>
</server>
```

Report a bug

## 18.8. JPA CACHE STORE

The JPA (Java Persistence API) Cache Store stores cache entries in the database using a formal schema, which allows other applications to read the persisted data and load data provided by other applications into Red Hat JBoss Data Grid. The database should not be used by the other applications concurrently with JBoss Data Grid.

**IMPORTANT**

In Red Hat JBoss Data Grid, JPA cache stores are only supported in Library mode.

Report a bug

### 18.8.1. JPA Cache Store Sample XML Configuration (Library Mode)

To configure JPA Cache Stores using XML in Red Hat JBoss Data Grid, add the following configuration to the **`persistence.xml`** file:

```
<persistence passivation="false"
      shared="true"
      preload="true">
 <jpaStore persistenceUnitName="MyPersistenceUnit"
      entityClassName="org.infinispan.loaders.jpa.entity.User" />
</persistence>
```

For details about the elements and parameters used in this sample configuration, see Section 18.2, “Cache Store Configuration Details (Library Mode)”.

Report a bug

### 18.8.2. JPA Cache Store Sample Programmatic Configuration

To configure JPA Cache Stores programatically in Red Hat JBoss Data Grid, use the following:

```
Configuration cacheConfig = new
ConfigurationBuilder().persistence().addStore(JpaStoreConfigurationBuilder
.class).persistenceUnitName("org.infinispan.loaders.jpa.configurationTest"
)
                  .entityClass(User.class)
                  .build();
```

The parameters used in this code sample are as follows:

- The ***persistenceUnitName*** parameter specifies the name of the JPA cache store in the configuration file (**persistence.xml**) that contains the JPA entity class.
- The ***entityClass*** parameter specifies the JPA entity class that is stored in this cache. Only one class can be specified for each configuration.

Report a bug

### 18.8.3. Storing Metadata in the Database

When ***storeMetadata*** is set to **true** (default value), meta information about the entries such as expiration, creation and modification timestamps, and versioning is stored in the database. JBoss Data Grid stores the metadata in an additional table named **__ispn_metadata__** because the entity table has a fixed layout that cannot accommodate the metadata.

The structure of this table depends on the database in use. Enable the automatic creation of this table using the same database as the test environment and then transfer the structure to the production database.

**Procedure 18.7. Configure persistence.xml for Metadata Entities**

1. Using Hibernate as the JPA implementation allows automatic creation of these tables using the property ***hibernate.hbm2ddl.auto*** in **persistence.xml** as follows:

   ```
   <property name="hibernate.hbm2ddl.auto" value="update"/>
   ```

2. Declare the metadata entity class to the JPA provider by adding the following to **persistence.xml**:

   ```
   <class>org.infinispan.persistence.jpa.impl.MetadataEntity</class>
   ```

As outlined, metadata is always stored in a new table. If metadata information collection and storage is not required, set the ***storeMetadata*** attribute to **false** in the JPA Store configuration.

Report a bug

### 18.8.4. Deploying JPA Cache Stores in Various Containers

Red Hat JBoss Data Grid's JPA Cache Store implementations are deployed normally for all supported containers, except Red Hat JBoss Enterprise Application Platform. JBoss Data Grid's JBoss EAP modules contain the JPA cache store and related libraries (such as Hibernate). As a result, the relevant libraries are not packaged inside the application, but instead the application refers to the libraries in the JBoss EAP modules that have them installed.

These modules are not required for containers other than JBoss EAP. As a result, all the relevant libraries are packages in the application's WAR/EAR file.

**Procedure 18.8. Deploy JPA Cache Stores in JBoss EAP**

- To add dependencies from the JBoss Data Grid modules to the application's classpath, provide the JBoss EAP deployer a list of dependencies in one of the following ways:

  a. Add a dependency configuration to the `MANIFEST.MF` file:

  ```
  Manifest-Version: 1.0
  Dependencies: org.infinispan:jdg-6.3 services,
  org.infinispan.persistence.jpa:jdg-6.3 services
  ```

  b. Add a dependency configuration to the `jboss-deployment-structure.xml` file:

  ```
  <jboss-deployment-structure xmlns="urn:jboss:deployment-
  structure:1.2">
      <deployment>
          <dependencies>
              <module name="org.infinispan" slot="jdg-6.3"
  services="export"/>
          </dependencies>
      </deployment>
  </jboss-deployment-structure>
  ```

**IMPORTANT**

JPA Cache Store is not supported in Apache Karaf in JBoss Data Grid 6.3.

Report a bug

## 18.9. CUSTOM CACHE STORES

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

In order to create a custom cache store (or loader), implement all or a subset of the following interfaces based on the need:

- `CacheLoader`
- `CacheWriter`
- `AdvancedCacheLoader`
- `AdvancedCacheWriter`

See Section 17.1, “Cache Loaders and Cache Writers” for individual functions of the interfaces.

**NOTE**

If the `AdvancedCacheWriter` is not implemented, the expired entries cannot be purged or cleared using the given writer.

**NOTE**

If the **AdvancedCacheLoader** is not implemented, the entries stored in the given loader will not be used for preloading and map/reduce iterations.

To migrate the existing cache store to the new API or to write a new store implementation, use **SingleFileStore** as an example. To view the **SingleFileStore** example code, download the JBoss Data Grid source code.

Use the following procedure to download **SingleFileStore** example code from the Customer Portal:

**Procedure 18.9. Download JBoss Data Grid Source Code**

1. To access the Red Hat Customer Portal, navigate to https://access.redhat.com/home in a browser.

2. Click **Downloads**.

3. In the box labeled **Red Hat JBoss Middleware**, click the **Download Software** button.

4. Enter the relevant credentials in the **Red Hat Login** and **Password** fields and click **Log In**.

5. In the **Software Downloads** page, select **Data Grid** from the list of drop down values.

6. From the list of downloadable files, locate **Red Hat JBoss Data Grid ${VERSION} Source Code** and click **Download**. Save and unpack it in a desired location.

7. Locate the **SingleFileStore** source code by navigating through ***jboss-datagrid-6.3.0-sources*infinispan-6.1.0.Final-redhat-4-src/core/src/main/java/org/infinispan/persistence/file/SingleFileStore.java** .

Report a bug

### 18.9.1. Custom Cache Store Configuration (Remote Client-Server Mode)

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

**Example 18.2. Custom Cache Store Configuration**

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

For details about the elements and parameters used in this sample configuration, see Section 18.3, "Cache Store Configuration Details (Remote Client-Server Mode)".

**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

### 18.9.2. Custom Cache Store Configuration (Library Mode)

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

**Example 18.3. Custom Cache Store Configuration**

```
<persistence>
 <store class="org.infinispan.custom.CustomCacheStore"
        preload="true"
        shared="true">
  <properties>
   <property name="customStoreProperty"
      value="10" />
  </properties>
 </store>
</persistence>
```

For details about the elements and parameters used in this sample configuration, see Section 18.2, “Cache Store Configuration Details (Library Mode)”.

Report a bug

# PART VIII. SET UP PASSIVATION

# CHAPTER 19. ACTIVATION AND PASSIVATION MODES

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).

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.

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

## 19.1. 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

## 19.2. CONFIGURE PASSIVATION

In Red Hat JBoss Data Grid's Remote Client-Server mode, add the ***passivation*** parameter to the cache store element to toggle passivation for it:

**Example 19.1. Toggle Passivation in Remote Client-Server Mode**

```
<local-cache>
 ...
 <file-store passivation="true"
      ... />
 ...
</local-cache>
```

In Library mode, add the ***passivation*** parameter to the `persistence` element to toggle passivation:

**Example 19.2. Toggle Passivation in Library Mode**

```
<persistence passivation="true" ... >
   ...
</persistence>
```

Report a bug

## 19.3. 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

### 19.3.1. 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

### 19.3.2. 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 19.1. 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

### 19.3.3. 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 19.2. Eviction when Passivation is Enabled**

| Step | Key in Memory | Key on Disk |
|---|---|---|
| Insert `keyOne` | Memory: `keyOne` | Disk: |
| Insert `keyTwo` | Memory: `keyOne`, `keyTwo` | 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

# PART IX. SET UP CACHE WRITING

# CHAPTER 20. CACHE WRITING MODES

Red Hat 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

## 20.1. WRITE-THROUGH CACHING

The Write-Through (or Synchronous) mode in Red Hat 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

### 20.1.1. 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

### 20.1.2. 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 procedure demonstrates a sample configuration file of a Write-Through unshared local file cache store.

**Procedure 20.1. Configure a Write-Through Local File Cache Store**

1. **Identify the namedCache**
   The ***name*** parameter specifies the name of the **namedCache** to use.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns="urn:infinispan:config:6.1">
    <global />
    <default />
    <namedCache name="persistentCache">
   ```

2. **Configure the Cache Loader**
   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 ***true*** and ***false***.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:infinispan:config:6.1">
 <global />
 <default />
 <namedCache name="persistentCache">
  <persistence shared="false">
```

3. **Specify the Loader Class**
   The ***class*** attribute defines the class of the cache loader implementation.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:infinispan:config:6.1">
 <global />
 <default />
 <namedCache name="persistentCache">
  <persistence shared="false">
  <store class="org.infinispan.loaders.file.FileCacheStore"
```

4. **Configure the *fetchPersistentState* Parameter**
   The ***fetchPersistentState*** parameter determines whether the persistent state is fetched when joining a cluster. Set this to **true** if using a replication and invalidation in a clustered environment. Additionally, if multiple cache stores are chained, only one cache store can have this property enabled. If a shared cache store is used, the cache does not allow a persistent state transfer despite this property being set to **true**. The ***fetchPersistentState*** parameter is **false** by default.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:infinispan:config:6.1">
 <global />
 <default />
 <namedCache name="persistentCache">
  <persistence shared="false">
  <store class="org.infinispan.loaders.file.FileCacheStore"
           fetchPersistentState="true"
```

5. **Set the *ignoreModifications* Parameter**
   The ***ignoreModifications*** parameter determines whether operations that modify the cache (e.g. put, remove, clear, store, etc.) do not affect the cache store. As a result, the cache store can become out of sync with the cache.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:infinispan:config:6.1">
 <global />
 <default />
 <namedCache name="persistentCache">
  <persistence shared="false">
  <store class="org.infinispan.loaders.file.FileCacheStore"
           fetchPersistentState="true"
           ignoreModifications="false"
```

6. **Configure Purge On Startup**
   The ***purgeOnStartup*** parameter specifies whether the cache is purged when initially started.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:infinispan:config:6.1">
 <global />
 <default />
 <namedCache name="persistentCache">
  <persistence shared="false">
  <store class="org.infinispan.loaders.file.FileCacheStore"
           fetchPersistentState="true"
           ignoreModifications="false"
           purgeOnStartup="false">
```

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

   a. The ***name*** parameter specifies the name of the property.

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:infinispan:config:6.1">
 <global />
 <default />
 <namedCache name="persistentCache">
  <persistence shared="false">
  <store class="org.infinispan.loaders.file.FileCacheStore"
           fetchPersistentState="true"
           ignoreModifications="false"
           purgeOnStartup="false">
    <properties>
     <property name="location"
        value="${java.io.tmpdir}" />
    </properties>
   </store>
  </persistence>
   </namedCache>
</infinispan>
```

Report a bug

## 20.2. WRITE-BEHIND CACHING

In Red Hat 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 contains stale data compared to the cache.

Report a bug

### 20.2.1. About Unscheduled Write-Behind Strategy

In the Unscheduled Write-Behind Strategy mode, Red Hat 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 the modifications 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

### 20.2.2. Unscheduled Write-Behind Strategy Configuration (Remote Client-Server Mode)

To set the write-behind strategy in Red Hat JBoss Data Grid's Remote Client-Server mode, add the **`write-behind`** element to the target cache store configuration as follows:

**Procedure 20.2. The `write-behind` Element**

The **`write-behind`** element uses the following configuration parameters:

1. **The *`modification-queue-size`* Parameter**
   The ***`modification-queue-size`*** parameter sets the modification queue size for the asynchronous store. If updates occur faster than the cache store can process the queue, the asynchronous store behaves like a synchronous store. The store behavior remains synchronous and blocks elements until the queue is able to accept them, after which the store behavior becomes asynchronous again.

```
<file-store passivation="false"
            path="${PATH}"
            purge="true"
            shared="false">
    <write-behind modification-queue-size="1024" />
```

2. **The *`shutdown-timeout`* Parameter**
   The ***`shutdown-timeout`*** parameter specifies the time in milliseconds after which the cache store is shut down. When the store is stopped some modifications may still need to be applied. Setting a large timeout value will reduce the chance of data loss. The default value for this parameter is **`25000`**.

```
<file-store passivation="false"
            path="${PATH}"
            purge="true"
            shared="false">
    <write-behind modification-queue-size="1024"
                  shutdown-timeout="25000" />
```

3. **The *`flush-lock-timeout`* Parameter**
   The ***`flush-lock-timeout`*** parameter specifies the time (in milliseconds) to acquire the lock that guards the state to be periodically flushed. The default value for this parameter is **`15000`**.

```
<file-store passivation="false"
            path="${PATH}"
            purge="true"
            shared="false">
    <write-behind modification-queue-size="1024"
                  shutdown-timeout="25000"
                  flush-lock-timeout="15000" />
```

4. **The *`thread-pool-size`* Parameter**
   The ***`thread-pool-size`*** parameter specifies the size of the thread pool. The threads in this thread pool apply modifications to the cache store. The default value for this parameter is **5**.

```
<file-store passivation="false"
            path="${PATH}"
            purge="true"
            shared="false">
    <write-behind modification-queue-size="1024"
                  shutdown-timeout="25000"
                  flush-lock-timeout="15000"
                  thread-pool-size="5" />
</file-store>
```

Report a bug

### 20.2.3. Unscheduled Write-Behind Strategy Configuration (Library Mode)

To enable the write-behind strategy of the cache entries to a store, add the **async** element to the store configuration as follows:

**Procedure 20.3. The async Element**

The **async** element uses the following configuration parameters:

1. The ***`modificationQueueSize`*** parameter sets the modification queue size for the asynchronous store. If updates occur faster than the cache store can process the queue, the asynchronous store behaves like a synchronous store. The store behavior remains synchronous and blocks elements until the queue is able to accept them, after which the store behavior becomes asynchronous again.

```
<persistence>
    <fileStore location="${LOCATION}">
        <async enabled="true"
                     modificationQueueSize="1024" />
```

2. The ***`shutdownTimeout`*** parameter specifies the time in milliseconds after which the cache store is shut down. This provides time for the asynchronous writer to flush data to the store when a cache is shut down. The default value for this parameter is **`25000`**.

```
<persistence>
    <fileStore location="${LOCATION}">
        <async enabled="true"
                     modificationQueueSize="1024"
                     shutdownTimeout="25000" />
```

3. The ***flushLockTimeout*** parameter specifies the time (in milliseconds) to acquire the lock that guards the state to be periodically flushed. The default value for this parameter is **15000**.

```
<persistence>
    <fileStore location="${LOCATION}">
        <async enabled="true"
                    modificationQueueSize="1024"
                    shutdownTimeout="25000"
                    flushLockTimeout="15000" />
```

4. The ***threadPoolSize*** parameter specifies the number of threads that concurrently apply modifications to the store. The default value for this parameter is **5**.

```
<persistence>
    <fileStore location="${LOCATION}">
        <async enabled="true"
                    modificationQueueSize="1024"
                    shutdownTimeout="25000"
                    flushLockTimeout="15000"
                    threadPoolSize="5"/>
    </fileStore>
</persistence>
```

Report a bug

# PART X. MONITOR CACHES AND CACHE MANAGERS

# CHAPTER 21. SET UP JAVA MANAGEMENT EXTENSIONS (JMX)

## 21.1. 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 Red Hat JBoss Data Grid to expose management and statistical information.

Report a bug

## 21.2. USING JMX WITH RED HAT JBOSS DATA GRID

Management in Red Hat 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

## 21.3. 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.

**IMPORTANT**

In Red Hat JBoss Data Grid, statistics are enabled by default. While statistics are useful in assessing the status of JBoss Data Grid, they adversely affect performance and must be disabled if they are not required.

Report a bug

## 21.4. 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

## 21.5. 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

## 21.6. DISABLING THE CACHESTORE VIA JMX

Red Hat JBoss Data Grid allows the CacheStore to be disabled via JMX by invoking the ***disconnectSource*** operation on the **RollingUpgradeManager** MBean.

**See Also:**

- Section A.15, “RollingUpgradeManager”

Report a bug

## 21.7. 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

## 21.8. MBEANS

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

Red Hat 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
```

**NOTE**

A full list of available MBeans, their supported operations and attributes, is available in the Appendix

Report a bug

### 21.8.1. 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 Red Hat 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

### 21.8.2. 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 Red Hat 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

# CHAPTER 22. SET UP JBOSS OPERATIONS NETWORK (JON)

## 22.1. 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 Red Hat 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.

**IMPORTANT**

In Red Hat JBoss Data Grid Remote Client-Server mode, statistics are enabled by default. While statistics are useful in assessing the status of JBoss Data Grid, they adversely affect performance and must be disabled if they are not required. In JBoss Data Grid Library mode, statistics are disabled by default and must be explicitly enabled when required.

**IMPORTANT**

To achieve full functionality of JBoss Operations Network library plugin for JBoss Data Grid's Library mode, upgrade to JBoss Operations Network 3.2.0 with patch `Update 02` or higher version. For information on upgrading the JBoss Operations Network, see the *Upgrading JBoss ON* section in the JBoss Operations Network *Installation Guide*.

Report a bug

## 22.2. DOWNLOAD JBOSS OPERATIONS NETWORK (JON)

### 22.2.1. Prerequisites for Installing JBoss Operations Network (JON)

In order to install JBoss Operations Network in Red Hat JBoss Data Grid, the following is required:

- A Linux, Windows, or Mac OSX operating system, and an x86_64, i686, or ia64 processor.
- Java 6 or higher is required to run both the JBoss Operations Network Server and the JBoss Operations Network Agent.
- Synchronized clocks on JBoss Operations Network Servers and Agents.
- An external database must be installed.

Report a bug

### 22.2.2. Download JBoss Operations Network

Use the following procedure to download Red Hat JBoss Operations Network (JON) from the Customer Portal:

**Procedure 22.1. Download JBoss Operations Network**

1. To access the Red Hat Customer Portal, navigate to https://access.redhat.com/home in a browser.

2. Click **Downloads**.

3. In the box labeled **`Red Hat JBoss Middleware`**, click the **Download Software** button.

4. Enter the relevant credentials in the **`Red Hat Login`** and **`Password`** fields and click **Log In**.

5. In the **`Software Downloads`** page, select **`JBoss Operations Network`** from the list of drop down values.

6. Select the appropriate version in the **`Version`** drop down menu list.

7. Click the **Download** button next to the desired download file.

Report a bug

## 22.2.3. Remote JMX Port Values

A port value must be provided to allow Red Hat 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

## 22.2.4. Download JBoss Operations Network (JON) Plugin

Complete this task to download the JBoss Operations Network (JON) plugin for Red Hat JBoss Data Grid from the Red Hat Customer Portal.

**Procedure 22.2. Download Installation Files**

1. Open http://access.redhat.com in a web browser.

2. Click **`Downloads`** in the menu across the top of the page.

3. Click **`Downloads`** in the list under JBoss Enterprise Middleware.

4. Enter your login information.

   You are taken to the Software Downloads page.

5. **Download the JBoss Operations Network Plugin**
   If you intend to use the JBoss Operations Network plugin for JBoss Data Grid, select **`JBoss ON for JDG`** from either the Software Downloads drop-down box, or the menu on the left.

   a. Click the **`JBoss Operations Network`** ***`VERSION`*** **`Base Distribution`** download link.

   b. Click the **`Download`** link to start the Base Distribution download.

c. Repeat the steps to download the `JDG Plugin Pack for JBoss ON` ***`VERSION`***

Report a bug

## 22.3. JBOSS OPERATIONS NETWORK SERVER INSTALLATION

The core of JBoss Operations Network is the server, which communicates with agents, maintains the inventory, manages resource settings, interacts with content providers, and provides a central management UI.

**NOTE**

For more detailed information about configuring JBoss Operations Network, see the JBoss Operations Network *Installation Guide*.

Report a bug

## 22.4. JBOSS OPERATIONS NETWORK AGENT

The JBoss Operations Network Agent is a standalone Java application. Only one agent is required per machine, regardless of how many resources you require the agent to manage.

The JBoss Operations Network Agent does not ship fully configured. Once the agent has been installed and configured it can be run as a Windows service from a console, or run as a daemon or **`init.d`** script in a UNIX environment.

A JBoss Operations Network Agent must be installed on each of the machines being monitored in order to collect data.

The JBoss Operations Network Agent is typically installed on the same machine on which Red Hat JBoss Data Grid is running, however where there are multiple machines an agent must be installed on each machine.

**NOTE**

For more detailed information about configuring JBoss Operations Network agents, see the JBoss Operations Network *Installation Guide*.

Report a bug

## 22.5. JBOSS OPERATIONS NETWORK FOR REMOTE CLIENT-SERVER MODE

In Red Hat JBoss Data Grid's Remote Client-Server mode, the JBoss Operations Network plug-in is used to

- initiate and perform installation and configuration operations.
- monitor resources and their metrics.

In Remote Client-Server mode, the JBoss Operations Network plug-in uses JBoss Enterprise Application Platform's management protocol to obtain metrics and perform operations on the JBoss Data Grid server.

Report a bug

### 22.5.1. Installing the JBoss Operations Network Plug-in (Remote Client-Server Mode)

The following procedure details how to install the JBoss Operations Network plug-ins for Red Hat JBoss Data Grid's Remote Client-Server mode.

1. **Install the plug-ins**
   - Copy the JBoss Data Grid server rhq plug-in to ***$JON_SERVER_HOME*/plugins**.
   - Copy the JBoss Enterprise Application Platform plug-in to ***$JON_SERVER_HOME*/plugins**.

   The server will automatically discover plug-ins here and deploy them. The plug-ins will be removed from the plug-ins directory after successful deployment.

2. **Obtain plug-ins**
   Obtain all available plug-ins from the JBoss Operations Network server. To do this, type the following into the agent's console:

   ```
   plugins update
   ```

3. **List installed plug-ins**
   Ensure the JBoss Enterprise Application Platform plug-in and the JBoss Data Grid server rhq plug-in are installed correctly using the following:

   ```
   plugins info
   ```

JBoss Operation Network can now discover running JBoss Data Grid servers.

Report a bug

## 22.6. JBOSS OPERATIONS NETWORK REMOTE-CLIENT SERVER PLUGIN

### 22.6.1. JBoss Operations Network Plugin Metrics

**Table 22.1. JBoss Operations Network Traits for the Cache Container (Cache Manager)**

| Trait Name | Display Name | Description |
|---|---|---|
| cache-manager-status | Cache Container Status | The current runtime status of a cache container. |
| cluster-name | Cluster Name | The name of the cluster. |
| members | Cluster Members | The names of the members of the cluster. |

| Trait Name | Display Name | Description |
|---|---|---|
| coordinator-address | Coordinator Address | The coordinator node's address. |
| local-address | Local Address | The local node's address. |
| version | Version | The cache manager version. |
| defined-cache-names | Defined Cache Names | The caches that have been defined for this manager. |

**Table 22.2. JBoss Operations Network Metrics for the Cache Container (Cache Manager)**

| Metric Name | Display Name | Description |
|---|---|---|
| cluster-size | Cluster Size | How many members are in the cluster. |
| defined-cache-count | Defined Cache Count | How many caches that have been defined for this manager. |
| running-cache-count | Running Cache Count | How many caches are running under this manager. |
| created-cache-count | Created Cache Count | How many caches have actually been created under this manager. |

**Table 22.3. JBoss Operations Network Traits for the Cache**

| Trait Name | Display Name | Description |
|---|---|---|
| cache-status | Cache Status | The current runtime status of a cache. |
| cache-name | Cache Name | The current name of the cache. |
| version | Version | The cache version. |

**Table 22.4. JBoss Operations Network Metrics for the Cache**

| Metric Name | Display Name | Description |
|---|---|---|
| cache-status | Cache Status | The current runtime status of a cache. |
| number-of-locks-available | [LockManager] Number of locks available | The number of exclusive locks that are currently available. |

| Metric Name | Display Name | Description |
|---|---|---|
| concurrency-level | [LockManager] Concurrency level | The LockManager's configured concurrency level. |
| average-read-time | [Statistics] Average read time | Average number of milliseconds required for a read operation on the cache to complete. |
| hit-ratio | [Statistics] Hit ratio | The result (in percentage) when the number of hits (successful attempts) is divided by the total number of attempts. |
| elapsed-time | [Statistics] Seconds since cache started | The number of seconds since the cache started. |
| read-write-ratio | [Statistics] Read/write ratio | The read/write ratio (in percentage) for the cache. |
| average-write-time | [Statistics] Average write time | Average number of milliseconds a write operation on a cache requires to complete. |
| hits | [Statistics] Number of cache hits | Number of cache hits. |
| evictions | [Statistics] Number of cache evictions | Number of cache eviction operations. |
| remove-misses | [Statistics] Number of cache removal misses | Number of cache removals where the key was not found. |
| time-since-reset | [Statistics] Seconds since cache statistics were reset | Number of seconds since the last cache statistics reset. |
| number-of-entries | [Statistics] Number of current cache entries | Number of entries currently in the cache. |
| stores | [Statistics] Number of cache puts | Number of cache put operations |
| remove-hits | [Statistics] Number of cache removal hits | Number of cache removal operation hits. |
| misses | [Statistics] Number of cache misses | Number of cache misses. |
| success-ratio | [RpcManager] Successful replication ratio | Successful replications as a ratio of total replications in numeric double format. |

| Metric Name | Display Name | Description |
|---|---|---|
| replication-count | [RpcManager] Number of successful replications | Number of successful replications |
| replication-failures | [RpcManager] Number of failed replications | Number of failed replications |
| average-replication-time | [RpcManager] Average time spent in the transport layer | The average time (in milliseconds) spent in the transport layer. |
| commits | [Transactions] Commits | Number of transaction commits performed since the last reset. |
| prepares | [Transactions] Prepares | Number of transaction prepares performed since the last reset. |
| rollbacks | [Transactions] Rollbacks | Number of transaction rollbacks performed since the last reset. |
| invalidations | [Invalidation] Number of invalidations | Number of invalidations. |
| passivations | [Passivation] Number of cache passivations | Number of passivation events. |
| activations | [Activations] Number of cache entries activated | Number of activation events. |
| cache-loader-loads | [Activation] Number of cache store loads | Number of entries loaded from the cache store. |
| cache-loader-misses | [Activation] Number of cache store misses | Number of entries that did not exist in the cache store. |
| cache-loader-stores | [CacheStore] Number of cache store stores | Number of entries stored in the cache stores. |

**NOTE**

Gathering of some of these statistics is disabled by default.

**JBoss Operations Network Metrics for Connectors**

The metrics provided by the JBoss Operations Network (JON) plugin for Red Hat JBoss Data Grid are for REST and Hot Rod endpoints only. For the REST protocol, the data must be taken from the Web subsystem metrics. For details about each of these endpoints, see the *Getting Started Guide*.

**Table 22.5. JBoss Operations Network Metrics for the Connectors**

| Metric Name | Display Name | Description |
|---|---|---|
| bytesRead | Bytes Read | Number of bytes read. |
| bytesWritten | Bytes Written | Number of bytes written. |

**NOTE**

Gathering of these statistics is disabled by default.

Report a bug

### 22.6.2. JBoss Operations Network Plugin Operations

**Table 22.6. JBoss ON Plugin Operations for the Cache**

| Operation Name | Description |
|---|---|
| Start Cache | Starts the cache. |
| Stop Cache | Stops the cache. |
| Clear Cache | Clears the cache contents. |
| Reset Statistics | Resets statistics gathered by the cache. |
| Reset Activation Statistics | Resets activation statistics gathered by the cache. |
| Reset Invalidation Statistics | Resets invalidations statistics gathered by the cache. |
| Reset Passivation Statistics | Resets passivation statistics gathered by the cache. |
| Reset Rpc Statistics | Resets replication statistics gathered by the cache. |
| Remove Cache | Removes the given cache from the cache-container. |
| Record Known Global Keyset | Records the global known keyset to a well-known key for retrieval by the upgrade process. |
| Synchronize Data | Synchronizes data from the old cluster to this using the specified migrator. |
| Disconnect Source | Disconnects the target cluster from the source cluster according to the specified migrator. |

**JBoss Operations Network Plugin Operations for the Cache Backups**

The cache backups used for these operations are configured using cross-datacenter replication. In the JBoss Operations Network (JON) User Interface, each cache backup is the child of a cache. For more information about cross-datacenter replication, see Chapter 29, *Set Up Cross-Datacenter Replication*

**Table 22.7. JBoss Operations Network Plugin Operations for the Cache Backups**

| Operation Name | Description |
|---|---|
| status | Display the site status. |
| bring-site-online | Brings the site online. |
| take-site-offline | Takes the site offline. |

**Cache (Transactions)**

Red Hat JBoss Data Grid does not support using Transactions in Remote Client-Server mode. As a result, none of the endpoints can use transactions.

Report a bug

### 22.6.3. JBoss Operations Network Plugin Attributes

**Table 22.8. JBoss ON Plugin Attributes for the Cache (Transport)**

| Attribute Name | Type | Description |
|---|---|---|
| cluster | string | The name of the group communication cluster. |
| executor | string | The executor used for the transport. |
| lock-timeout | long | The timeout period for locks on the transport. The default value is `240000`. |
| machine | string | A machine identifier for the transport. |
| rack | string | A rack identifier for the transport. |
| site | string | A site identifier for the transport. |
| stack | string | The JGroups stack used for the transport. |

Report a bug

## 22.7. JBOSS OPERATIONS NETWORK FOR LIBRARY MODE

In Red Hat JBoss Data Grid's Library mode, the JBoss Operations Network plug-in is used to

- initiate and perform installation and configuration operations.
- monitor resources and their metrics.

In Library mode, the JBoss Operations Network plug-in uses JMX to obtain metrics and perform operations on an application using the JBoss Data Grid library.

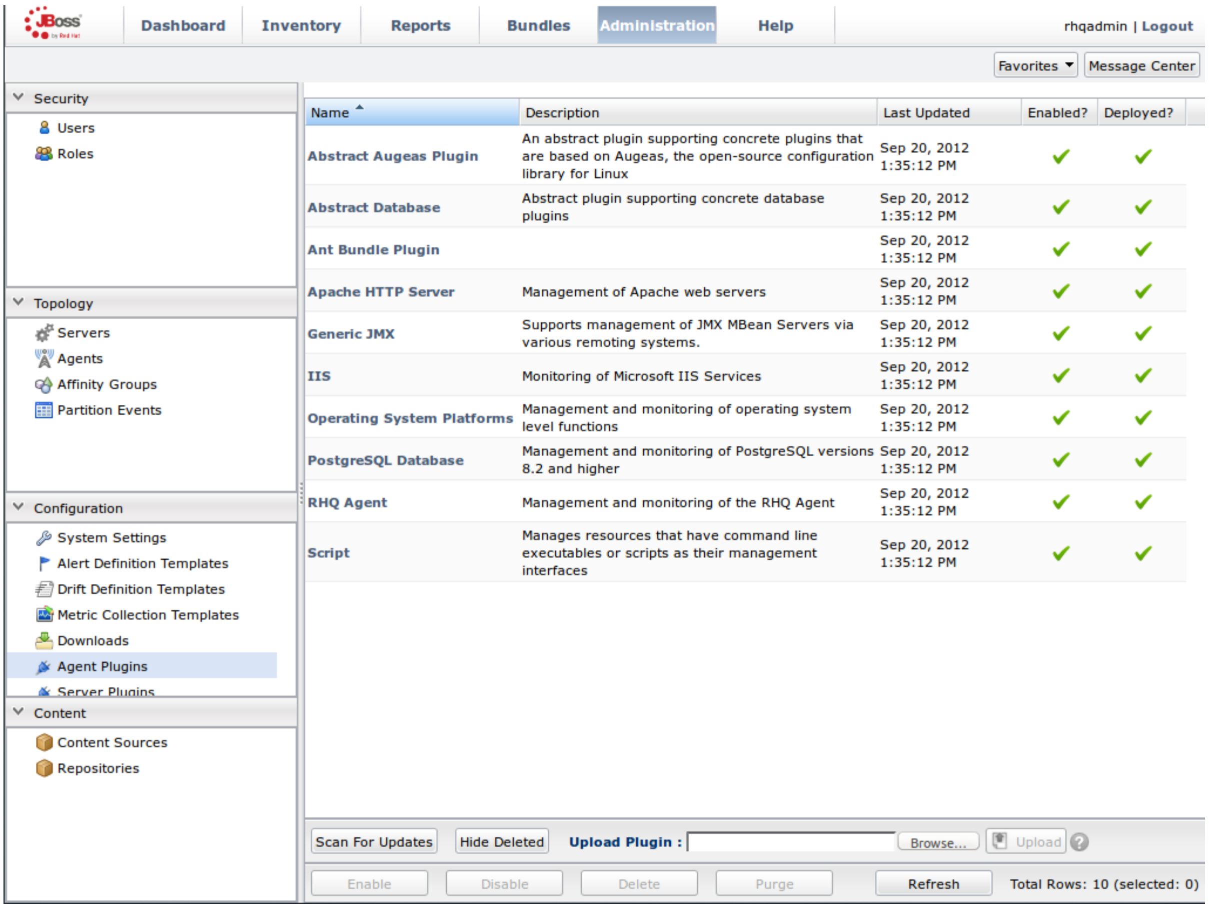
Report a bug

### 22.7.1. Installing the JBoss Operations Network Plug-in (Library Mode)

Use the following procedure to install the JBoss Operations Network plug-in for Red Hat JBoss Data Grid's Library mode.

**Procedure 22.3. Install JBoss Operations Network Library Mode Plug-in**

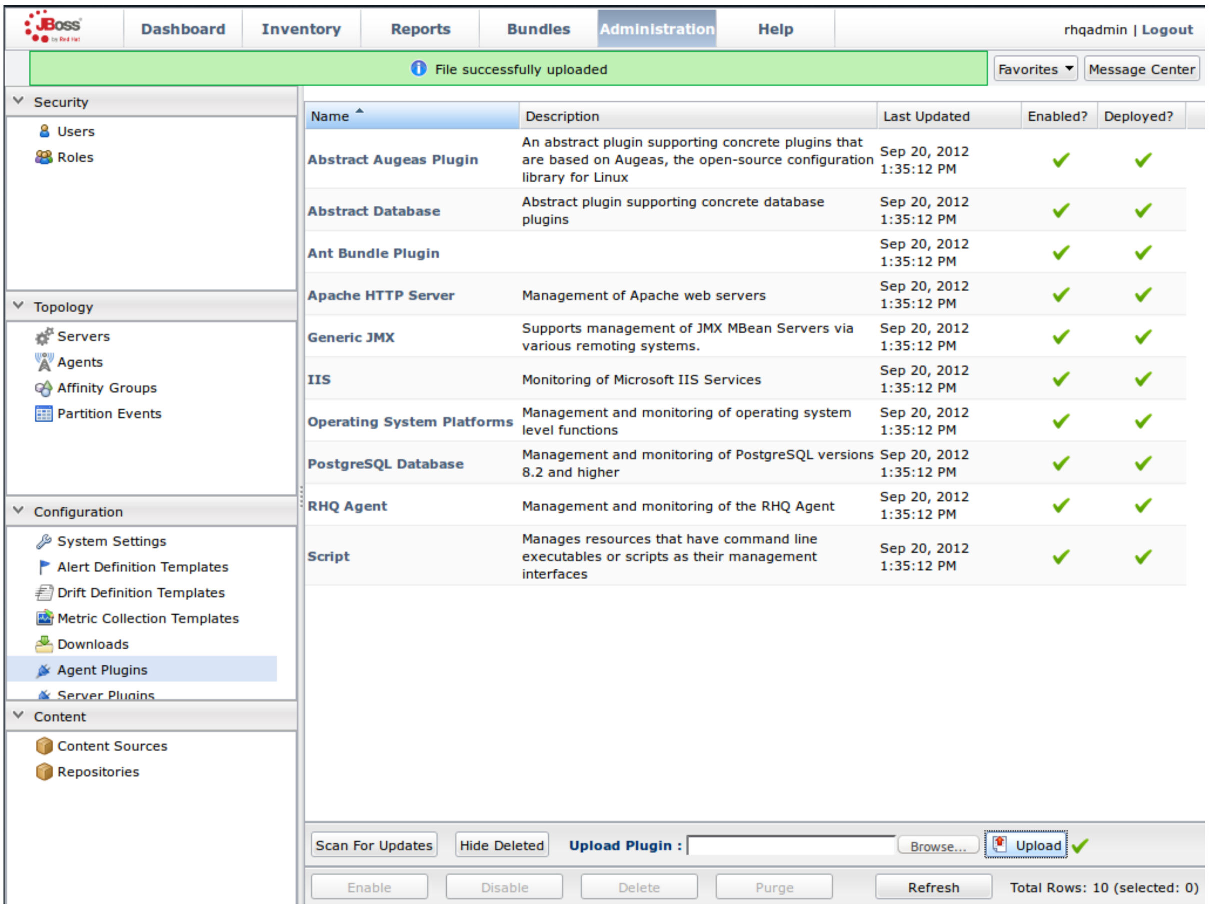
1. **Open the JBoss Operations Network Console**
   a. From the JBoss Operations Network console, select **Administration**.
   b. Select **Agent Plugins** from the **Configuration** options on the left side of the console.



| Name | Description | Last Updated | Enabled? | Deployed? |
|---|---|---|---|---|
| Abstract Augeas Plugin | An abstract plugin supporting concrete plugins that are based on Augeas, the open-source configuration library for Linux | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Abstract Database | Abstract plugin supporting concrete database plugins | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Ant Bundle Plugin | | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Apache HTTP Server | Management of Apache web servers | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Generic JMX | Supports management of JMX MBean Servers via various remoting systems. | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| IIS | Monitoring of Microsoft IIS Services | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Operating System Platforms | Management and monitoring of operating system level functions | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| PostgreSQL Database | Management and monitoring of PostgreSQL versions 8.2 and higher | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| RHQ Agent | Management and monitoring of the RHQ Agent | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Script | Manages resources that have command line executables or scripts as their management interfaces | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |

**Figure 22.1. JBoss Operations Network Console for JBoss Data Grid**

2. **Upload the Library Mode Plug-in**
   a. Click **`Browse`**, locate the **`InfinispanPlugin`** on your local file system.
   b. Click **`Upload`** to add the plug-in to the JBoss Operations Network Server.



File successfully uploaded

| Name | Description | Last Updated | Enabled? | Deployed? |
|---|---|---|---|---|
| Abstract Augeas Plugin | An abstract plugin supporting concrete plugins that are based on Augeas, the open-source configuration library for Linux | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Abstract Database | Abstract plugin supporting concrete database plugins | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Ant Bundle Plugin | | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Apache HTTP Server | Management of Apache web servers | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Generic JMX | Supports management of JMX MBean Servers via various remoting systems. | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| IIS | Monitoring of Microsoft IIS Services | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Operating System Platforms | Management and monitoring of operating system level functions | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| PostgreSQL Database | Management and monitoring of PostgreSQL versions 8.2 and higher | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| RHQ Agent | Management and monitoring of the RHQ Agent | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Script | Manages resources that have command line executables or scripts as their management interfaces | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |

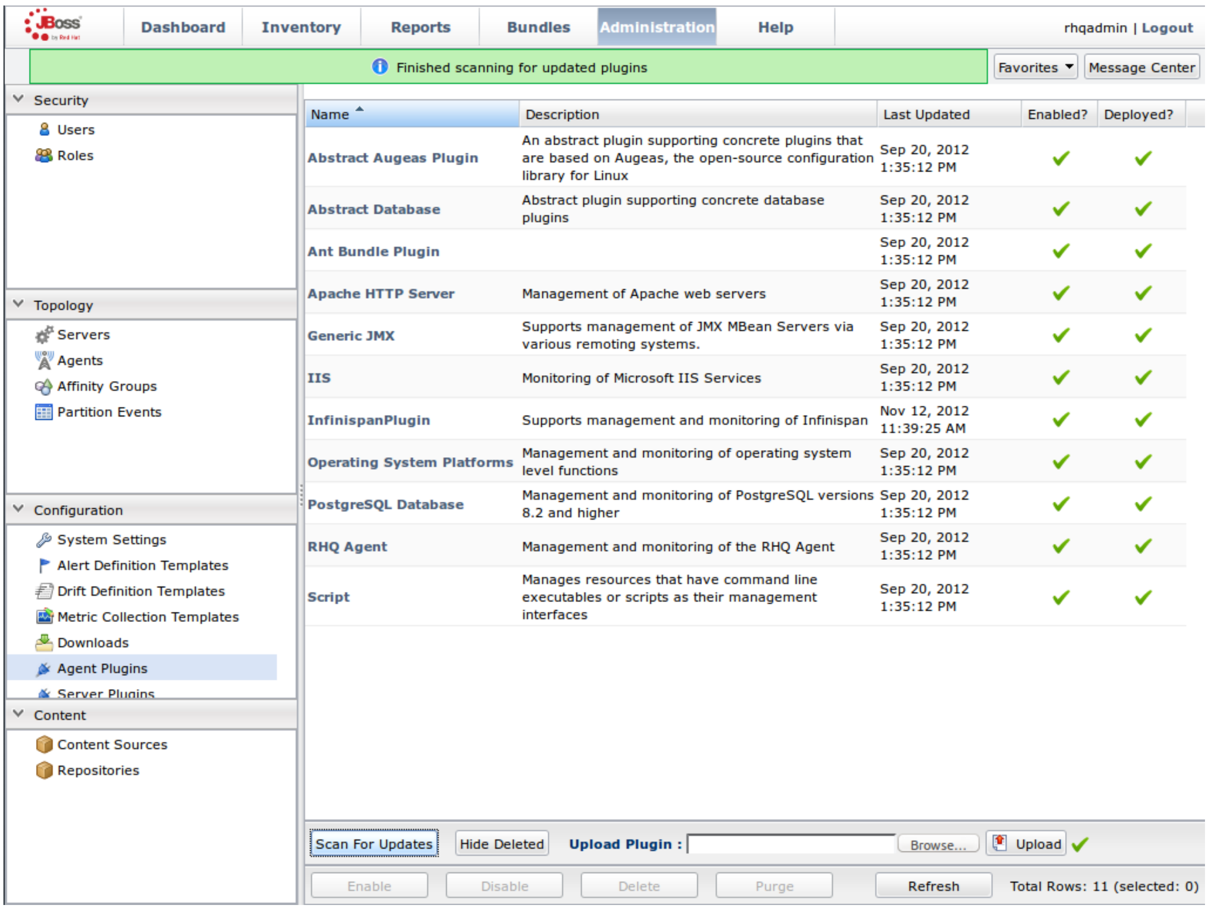
Total Rows: 10 (selected: 0)

**Figure 22.2. Upload the `InfinispanPlugin`.**

3. **Scan for Updates**

   a. Once the file has successfully uploaded, click **`Scan For Updates`** at the bottom of the screen.

   b. The **`InfinispanPlugin`** will now appear in the list of installed plug-ins.



| Name | Description | Last Updated | Enabled? | Deployed? |
|---|---|---|---|---|
| Abstract Augeas Plugin | An abstract plugin supporting concrete plugins that are based on Augeas, the open-source configuration library for Linux | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Abstract Database | Abstract plugin supporting concrete database plugins | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Ant Bundle Plugin | | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Apache HTTP Server | Management of Apache web servers | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Generic JMX | Supports management of JMX MBean Servers via various remoting systems. | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| IIS | Monitoring of Microsoft IIS Services | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| InfinispanPlugin | Supports management and monitoring of Infinispan | Nov 12, 2012 11:39:25 AM | ✓ | ✓ |
| Operating System Platforms | Management and monitoring of operating system level functions | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| PostgreSQL Database | Management and monitoring of PostgreSQL versions 8.2 and higher | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| RHQ Agent | Management and monitoring of the RHQ Agent | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |
| Script | Manages resources that have command line executables or scripts as their management interfaces | Sep 20, 2012 1:35:12 PM | ✓ | ✓ |

**Figure 22.3. Scan for Updated Plug-ins.**

Report a bug

## 22.7.2. Monitoring Of JBoss Data Grid Instances in Library Mode

### 22.7.2.1. Prerequisites

The following is a list of common prerequisites for Section 22.7.2.3.1, “Monitor an Application Deployed in Standalone Mode”, Section 22.7.2.3.2, “Monitor an Application Deployed in Domain Mode” and Section 22.7.2.2.1, “Manually Adding JBoss Data Grid Instances in Library Mode” .

- A correctly configured instance of JBoss Operations Network (JON) 3.2.0 with patch `Update 02` or higher version.
- A running instance of JON Agent on the server where the application will run. For more information, see Section 22.4, “JBoss Operations Network Agent”
- An operational instance of the RHQ agent with a full JDK. Ensure that the agent has access to the `tools.jar` file from the JDK in particular. In the JON agent's environment file ( `bin/rhq-env.sh`), set the value of the `RHQ_AGENT_JAVA_HOME` property to point to a full JDK home.
- The RHQ agent must have been initiated using the same user as the JBoss Enterprise Application Platform instance. As an example, running the JON agent as a user with root privileges and the JBoss Enterprise Application Platform process under a different user does not work as expected and must be avoided.
- An installed JON plugin for JBoss Data Grid Library Mode. For more information, see Section 22.7.1, “Installing the JBoss Operations Network Plug-in (Library Mode)”

- `Generic JMX plugin` from JBoss Operation Networks 3.2.0 with patch `Update 02` or better version in use.

- A custom application using Red Hat JBoss Data Grid's Library mode with enabled JMX statistics for library mode caches in order to make statistics and monitoring working. For details how to enable JMX statistics for cache instances, see Section 21.4, “Enable JMX for Cache Instances” and to enable JMX for cache managers see Section 21.5, “Enable JMX for CacheManagers”

- The Java Virtual Machine (JVM) must be configured to expose the JMX MBean Server. For the Oracle/Sun JDK, see http://docs.oracle.com/javase/1.5.0/docs/guide/management/agent.html

- A correctly added and configured management user for JBoss Enterprise Application Platform.

Report a bug

### 22.7.2.2. Manually Adding JBoss Data Grid Instances in Library Mode

#### 22.7.2.2.1. Manually Adding JBoss Data Grid Instances in Library Mode

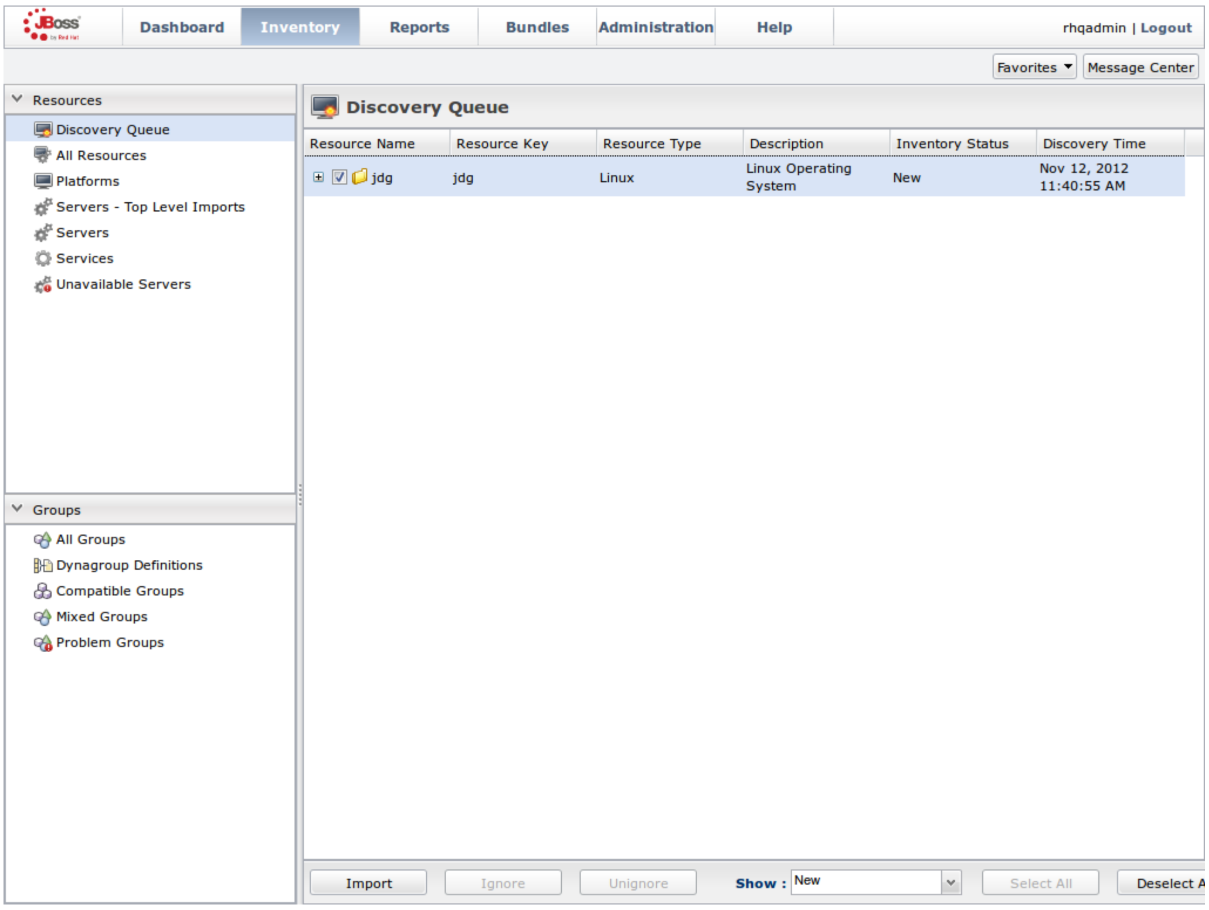
To add Red Hat JBoss Data Grid instances to JBoss Operations Network manually, use the following procedure in the JBoss Operations Network interface.

**Procedure 22.4. Add JBoss Data Grid Instances in Library Mode**

1. **Import the Platform**

   a. Navigate to the **Inventory** and select **Discovery Queue** from the **Resources** list on the left of the console.

   b. Select the platform on which the application is running and click `Import` at the bottom of the screen.



**Discovery Queue**

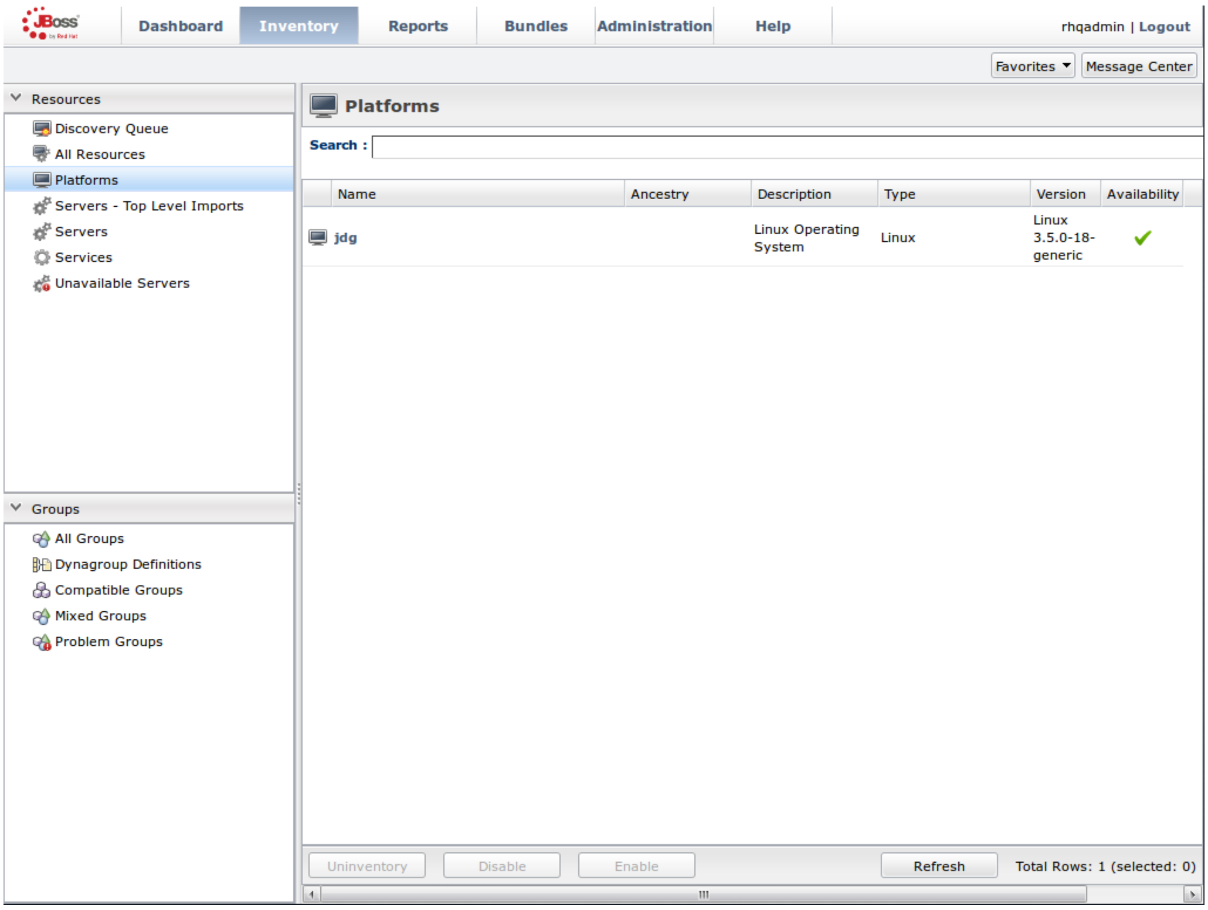
| Resource Name | Resource Key | Resource Type | Description | Inventory Status | Discovery Time |
|---|---|---|---|---|---|
| jdg | jdg | Linux | Linux Operating System | New | Nov 12, 2012 11:40:55 AM |

**Figure 22.4. Import the Platform from the Discovery Queue.**

2. **Access the Servers on the Platform**

   a. The **`jdg`** Platform now appears in the **`Platforms`** list.

   b. Click on the Platform to access the servers that are running on it.



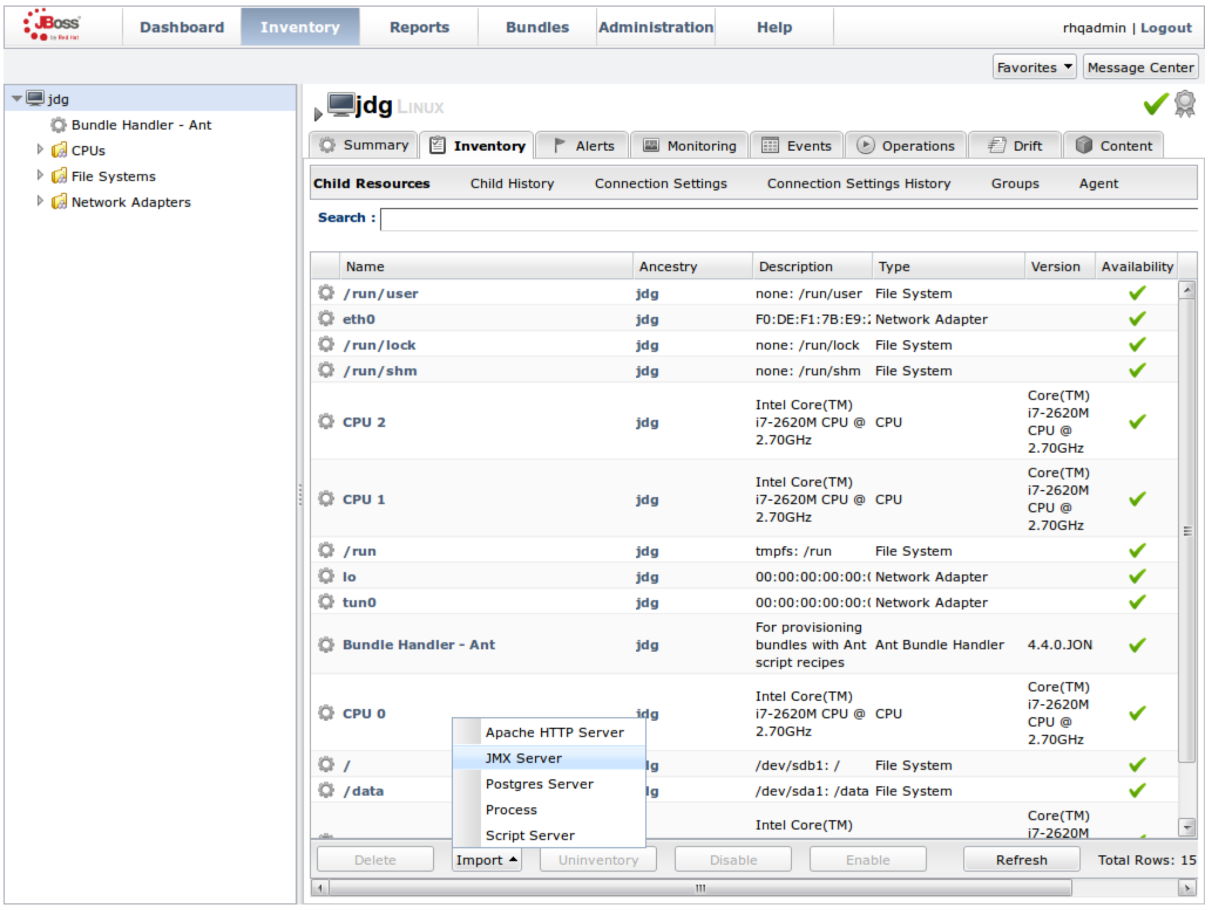
| Name | Ancestry | Description | Type | Version | Availability |
|---|---|---|---|---|---|
| jdg | | Linux Operating System | Linux | Linux 3.5.0-18-generic | ✓ |

**Figure 22.5. Open the `jdg` Platform to view the list of servers.**

3. **Import the JMX Server**

   a. From the **Inventory** tab, select **Child Resources**.

   b. Click the **`Import`** button at the bottom of the screen and select the **JMX Server** option from the list.

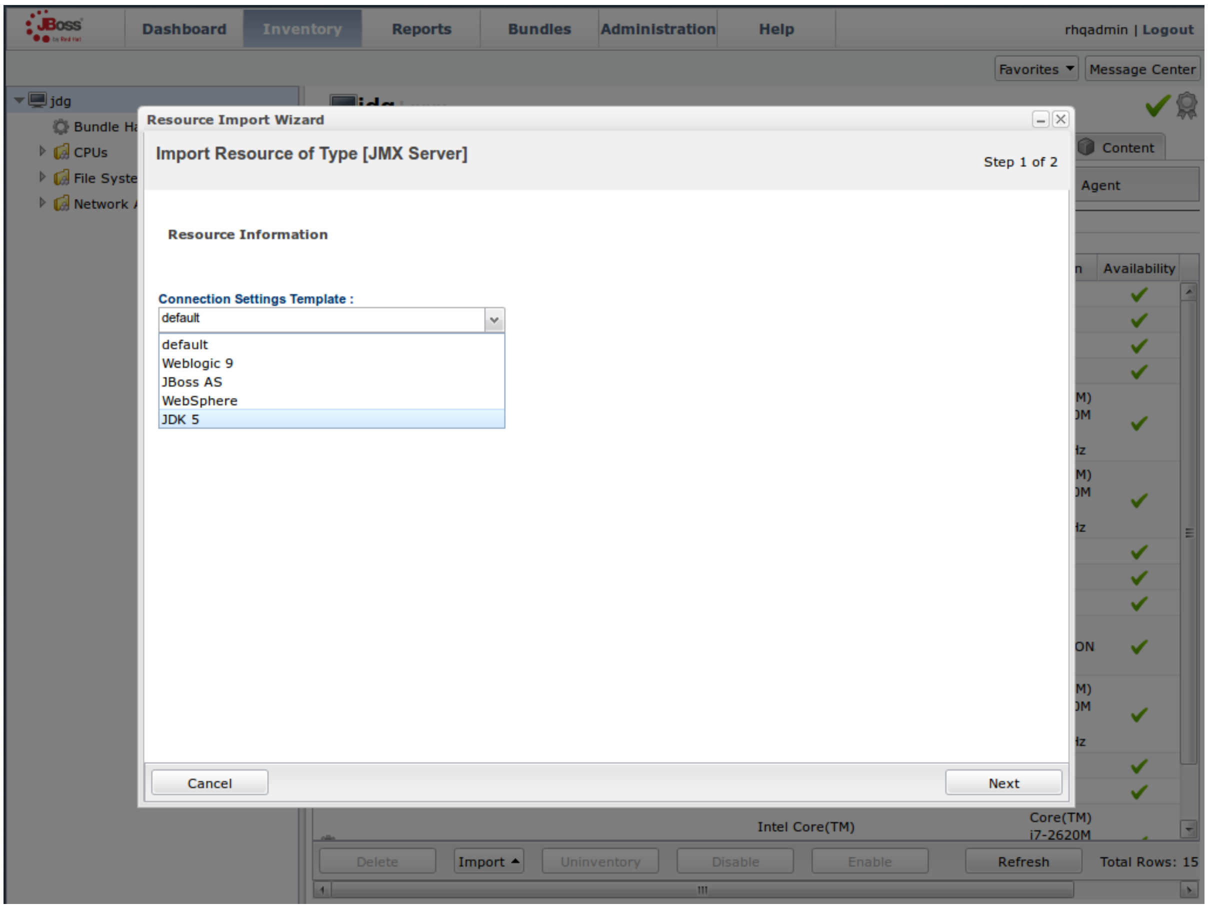


| Name | Ancestry | Description | Type | Version | Availability |
|---|---|---|---|---|---|
| /run/user | jdg | none: /run/user | File System | | ✓ |
| eth0 | jdg | F0:DE:F1:7B:E9:: | Network Adapter | | ✓ |
| /run/lock | jdg | none: /run/lock | File System | | ✓ |
| /run/shm | jdg | none: /run/shm | File System | | ✓ |
| CPU 2 | jdg | Intel Core(TM) i7-2620M CPU @ 2.70GHz | CPU | Core(TM) i7-2620M CPU @ 2.70GHz | ✓ |
| CPU 1 | jdg | Intel Core(TM) i7-2620M CPU @ 2.70GHz | CPU | Core(TM) i7-2620M CPU @ 2.70GHz | ✓ |
| /run | jdg | tmpfs: /run | File System | | ✓ |
| lo | jdg | 00:00:00:00:00:( | Network Adapter | | ✓ |
| tun0 | jdg | 00:00:00:00:00:( | Network Adapter | | ✓ |
| Bundle Handler - Ant | jdg | For provisioning bundles with Ant script recipes | Ant Bundle Handler | 4.4.0.JON | ✓ |
| CPU 0 | jdg | Intel Core(TM) i7-2620M CPU @ 2.70GHz | CPU | Core(TM) i7-2620M CPU @ 2.70GHz | ✓ |
| / | ig | /dev/sdb1: / | File System | | ✓ |
| /data | ig | /dev/sda1: /data | File System | | ✓ |
| | | Intel Core(TM) | | Core(TM) i7-2620M | |

Figure 22.6. Import the JMX Server

4. **Enable JDK Connection Settings**

   a. In the **Resource Import Wizard** window, specify **JDK 5** from the list of **Connection Settings Template** options.



Connection Settings Template : default

Figure 22.7. Select the JDK 5 Template.

5. **Modify the Connector Address**

   a. In the **Deployment Options** menu, modify the supplied **Connector Address** with the hostname and JMX port of the process containing the Infinispan Library.

   b. Enter the JMX connector address of the new JBoss Data Grid instance you want to monitor. For example:

      Connector Address:

      ```
      service:jmx:rmi://127.0.0.1/jndi/rmi://127.0.0.1:7997/jmxrmi
      ```

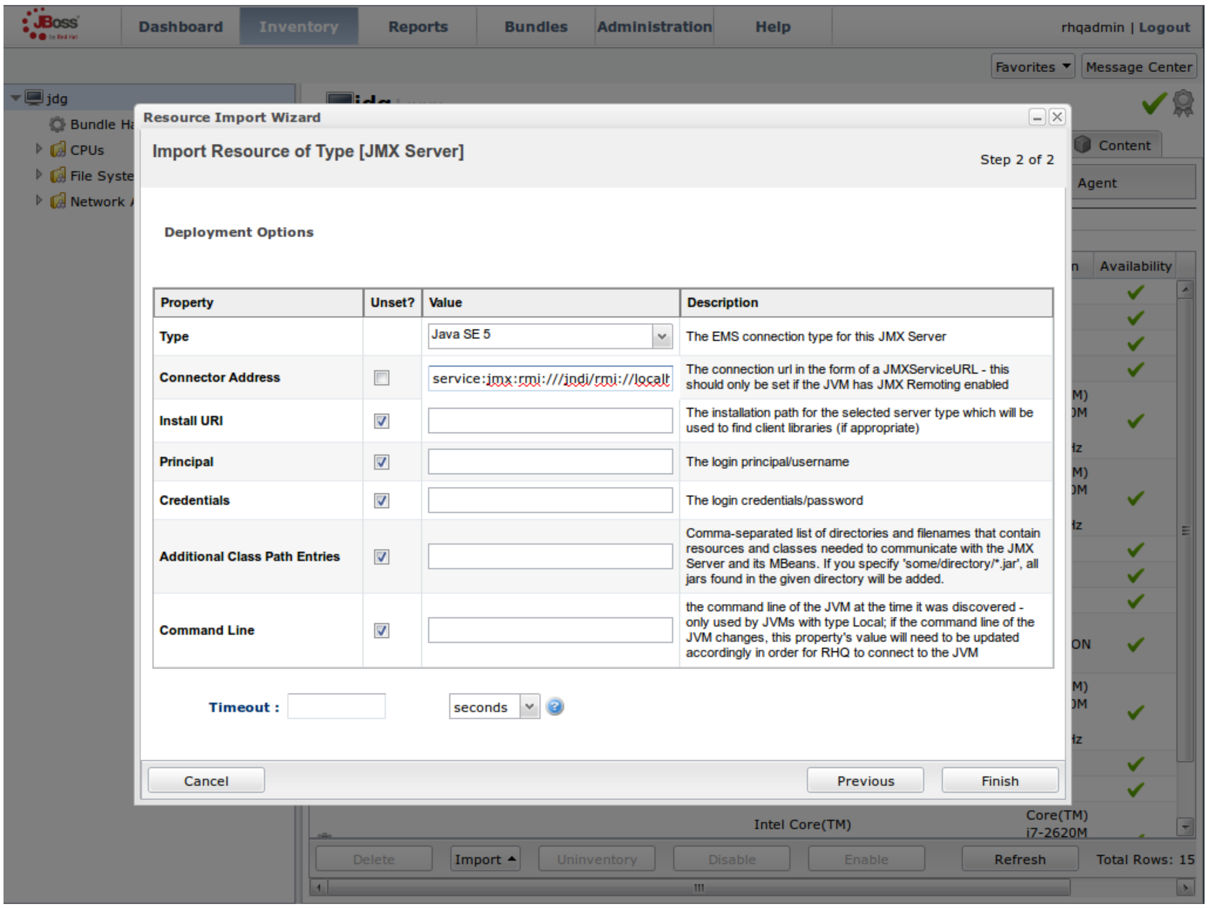
      **NOTE**

      The connector address varies depending on the host and the JMX port assigned to the new instance. In this case, instances require the following system properties at start up:

      ```
      -Dcom.sun.management.jmxremote.port=7997 -
      Dcom.sun.management.jmxremote.ssl=false -
      Dcom.sun.management.jmxremote.authenticate=false
      ```

   c. Specify the **Principal** and **Credentials** information if required.

   d. Click **Finish**.



Resource Import Wizard

Import Resource of Type [JMX Server]

Step 2 of 2

Deployment Options

| Property | Unset? | Value | Description |
|---|---|---|---|
| Type | | Java SE 5 | The EMS connection type for this JMX Server |
| Connector Address | ☐ | service:jmx:rmi:///jndi/rmi://localt | The connection url in the form of a JMXServiceURL - this should only be set if the JVM has JMX Remoting enabled |
| Install URI | ☑ | | The installation path for the selected server type which will be used to find client libraries (if appropriate) |
| Principal | ☑ | | The login principal/username |
| Credentials | ☑ | | The login credentials/password |
| Additional Class Path Entries | ☑ | | Comma-separated list of directories and filenames that contain resources and classes needed to communicate with the JMX Server and its MBeans. If you specify 'some/directory/*.jar', all jars found in the given directory will be added. |
| Command Line | ☑ | | the command line of the JVM at the time it was discovered - only used by JVMs with type Local; if the command line of the JVM changes, this property's value will need to be updated accordingly in order for RHQ to connect to the JVM |

Timeout : seconds

**Figure 22.8. Modify the values in the Deployment Options screen.**

6. **View Cache Statistics and Operations**

   a. Click **`Refresh`** to refresh the list of servers.

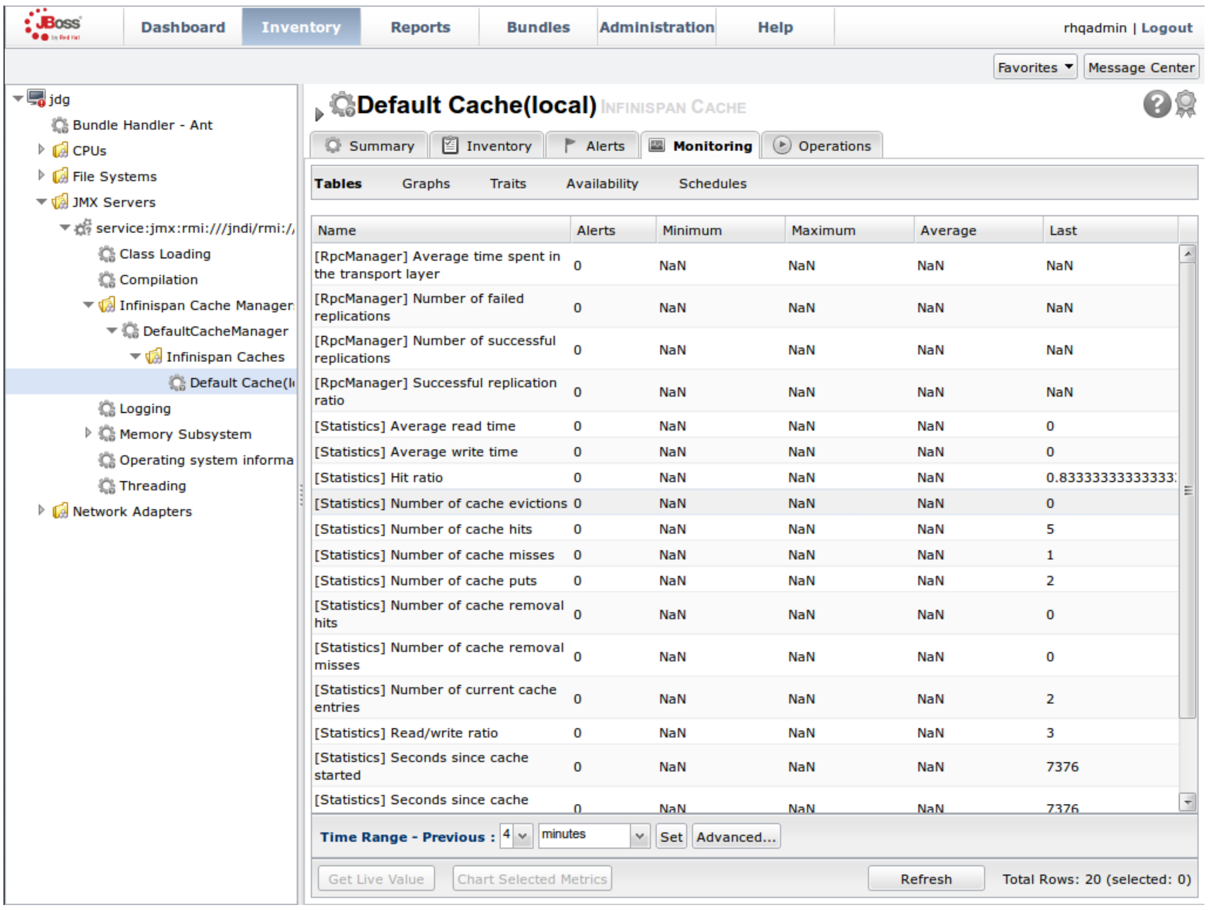
   b. The **JMX Servers** tree in the panel on the left side of the screen contains the **Infinispan Cache Managers** node, which contains the available cache managers. The available cache managers contain the available caches.

   c. Select a cache from the available caches to view metrics.

   d. Select the **Monitoring** tab.

   e. The **`Tables`** view shows statistics and metrics.

   f. The **Operations** tab provides access to the various operations that can be performed on the services.



| Name | Alerts | Minimum | Maximum | Average | Last |
|---|---|---|---|---|---|
| [RpcManager] Average time spent in the transport layer | 0 | NaN | NaN | NaN | NaN |
| [RpcManager] Number of failed replications | 0 | NaN | NaN | NaN | NaN |
| [RpcManager] Number of successful replications | 0 | NaN | NaN | NaN | NaN |
| [RpcManager] Successful replication ratio | 0 | NaN | NaN | NaN | NaN |
| [Statistics] Average read time | 0 | NaN | NaN | NaN | 0 |
| [Statistics] Average write time | 0 | NaN | NaN | NaN | 0 |
| [Statistics] Hit ratio | 0 | NaN | NaN | NaN | 0.83333333333333 |
| [Statistics] Number of cache evictions | 0 | NaN | NaN | NaN | 0 |
| [Statistics] Number of cache hits | 0 | NaN | NaN | NaN | 5 |
| [Statistics] Number of cache misses | 0 | NaN | NaN | NaN | 1 |
| [Statistics] Number of cache puts | 0 | NaN | NaN | NaN | 2 |
| [Statistics] Number of cache removal hits | 0 | NaN | NaN | NaN | 0 |
| [Statistics] Number of cache removal misses | 0 | NaN | NaN | NaN | 0 |
| [Statistics] Number of current cache entries | 0 | NaN | NaN | NaN | 2 |
| [Statistics] Read/write ratio | 0 | NaN | NaN | NaN | 3 |
| [Statistics] Seconds since cache started | 0 | NaN | NaN | NaN | 7376 |
| [Statistics] Seconds since cache | 0 | NaN | NaN | NaN | 7376 |

Time Range - Previous : 4 minutes Set Advanced...

Get Live Value Chart Selected Metrics Refresh Total Rows: 20 (selected: 0)

**Figure 22.9. Metrics and operational data relayed through JMX is now available in the JBoss Operations Network console.**

Report a bug

### 22.7.2.3. Monitor Custom Applications Using Library Mode Deployed On JBoss Enterprise Application Platform

#### 22.7.2.3.1. Monitor an Application Deployed in Standalone Mode

Use the following instructions to monitor an application deployed in JBoss Enterprise Application Platform using its standalone mode:

**Procedure 22.5. Monitor an Application Deployed in Standalone Mode**

1. **Start the JBoss Enterprise Application Platform Instance**
   Start the JBoss Enterprise Application Platform instance as follows:

   a. Enter the following command at the command line or change standalone configuration file (`/bin/standalone.conf`) respectively:

   ```
   JAVA_OPTS="$JAVA_OPTS -Dorg.rhq.resourceKey=MyEAP"
   ```

   b. Start the JBoss Enterprise Application Platform instance in standalone mode as follows:

   ```
   $JBOSS_HOME/bin/standalone.sh
   ```

2. **Deploy the Red Hat JBoss Data Grid Application**

Deploy the WAR file that contains the JBoss Data Grid Library mode application with **globalJmxStatistics** and **jmxStatistics** enabled.

3. **Run JBoss Operations Network (JON) Discovery**
   Run the **discovery --full** command in the JBoss Operations Network (JON) agent.

4. **Locate Application Server Process**
   In the JBoss Operations Network (JON) web interface, the JBoss Enterprise Application Platform process is listed as a JMX server.

5. **Import the Process Into Inventory**
   Import the process into the JBoss Operations Network (JON) inventory.

6. **Optional: Run Discovery Again**
   If required, run the **discovery --full** command again to discover the new resources.

**Result**

The JBoss Data Grid Library mode application is now deployed in JBoss Enterprise Application Platform's standalone mode and can be monitored using the JBoss Operations Network (JON).

Report a bug

#### 22.7.2.3.2. Monitor an Application Deployed in Domain Mode

Use the following instructions to monitor an application deployed in JBoss Enterprise Application Platform 6 using its domain mode:

**Procedure 22.6. Monitor an Application Deployed in Domain Mode**

1. **Edit the Host Configuration**
   Edit the **domain/configuration/host.xml** file to replace the **server** element with the following configuration:

```
<servers>
 <server name="server-one" group="main-server-group">
  <jvm name="default">
   <jvm-options>
    <option value="-Dorg.rhq.resourceKey=EAP1"/>
   </jvm-options>
  </jvm>
 </server>
 <server name="server-two" group="main-server-group" auto-
start="true">
  <socket-bindings port-offset="150"/>
  <jvm name="default">
   <jvm-options>
    <option value="-Dorg.rhq.resourceKey=EAP2"/>
   </jvm-options>
  </jvm>
 </server>
</servers>
```

2. **Start JBoss Enterprise Application Platform 6**

Start JBoss Enterprise Application Platform 6 in domain mode:

```
$JBOSS_HOME/bin/domain.sh
```

3. **Deploy the Red Hat JBoss Data Grid Application**
   Deploy the WAR file that contains the JBoss Data Grid Library mode application with **`globalJmxStatistics`** and **`jmxStatistics`** enabled.

4. **Run Discovery in JBoss Operations Network (JON)**
   If required, run the **`discovery --full`** command for the JBoss Operations Network (JON) agent to discover the new resources.

**Result**

The JBoss Data Grid Library mode application is now deployed in JBoss Enterprise Application Platform's domain mode and can be monitored using the JBoss Operations Network (JON).

Report a bug

## 22.8. 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

# PART XI. COMMAND LINE TOOLS

Red Hat JBoss Data Grid includes two command line tools for interacting with the caches in the data grid:

- The JBoss Data Grid Library CLI. For more information, see Section 23.1, “Red Hat JBoss Data Grid Library Mode CLI”.
- The JBoss Data Grid Server CLI. For more information, see Section 23.2, “Red Hat Data Grid Server CLI”.

Report a bug

# CHAPTER 23. RED HAT JBOSS DATA GRID CLIS

Red Hat JBoss Data Grid includes two Command Line Interfaces: a Library Mode CLI (see Section 23.1, “Red Hat JBoss Data Grid Library Mode CLI” for details) and a Server Mode CLI (see Section 23.2, “Red Hat Data Grid Server CLI” for details).

Report a bug

## 23.1. RED HAT JBOSS DATA GRID LIBRARY MODE CLI

Red Hat JBoss Data Grid includes the Red Hat JBoss Data Grid Library Mode Command Line Interface (CLI) that is used to inspect and modify data within caches and internal components (such as transactions, cross-datacenter replication sites, and rolling upgrades). The JBoss Data Grid Library Mode CLI can also be used for more advanced operations such as transactions.

The Library Mode CLI consists of a server-side module and a client command tool. The server-side module (**`infinispan-cli-server-`*`$VERSION`*`.jar`**) includes an interpreter for commands and must be included in the application.

Report a bug

### 23.1.1. Start the Library Mode CLI (Server)

Start the Red Hat JBoss Data Grid CLI's server-side module with the **`standalone`** and **`cluster`** files. For Linux, use the **`standlaone.sh`** or **`clustered.sh`** script and for Windows, use the **`standalone.bat`** or **`clustered.bat`** file.

Report a bug

### 23.1.2. Start the Library Mode CLI (Client)

Start the Red Hat JBoss Data Grid CLI client using the **`cli`** files in the **`bin`** directory. For Linux, run **`bin/cli.sh`** and for Windows, run **`bin\cli.bat`**.

When starting up the CLI client, specific command line switches can be used to customize the start up.

Report a bug

### 23.1.3. CLI Client Switches for the Command Line

The listed command line switches are appended to the command line when starting the Red Hat JBoss Data Grid CLI command:

**Table 23.1. CLI Client Command Line Switches**

| Short Option | Long Option | Description |
|---|---|---|

| Short Option | Long Option | Description |
|---|---|---|
| -c | --connect=${URL} | Connects to a running Red Hat JBoss Data Grid instance. For example, for JMX over RMI use **`jmx://[username[:password]]@host:port[/container[/cache]]`** and for JMX over JBoss Remoting use **`remoting://[username[:password]]@host:port[/container[/cache]]`** |
| -f | --file=${FILE} | Read the input from the specified file rather than using interactive mode. If the value is set to **`-`** then the ***`stdin`*** is used as the input. |
| -h | --help | Displays the help information. |
| -v | --version | Displays the CLI version information. |

Report a bug

### 23.1.4. Connect to the Application

Use the following command to connect to the application using the CLI:

```
[disconnected//]> connect jmx://localhost:12000
[jmx://localhost:12000/MyCacheManager/>
```

**NOTE**

The port value **`12000`** depends on the value the JVM is started with. For example, starting the JVM with the **`-Dcom.sun.management.jmxremote.port=12000`** command line parameter uses this port, but otherwise a random port is chosen. When the remoting protocol (**`remoting://localhost:9999`**) is used, the Red Hat JBoss Data Grid server administration port is used (the default is port **`9999`**).

The command line prompt displays the active connection information, including the currently selected **`CacheManager`**.

Use the **cache** command to select a cache before performing cache operations. The CLI supports tab completion, therefore using the **cache** and pressing the tab button displays a list of active caches:

```
[[jmx://localhost:12000/MyCacheManager/> cache
___defaultcache  namedCache
[jmx://localhost:12000/MyCacheManager/]> cache ___defaultcache
[jmx://localhost:12000/MyCacheManager/___defaultcache]>
```

Additionally, pressing tab displays a list of valid commands for the CLI.

Report a bug

### 23.1.5. Stopping a Red Hat JBoss Data Grid Instance with the CLI

**Library Mode**

When running a Red Hat JBoss Data Grid instance in a container as a library mode deployment, the life cycle of JBoss Data Grid is bound to the life cycle of the user deployment.

Stop or undeploy the user deployment using you container's management interfaces. For example, in Red Hat JBoss Enterprise Application Platform, use the Management CLI. See the *Management Interfaces* chapter in the JBoss Enterprise Application Platform *Administration and Configuration Guide* for more information.

**Remote Client-Server Mode**

A remote client-server JBoss Data Grid instance can be stopped using the following script:

```
jboss-datagrid-{VERSION}-server/bin/init.d/jboss-datagrid.sh stop
```

Alternatively, you can use the **kill** commands directly:

```
kill -15 $pid # send the TERM signal
```

If after a period of time the **PID** is still there, use the following:

```
kill -9 $pid
```

Report a bug

## 23.2. RED HAT DATA GRID SERVER CLI

Red Hat JBoss Data Grid includes a new Remote Client-Server mode CLI. This CLI can only be used for specific use cases, such as manipulating the server subsystem for the following:

- configuration
- management
- obtaining metrics

Report a bug

### 23.2.1. Start the Server Mode CLI

Use the following commands to run the JBoss Data Grid Server CLI from the command line:

For Linux:

```
$ JDG_HOME/bin/cli.sh
```

For Windows:

```
C:\>JDG_HOME\bin\cli.bat
```

Report a bug

## 23.3. CLI COMMANDS

Unless specified otherwise, all listed commands for the JBoss Data Grid CLIs can be used with both the Library Mode and Server Mode CLIs. Specifically, the **deny** (see Section 23.3.8, “The deny Command”), **`grant`** (see Section 23.3.14, “The grant Command” ), and **`roles`** (see Section 23.3.19, “The roles command”) commands are only available on the Server Mode CLI.

Report a bug

### 23.3.1. The abort Command

The **`abort`** command aborts a running batch initiated using the **`start`** command. Batching must be enabled for the specified cache. The following is a usage example:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> start
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> abort
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
null
```

Report a bug

### 23.3.2. The begin Command

The **`begin`** command starts a transaction. This command requires transactions enabled for the cache it targets. An example of this command's usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> begin
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> commit
```

Report a bug

### 23.3.3. The cache Command

The **cache** command specifies the default cache used for all subsequent operations. If invoked without any parameters, it shows the currently selected cache. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> cache ___defaultcache
[jmx://localhost:12000/MyCacheManager/___defaultcache]> cache
___defaultcache
[jmx://localhost:12000/MyCacheManager/___defaultcache]>
```

Report a bug

### 23.3.4. The clear Command

The **clear** command clears all content from the cache. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> clear
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
null
```

Report a bug

### 23.3.5. The commit Command

The **commit** command commits changes to an ongoing transaction. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> begin
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> commit
```

Report a bug

### 23.3.6. The container Command

The **container** command selects the default cache container (cache manager). When invoked without any parameters, it lists all available containers. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> container
MyCacheManager OtherCacheManager
[jmx://localhost:12000/MyCacheManager/namedCache]> container
OtherCacheManager
[jmx://localhost:12000/OtherCacheManager/]>
```

Report a bug

### 23.3.7. The create Command

The **create** command creates a new cache based on the configuration of an existing cache definition. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> create newCache like
namedCache
[jmx://localhost:12000/MyCacheManager/namedCache]> cache newCache
[jmx://localhost:12000/MyCacheManager/newCache]>
```

Report a bug

### 23.3.8. The deny Command

When authorization is enabled and the role mapper has been configured to be the ClusterRoleMapper, principal to role mappings are stored within the cluster registry (a replicated cache available to all nodes). The **deny** command can be used to deny roles previously assigned to a principal:

```
[remoting://localhost:9999]> deny supervisor to user1
```

**NOTE**

The **deny** command is only available to the JBoss Data Grid Server Mode CLI.

Report a bug

### 23.3.9. The disconnect Command

The **disconnect** command disconnects the currently active connection, which allows the CLI to connect to another instance. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> disconnect
[disconnected//]
```

Report a bug

### 23.3.10. The encoding Command

The **encoding** command sets a default codec to use when reading and writing entries to and from a cache. If invoked with no arguments, the currently selected codec is displayed. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> encoding
none
[jmx://localhost:12000/MyCacheManager/namedCache]> encoding --list
memcached
hotrod
none
rest
[jmx://localhost:12000/MyCacheManager/namedCache]> encoding hotrod
```

Report a bug

### 23.3.11. The end Command

The **end** command ends a running batch initiated using the **start** command. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> start
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> end
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
a
```

Report a bug

### 23.3.12. The evict Command

The **evict** command evicts an entry associated with a specific key from the cache. An example of it usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> evict a
```

Report a bug

### 23.3.13. The get Command

The **get** command shows the value associated with a specified key. For primitive types and Strings, the **get** command prints the default representation. For other objects, a  JSON representation of the object is printed. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
a
```

Report a bug

### 23.3.14. The grant Command

When authorization is enabled and the role mapper has been configured to be the **ClusterRoleMapper**, the principal to role mappings are stored within the cluster registry (a replicated cache available to all nodes). The **grant** command can be used to grant new roles to a principal as follows:

```
[remoting://localhost:9999]> grant supervisor to user1
```

**NOTE**

The **grant** command is only available to the JBoss Data Grid Server Mode CLI.

Report a bug

### 23.3.15. The info Command

The **info** command displaysthe configuration of a selected cache or container. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> info
GlobalConfiguration{asyncListenerExecutor=ExecutorFactoryConfiguration{fac
tory=org.infinispan.executors.DefaultExecutorFactory@98add58},
asyncTransportExecutor=ExecutorFactoryConfiguration{factory=org.infinispan
.executors.DefaultExecutorFactory@7bc9c14c},
evictionScheduledExecutor=ScheduledExecutorFactoryConfiguration{factory=or
g.infinispan.executors.DefaultScheduledExecutorFactory@7ab1a411},
replicationQueueScheduledExecutor=ScheduledExecutorFactoryConfiguration{fa
ctory=org.infinispan.executors.DefaultScheduledExecutorFactory@248a9705},
globalJmxStatistics=GlobalJmxStatisticsConfiguration{allowDuplicateDomains
=true, enabled=true, jmxDomain='jboss.infinispan',
mBeanServerLookup=org.jboss.as.clustering.infinispan.MBeanServerProvider@6
c0dc01, cacheManagerName='local', properties={}},
transport=TransportConfiguration{clusterName='ISPN', machineId='null',
rackId='null', siteId='null', strictPeerToPeer=false,
distributedSyncTimeout=240000, transport=null, nodeName='null',
properties={}},
serialization=SerializationConfiguration{advancedExternalizers=
{1100=org.infinispan.server.core.CacheValue$Externalizer@5fabc91d,
1101=org.infinispan.server.memcached.MemcachedValue$Externalizer@720bffd,
1104=org.infinispan.server.hotrod.ServerAddress$Externalizer@771c7eb2},
marshaller=org.infinispan.marshall.VersionAwareMarshaller@6fc21535,
version=52,
classResolver=org.jboss.marshalling.ModularClassResolver@2efe83e5},
shutdown=ShutdownConfiguration{hookBehavior=DONT_REGISTER}, modules={},
site=SiteConfiguration{localSite='null'}}
```

Report a bug

### 23.3.16. The locate Command

The **locate** command displays the physical location of a specified entry in a distributed cluster. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> locate a
[host/node1,host/node2]
```

Report a bug

### 23.3.17. The put Command

The **put** command inserts an entry into the cache. If a mapping exists for a key, the **put** command overwrites the old value. The CLI allows control over the type of data used to store the key and value. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b 100
[jmx://localhost:12000/MyCacheManager/namedCache]> put c 4139l
[jmx://localhost:12000/MyCacheManager/namedCache]> put d true
[jmx://localhost:12000/MyCacheManager/namedCache]> put e {
"package.MyClass": {"i": 5, "x": null, "b": true } }
```

Optionally, the **put** can specify a life span and maximum idle time value as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a expires 10s
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a expires 10m
maxidle 1m
```

Report a bug

### 23.3.18. The replace Command

The **replace** command replaces an existing entry in the cache with a specified new value. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> replace a b
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
```

```
b
[jmx://localhost:12000/MyCacheManager/namedCache]> replace a b c
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
c
[jmx://localhost:12000/MyCacheManager/namedCache]> replace a b d
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
c
```

Report a bug

### 23.3.19. The roles command

When authorization is enabled and the role mapper has been configured to be the **ClusterRoleMapper**, the principal to role mappings are stored within the cluster registry (a replicated cache available to all nodes). The **roles** command can be used to list the roles associated to a specific user, or to all users if one is not given:

```
[remoting://localhost:9999]> roles user1
[supervisor, reader]
```

**NOTE**

The **roles** command is only available to the JBoss Data Grid Server Mode CLI.

Report a bug

### 23.3.20. The rollback Command

The **rollback** command rolls back any changes made by an ongoing transaction. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> begin
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> rollback
```

Report a bug

### 23.3.21. The site Command

The **site** command performs administration tasks related to cross-datacenter replication. This command also retrieves information about the status of a site and toggles the status of a site. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> site --status NYC
online
[jmx://localhost:12000/MyCacheManager/namedCache]> site --offline NYC
ok
[jmx://localhost:12000/MyCacheManager/namedCache]> site --status NYC
offline
[jmx://localhost:12000/MyCacheManager/namedCache]> site --online NYC
```

Report a bug

### 23.3.22. The start Command

The **start** command initiates a batch of operations. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> start
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> end
```

Report a bug

### 23.3.23. The stats Command

The **stats** command displays statistics for the cache. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> stats
Statistics: {
  averageWriteTime: 143
  evictions: 10
  misses: 5
  hitRatio: 1.0
  readWriteRatio: 10.0
  removeMisses: 0
  timeSinceReset: 2123
  statisticsEnabled: true
  stores: 100
  elapsedTime: 93
  averageReadTime: 14
  removeHits: 0
  numberOfEntries: 100
  hits: 1000
}
LockManager: {
  concurrencyLevel: 1000
  numberOfLocksAvailable: 0
  numberOfLocksHeld: 0
}
```

Report a bug

### 23.3.24. The upgrade Command

The **upgrade** command implements the rolling upgrade procedure. For details about rolling upgrades, see the *Rolling Upgrades* chapter in the Red Hat JBoss Data Grid *Developer Guide*.

An example of the **upgrade** command's use is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> upgrade --
synchronize=hotrod --all
[jmx://localhost:12000/MyCacheManager/namedCache]> upgrade --
disconnectsource=hotrod --all
```

Report a bug

### 23.3.25. The version Command

The **`version`** command displays version information for the CLI client and server. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> version
Client Version 5.2.1.Final
Server Version 5.2.1.Final
```

Report a bug

# PART XII. OTHER RED HAT JBOSS DATA GRID FUNCTIONS

# CHAPTER 24. SET UP THE L1 CACHE

## 24.1. 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 only available when Red Hat JBoss Data Grid's cache mode is set to distribution. In other cache modes any configuration related to the L1 cache is ignored.

When caches are configured with distributed mode, the entries are evenly distributed between all clustered caches. Each entry is copied to a desired number of owners, which can be less than the total number of caches. As a result, the system's scalability is improved but also means that some entries are not available on all nodes and must be fetched from their owner node. In this situation, configure the Cache component to use the L1 Cache to temporarily store entries that it does not own to prevent repeated fetching for subsequent uses.

Each time a key is updated an invalidation message is generated. 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. Also, when the location of an entry changes in the cluster, the corresponding L1 cache entry is invalidated to prevent outdated cache entries.

Report a bug

## 24.2. L1 CACHE CONFIGURATION

### 24.2.1. L1 Cache Configuration (Library Mode)

The following sample configuration shows the L1 cache default values in Red Hat JBoss Data Grid's Library Mode.

**Example 24.1. L1 Cache Configuration in Library Mode**

```
<clustering mode="distribution">
 <sync/>
 <l1 enabled="true"
           lifespan="60000" />
</clustering>
```

The `l1` element configures the cache behavior in distributed cache instances. If used with non-distributed caches, this element is ignored.

- The ***enabled*** parameter enables the L1 cache.
- The ***lifespan*** parameter sets the maximum life span of an entry when it is placed in the L1 cache.

Report a bug

### 24.2.2. L1 Cache Configuration (Remote Client-Server Mode)

The following sample configuration shows the L1 cache default values in Red Hat JBoss Data Grid's Remote Client-Server mode.

**Example 24.2. L1 Cache Configuration for Remote Client-Server Mode**

```
<distributed-cache l1-lifespan="${VALUE}">
  ...
</distributed-cache>
```

The **`l1-lifespan`** element is added to a **`distributed-cache`** element to enable L1 caching and to set the life span of the L1 cache entries for the cache. This element is only valid for distributed caches.

If **`l1-lifespan`** is set to **`0`** or a negative number ( **`-1`**), L1 caching is disabled. L1 caching is enabled when the **`l1-lifespan`** value is greater than **`0`**.

**NOTE**

When the cache is accessed remotely via the Hot Rod protocol, the client accesses the owner node directly. Therefore, using L1 Cache in this situation does not offer any performance improvement and is not recommended. Other remote clients (Memcached, REST) cannot target the owner, therefore, using L1 Cache may increase the performance (at the cost of higher memory consumption).

**NOTE**

In Remote Client-Server mode, the L1 cache was enabled by default when distributed cache was used, even if the ***`l1-lifespan`*** attribute is not set. The default lifespan value was 10 minutes. Since JBoss Data Grid 6.3, the default lifespan is 0 which disables the L1 cache. Set a non-zero value for the ***`l1-lifespan`*** parameter to enable the L1 cache.

Report a bug

# CHAPTER 25. SET UP TRANSACTIONS

## 25.1. ABOUT TRANSACTIONS

A transaction consists of a collection of interdependent or related operations or tasks. All operations within a single transaction must succeed for the overall success of the transaction. If any operations within a transaction fail, the transaction as a whole fails and rolls back any changes. Transactions are particularly useful when dealing with a series of changes as part of a larger operation.

In Red Hat JBoss Data Grid, transactions are only available in Library mode.

Report a bug

### 25.1.1. About the Transaction Manager

In Red Hat JBoss Data Grid, the Transaction Manager coordinates transactions across a single or multiple resources. The responsibilities of a Transaction Manager include:

- initiating and concluding transactions
- managing information about each transaction
- coordinating transactions as they operate over multiple resources
- recovering from a failed transaction by rolling back changes

Report a bug

### 25.1.2. XA Resources and Synchronizations

XA Resources are fully fledged transaction participants. In the prepare phase, the XA Resource returns a vote with either the value **`OK`** or **`ABORT`**. If the Transaction Manager receives **`OK`** votes from all XA Resources, the transaction is committed, otherwise it is rolled back.

Synchronizations are a type of listener that receive notifications about events leading to the transaction life cycle. Synchronizations receive an event before and after the operation completes.

Unless recovery is required, it is not necessary to register as a full XA resource. An advantage of synchronizations is that they allow the Transaction Manager to optimize 2PC (Two Phase Commit) with a 1PC (One Phase Commit) where only one other resource is enlisted with that transaction (last resource commit optimization). This makes registering a synchronization more efficient.

However, if the operation fails in the prepare phase within Red Hat JBoss Data Grid, the transaction is not rolled back and if there are more participants in the transaction, they can ignore this failure and commit. Additionally, errors encountered in the commit phase are not propagated to the application code that commits the transaction.

By default JBoss Data Grid registers to the transaction as a synchronization.

Report a bug

### 25.1.3. Optimistic and Pessimistic Transactions

Pessimistic transactions acquire the locks when the first write operation on the key executes. After the key is locked, no other transaction can modify the key until this transaction is committed or rolled back. It is up to the application code to acquire the locks in correct order to prevent deadlocks.

With optimistic transactions locks are acquired at transaction prepare time and are held until the transaction commits (or rolls back). Also, Red Hat JBoss Data Grid sorts keys for all entries modified within a transaction automatically, preventing any deadlocks occurring due to the incorrect order of keys being locked. This results in:

- less messages being sent during the transaction execution
- locks held for shorter periods
- improved throughput

**NOTE**

Read operations never acquire any locks. Acquiring the lock for a read operation on demand is possible only with pessimistic transactions, using the **`FORCE_WRITE_LOCK`** flag with the operation.

Report a bug

### 25.1.4. Write Skew Checks

A common use case for entries is that they are read and subsequently written in a transaction. However, a third transaction can modify the entry between these two operations. In order to detect such a situation and roll back a transaction Red Hat JBoss Data Grid offers entry versioning and write skew checks. If the modified version is not the same as when it was last read during the transaction, the write skew checks throws an exception and the transaction is rolled back.

Enabling write skew check requires the **REPEATABLE_READ** isolation level. Also, in clustered mode (distributed or replicated modes), set up entry versioning. For local mode, entry versioning is not required.

**IMPORTANT**

With optimistic transactions, write skew checks are required for (atomic) conditional operations.

Report a bug

### 25.1.5. Transactions Spanning Multiple Cache Instances

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

Report a bug

## 25.2. CONFIGURE TRANSACTIONS

### 25.2.1. Configure Transactions (Library Mode)

In Red Hat JBoss Data Grid, transactions in Library mode can be configured with synchronization and transaction recovery. Transactions in their entirety (which includes synchronization and transaction recovery) are not available in Remote Client-Server mode.

In Library mode, transactions are configured as follows:

**Procedure 25.1. Configure Transactions in Library Mode (XML Configuration)**

1. **Set the Transaction Mode**
   See the table below this procedure for a list of available lookup classes.

```
<namedCache ...>
 <transaction transactionMode="{TRANSACTIONAL,NON_TRANSACTIONAL}">
        ...
</namedCache>
```

2. **Configure the Transaction Manager**
   The ***transactionMode*** element configures whether or not the cache is transactional.

```
<namedCache ...>
 <transaction transactionMode="TRANSACTIONAL"
       transactionManagerLookupClass="
{TransactionManagerLookupClass}">
</namedCache>
```

3. **Configure Locking Mode**
   The ***lockingMode*** parameter determines if the optimistic or pessimistic locking method is used. If the cache is non-transactional, the locking mode is ignored. The default value for this parameter is **OPTIMISTIC**.

```
<namedCache ...>
 <transaction transactionMode="TRANSACTIONAL"
       transactionManagerLookupClass="
{TransactionManagerLookupClass}"
       lockingMode="{OPTIMISTIC,PESSIMISTIC}">
</namedCache>
```

4. **Specify Synchronization**
   The **useSynchronization** element configures the cache to register a synchronization with the transaction manager, or register itself as an XA resource. The default value for this element is **true** (use synchronization).

```
<namedCache ...>
 <transaction transactionMode="TRANSACTIONAL"
       transactionManagerLookupClass="
{TransactionManagerLookupClass}"
       lockingMode="{OPTIMISTIC,PESSIMISTIC}"
       useSynchronization="{true,false}">
</namedCache>
```

5. **Configure Recovery**

The **recovery** element enables recovery for the cache when set to **true**.

The ***recoveryInfoCacheName*** parameter sets the name of the cache where recovery information is held. The default name of the cache is **__recoveryInfoCacheName__**.

```
<namedCache ...>
 <transaction transactionMode="TRANSACTIONAL"
       transactionManagerLookupClass="
{TransactionManagerLookupClass}"
       lockingMode="{OPTIMISTIC,PESSIMISTIC}"
       useSynchronization="{true,false}">
  <recovery enabled="true"
     recoveryInfoCacheName="{CacheName}" />
</namedCache>
```

6. **Configure the Write Skew Check**
   The **writeSkew** check determines if a modification to the entry from a different transaction should roll back the transaction. Write skew set to **true** requires ***isolation_level*** set to **REPEATABLE_READ**. The default value for ***writeSkew*** and **isolation_level** are **false** and **READ_COMMITTED** respectively.

```
<namedCache ...>
 <transaction ...>
 <locking isolation_level="{READ_COMMITTED,REPEATABLE_READ}"
    writeSkew="{true,false}" />
        ...
</namedCache>
```

7. **Configure Entry Versioning**
   For clustered caches, enable write skew check by enabling entry versioning and setting its value to **SIMPLE**.

```
<namedCache ...>
 <transaction ...>
 <locking ...>
 <versioning enabled="{true,false}"
      versioningScheme="{NONE|SIMPLE}"/>
        ...
</namedCache>
```

**Procedure 25.2. Configure Transactions in Library Mode (Programmatic Configuration)**

1. **Set the Transaction Mode**
   Set the transaction mode as follows:

```
Configuration config = new ConfigurationBuilder()/* ...
*/.transaction()
        .transactionMode(TransactionMode.TRANSACTIONAL);
```

2. **Configure the Transaction Manager**
   See the table below this procedure for a list of available lookup classes.

```
Configuration config = new ConfigurationBuilder()/* ...
```

```
*/.transaction()
        .transactionMode(TransactionMode.TRANSACTIONAL)
        .transactionManagerLookup(new
GenericTransactionManagerLookup());
```

3. **Configure Locking Mode**
   The **`lockingMode`** value determines whether optimistic or pessimistic locking is used. If the cache is non-transactional, the locking mode is ignored. The default value is **`OPTIMISTIC`**.

```
Configuration config = new ConfigurationBuilder()/* ...
*/.transaction()
        .transactionMode(TransactionMode.TRANSACTIONAL)
        .transactionManagerLookup(new
GenericTransactionManagerLookup());
        .lockingMode(LockingMode.OPTIMISTIC);
```

4. **Specify Synchronization**
   The **`useSynchronization`** value configures the cache to register a synchronization with the transaction manager, or register itself as an XA resource. The default value is **`true`** (use synchronization).

```
Configuration config = new ConfigurationBuilder()/* ...
*/.transaction()
        .transactionMode(TransactionMode.TRANSACTIONAL)
        .transactionManagerLookup(new
GenericTransactionManagerLookup());
        .lockingMode(LockingMode.OPTIMISTIC)
        .useSynchronization(true);
```

5. **Configure Recovery**
   The ***`recovery`*** parameter enables recovery for the cache when set to **`true`**.

   The **`recoveryInfoCacheName`** sets the name of the cache where recovery information is held. The default name of the cache is specified by ***`RecoveryConfiguration.DEFAULT_RECOVERY_INFO_CACHE`***.

```
Configuration config = new ConfigurationBuilder()/* ...
*/.transaction()
        .transactionMode(TransactionMode.TRANSACTIONAL)
        .transactionManagerLookup(new
GenericTransactionManagerLookup());
        .lockingMode(LockingMode.OPTIMISTIC)
        .useSynchronization(true)
        .recovery()
            .recoveryInfoCacheName("anotherRecoveryCacheName");
```

6. **Configure Write Skew Check**
   The ***`writeSkew`*** check determines if a modification to the entry from a different transaction should roll back the transaction. Write skew set to **`true`** requires ***`isolation_level`*** set to **REPEATABLE_READ**. The default value for ***`writeSkew`*** and ***`isolation_level`*** are **`false`** and **READ_COMMITTED** respectively.

```
Configuration config = new ConfigurationBuilder()/* ... */.locking()
```

```
.isolationLevel(IsolationLevel.REPEATABLE_READ).writeSkewCheck(true)
;
```

7. **Configure Entry Versioning**
   For clustered caches, enable write skew check by enabling entry versioning and setting its value to **SIMPLE**.

```
Configuration config = new ConfigurationBuilder()/* ...
*/.versioning()
          .enable()
          .scheme(VersioningScheme.SIMPLE);
```

**Table 25.1. Transaction Manager Lookup Classes**

| Class Name | Details |
|---|---|
| org.infinispan.transaction.lookup.DummyTransactionManagerLookup | Used primarily for testing environments. 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. |
| org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup | The default transaction manager when Red Hat 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 **DummyTransactionManager**. |
| org.infinispan.transaction.lookup.GenericTransactionManagerLookup | GenericTransactionManagerLookup is used by default when no transaction lookup class is specified. This lookup class is recommended when using JBoss Data Grid with Java EE-compatible environment that provides a TransactionManager interface, and is capable of locating the Transaction Manager in most Java EE application servers. If no transaction manager is located, it defaults to **DummyTransactionManager**. |
| org.infinispan.transaction.lookup.JBossTransactionManagerLookup | The **JbossTransactionManagerLookup** finds the standard transaction manager running in the application server. This lookup class uses JNDI to look up the TransactionManager instance, and is recommended when custom caches are being used in JTA transactions. |

Report a bug

### 25.2.2. Configure Transactions (Remote Client-Server Mode)

Red Hat JBoss Data Grid does not offer transactions in Remote Client-Server mode. The default and only supported configuration is non-transactional, which is set as follows:

**Example 25.1. Transaction Configuration in Remote Client-Server Mode**

```
<cache>
 ...
  <transaction mode="NONE" />
 ...
</cache>
```

**IMPORTANT**

In Remote Client-Server mode, the transactions element is set to **NONE** unless JBoss Data Grid is used in a compatibility mode where the cluster contains both JBoss Data Grid server and library instances. In this case, if transactions are configured in the library mode instance, they must also be configured in the server instance.

Report a bug

## 25.3. TRANSACTION RECOVERY

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

JBoss Data Grid uses JMX for operations that explicitly force transactions to commit or roll back. 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

### 25.3.1. Transaction Recovery Process

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

**Procedure 25.3. 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

### 25.3.2. 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 Red Hat JBoss Data Grid.

**Example 25.2. Money Transfer from an Account Stored in a Database to an Account in JBoss Data Grid**

1. The `TransactionManager.commit()` method is invoked 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.

**NOTE**

To use JMX to manage transaction recoveries, JMX support must be explicitly enabled.

Report a bug

## 25.4. 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.

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

Report a bug

### 25.4.1. Enable Deadlock Detection

Deadlock detection in Red Hat 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="100"/>
```

The ***`spinDuration`*** attribute defines how often lock acquisition is attempted within the maximum time allowed to acquire a particular lock (in milliseconds).

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.

Report a bug

# CHAPTER 26. CONFIGURE JGROUPS

JGroups is the underlying group communication library used to connect Red Hat JBoss Data Grid instances.

Report a bug

## 26.1. CONFIGURE RED HAT JBOSS DATA GRID INTERFACE BINDING (REMOTE CLIENT-SERVER MODE)

### 26.1.1. Interfaces

Red Hat JBoss Data Grid allows users to specify an interface type rather than a specific (unknown) IP address.

- ***link-local***: Uses a **169.*x*.*x*.*x*** or **254.*x*.*x*.*x*** address. This suits the traffic within one box.

```
<interfaces>
    <interface name="link-local">
        <link-local-address/>
    </interface>
    ...
</interfaces>
```

- ***site-local***: Uses a private IP address, for example **192.168.*x*.*x***. This prevents extra bandwidth charged from GoGrid, and similar providers.

```
<interfaces>
    <interface name="site-local">
        <site-local-address/>
    </interface>
    ...
</interfaces>
```

- ***global***: Picks a public IP address. This should be avoided for replication traffic.

```
<interfaces>
    <interface name="global">
        <any-address/>
    </interface>
    ...
</interfaces>
```

- ***non-loopback***: Uses the first address found on an active interface that is not a **127.*x*.*x*.*x*** address.

```
<interfaces>
    <interface name="non-loopback">
        <not>
     <loopback />
```

```
         </not>
      </interface>
   </interfaces>
```

Report a bug

## 26.1.2. Binding Sockets

Socket bindings provide a named the combination of interface and port. Sockets can be bound to the interface either individually or using a socket binding group.

Report a bug

### 26.1.2.1. 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.

**Example 26.1. Socket Binding**

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

Report a bug

### 26.1.2.2. 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:

**Example 26.2. Bind a Group**

```
<socket-binding-group name="ha-sockets" default-interface="global">
 ...
 <socket-binding name="jgroups-tcp" port="7600"/>
 <socket-binding name="jgroups-tcp-fd" port="57600"/>
 ...
</socket-binding-group>
```

The two sample socket bindings in the example are bound to the same ***default-interface*** (**global**), therefore the interface attribute does not need to be specified.

Report a bug

## 26.1.3. Configure JGroups Socket Binding

Each JGroups stack, configured in the JGroups subsystem, uses a specific socket binding. Set up the socket binding as follows:

**Example 26.3. JGroups Socket Binding Configuration**

```
<subsystem xmlns="urn:jboss:domain:jgroups:1.2" default-stack="udp">
    <stack name="udp">
        <transport type="UDP" socket-binding="jgroups-udp">
            ...
        </transport>
        <!-- rest of protocols -->
    </stack>
</subsystem>
```

**IMPORTANT**

When using UDP as the JGroups transport, the socket binding has to specify the regular (unicast) port, multicast address, and multicast port.

Report a bug

## 26.2. CONFIGURE JGROUPS (LIBRARY MODE)

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

**Example 26.4. JGroups Programmatic Configuration**

```
GlobalConfiguration gc = new GlobalConfigurationBuilder()
  .transport()
  .defaultTransport()
  .addProperty("configurationFile","jgroups.xml")
  .build();
```

**Example 26.5. JGroups XML Configuration**

```
<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

### 26.2.1. JGroups Transport Protocols

A transport protocol is the protocol at the bottom of a protocol stack. Transport Protocols are responsible for sending and receiving messages from the network.

Red Hat JBoss Data Grid ships with both UDP and TCP transport protocols.

Report a bug

#### 26.2.1.1. The UDP Transport Protocol

UDP is a transport protocol that uses:

- IP multicasting to send messages to all members of a cluster.
- UDP datagrams for unicast messages, which are sent to a single member.

When the UDP transport is started, it opens a unicast socket and a multicast socket. The unicast socket is used to send and receive unicast messages, the multicast socket sends and receives multicast sockets. The physical address of the channel with be the same as the address and port number of the unicast socket.

Report a bug

#### 26.2.1.2. The TCP Transport Protocol

TCP/IP is a replacement transport for UDP in situations where IP multicast cannot be used, such as operations over a WAN where routers may discard IP multicast packets.

TCP is a transport protocol used to send unicast and multicast messages.

- When sending multicast messages, TCP sends multiple unicast messages.
- When using TCP, each message to all cluster members is sent as multiple unicast messages, or one to each member.

As IP multicasting cannot be used to discover initial members, another mechanism must be used to find initial membership.

Red Hat JBoss Data Grid's Hot Rod is a custom TCP client/server protocol.

Report a bug

#### 26.2.1.3. Using the TCPPing Protocol

Some networks only allow TCP to be used. The pre-configured `jgroups-tcp.xml` includes the **`MPING`** protocol, which uses **`UDP`** multicast for discovery. When **`UDP`** multicast is not available, the **`MPING`** protocol, has to be replaced by a different mechanism. The recommended alternative is the **`TCPPING`** protocol. The **`TCPPING`** configuration contains a static list of IP addresses which are contacted for node discovery.

**Example 26.6. Configure the JGroups Subsystem to Use `TCPPING`**

```
<TCP bind_port="7800" />
<TCPPING timeout="3000"
```

```
initial_hosts="${jgroups.tcpping.initial_hosts:HostA[7800],HostB[7801]}"
          port_range="1"
          num_initial_members="3"/>
```

Report a bug

### 26.2.2. Pre-Configured JGroups Files

Red Hat 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`**
- **`jgroups-ec2.xml`**

Report a bug

#### 26.2.2.1. jgroups-udp.xml

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

- uses UDP as a transport and UDP multicast for discovery.
- is suitable for large clusters (over 8 nodes).
- is suitable if using Invalidation or Replication modes.

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 26.1. 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

#### 26.2.2.2. jgroups-tcp.xml

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

- uses TCP as a transport and UDP multicast for discovery.
- is generally only used where multicast UDP is not an option.
- TCP does not perform as well as UDP for clusters of eight or more nodes. Clusters of four nodes or fewer result in roughly the same level of performance for both UDP and TCP.

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 26.2. 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

#### 26.2.2.3. jgroups-ec2.xml

`jgroups-ec2.xml` is a pre-configured JGroups file in Red Hat JBoss Data Grid. The `jgroups-ec2.xml` configuration

- uses TCP as a transport and S3_PING for discovery.
- is suitable on Amazon EC2 nodes where UDP multicast isn't available.

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 26.3. jgroups-ec2.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.s3.access_key | The Amazon S3 access key used to access an S3 bucket | | Yes |
| jgroups.s3.secret_access_key | The Amazon S3 secret key used to access an S3 bucket | | Yes |
| jgroups.s3.bucket | Name of the Amazon S3 bucket to use. Must be unique and must already exist | | Yes |
| jgroups.s3.pre_signed_delete_url | The pre-signed URL to be used for the DELETE operation. | | Yes |
| jgroups.s3.pre_signed_put_url | The pre-signed URL to be used for the PUT operation. | | Yes |
| jgroups.s3.prefix | If set, S3_PING searches for a bucket with a name that starts with the prefix value. | | No |

Report a bug

## 26.3. TEST MULTICAST USING JGROUPS

Learn how to ensure that the system has correctly configured multicasting within the cluster.

Report a bug

### 26.3.1. Testing With Different Red Hat JBoss Data Grid Versions

The following table details which Red Hat JBoss Data Grid versions are compatible with this multicast test:

**Table 26.4. Testing with Different JBoss Data Grid Versions**

| Version | Test Case | Details |
|---|---|---|
| JBoss Data Grid 6.0.0 | Not Available | This version of JBoss Data Grid is based on JBoss Enterprise Application Server 6.0, which does not include the test classes used for this test. |
| JBoss Data Grid 6.0.1 | Not Available | This version of JBoss Data Grid is based on JBoss Enterprise Application Platform 6.0, which does not include the test classes used for this test. |
| JBoss Data Grid 6.1.0 | Available | This version of JBoss Data Grid is based on JBoss Enterprise Application Platform 6.0.1, but contains a newer version of the JGroups JAR file than the JAR file included in JBoss Enterprise Application Platform. As a result, the test class required for this test is available in the **`$JBOSS_HOME/modules/org/jgroups/main`** directory for JBoss Data Grid 6.1. |
| JBoss Data Grid 6.2.0 | Available | This version of JBoss Data Grid is based on JBoss Enterprise Application Platform 6.1. The JAR file is named according to the version, for example **`jgroups-3.2.7.Final-redhat-1.jar`** and is available in the **`$JBOSS_HOME/modules/system/layers/base/org/jgroups/main`** directory. |

| Version | Test Case | Details |
|---|---|---|
| JBoss Data Grid 6.2.1 | Available | This version of JBoss Data Grid is based on JBoss Enterprise Application Plat form 6.1.1 The JAR file is named according to the version (**`jgroups-3.4.3.Final-redhat-1.jar`** and is available in the **`$JBOSS_HOME/modules/system/layers/base/org/jgroups/main`** directory. |
| JBoss Data Grid 6.3.0 | Available | This version of JBoss Data Grid is based on JBoss Enterprise Application Plat form 6.2.4. The JAR file is named according to the version (**`jgroups-3.4.4.Final-redhat-5.jar`** and is available in the **`$JBOSS_HOME/modules/system/layers/base/org/jgroups/main`** directory. |

Report a bug

### 26.3.2. Testing Multicast Using JGroups

The following procedure details the steps to test multicast using JGroups if you are using Red Hat JBoss Data Grid :

#### Prerequisites

Ensure that the following prerequisites are met before starting the testing procedure.

1. Set the ***`bind_addr`*** value to the appropriate IP address for the instance.
2. For added accuracy, set ***`mcast_addr`*** and ***`port`*** values that are the same as the cluster communication values.
3. Start two command line terminal windows. Navigate to the location of the JGroups JAR file for one of the two nodes in the first terminal and the same location for the second node in the second terminal.

**Procedure 26.1. Test Multicast Using JGroups**

1. **Run the Multicast Server on Node One**
   Run the following command on the command line terminal for the first node:

```
java -cp jgroups.jar org.jgroups.tests.McastReceiverTest -mcast_addr
230.1.2.3 -port 5555 -bind_addr $YOUR_BIND_ADDRESS
```

2. **Run the Multicast Server on Node Two**
   Run the following command on the command line terminal for the second node:

   ```
   java -cp jgroups.jar org.jgroups.tests.McastSenderTest -mcast_addr
   230.1.2.3 -port 5555 -bind_addr $YOUR_BIND_ADDRESS
   ```

3. **Transmit Information Packets**
   Enter information on instance for node two (the node sending packets) and press enter to send the information.

4. **View Receives Information Packets**
   View the information received on the node one instance. The information entered in the previous step should appear here.

5. **Confirm Information Transfer**
   Repeat steps 3 and 4 to confirm all transmitted information is received without dropped packets.

6. **Repeat Test for Other Instances**
   Repeat steps 1 to 4 for each combination of sender and receiver. Repeating the test identifies other instances that are incorrectly configured.

**Result**

All information packets transmitted from the sender node must appear on the receiver node. If the sent information does not appear as expected, multicast is incorrectly configured in the operating system or the network.

Report a bug

# CHAPTER 27. USE RED HAT DATA GRID WITH AMAZON WEB SERVICES

## 27.1. THE S3_PING JGROUPS DISCOVERY PROTOCOL

**`S3_PING`** is a discovery protocol that is ideal for use with Amazon's Elastic Compute Cloud (EC2) because EC2 does not allow multicast and therefore **`MPING`** is not allowed.

Each EC2 instance adds a small file to an S3 data container, known as a bucket. Each instance then reads the files in the bucket to discover the other members of the cluster.

Report a bug

## 27.2. S3_PING CONFIGURATION OPTIONS

Red Hat JBoss Data Grid works with Amazon Web Services in two ways:

- In Library mode, use JGroups' **`jgroups-ec2.xml`** file (see Section 26.2.2.3, "jgroups-ec2.xml" for details) or use the **`S3_PING`** protocol.
- In Remote Client-Server mode, use JGroups' **`S3_PING`** protocol.

In Library and Remote Client-Server mode, there are three ways to configure the **`S3_PING`** protocol for clustering to work in Amazon AWS:

- Use Private S3 Buckets. These buckets use Amazon AWS credentials.
- Use Pre-Signed URLs. These pre-assigned URLs are assigned to buckets with private write and public read rights.
- Use Public S3 Buckets. These buckets do not have any credentials.

Report a bug

### 27.2.1. Using Private S3 Buckets

This configuration requires access to a private bucket that can only be accessed with the appropriate AWS credentials. To confirm that the appropriate permissions are available, confirm that the user has the following permissions for the bucket:

- List
- Upload/Delete
- View Permissions
- Edit Permissions

Ensure that the **`S3_PING`** configuration includes the following properties:

- either the ***`location`*** or the ***`prefix`*** property to specify the bucket, but not both. If the ***`prefix`*** property is set, **`S3_PING`** searches for a bucket with a name that starts with the prefix value. If a bucket with the prefix at the beginning of the name is found, **`S3_PING`** uses that bucket. If a bucket with the prefix is not found, **S3_PING** creates a bucket using the AWS credentials and names it based on the prefix and a UUID (the naming format is *{prefix value}-{UUID}*).

- the ***access_key*** and ***secret_access_key*** properties for the AWS user.

**NOTE**

If a **403** error displays when using this configuration, verify that the properties have the correct values. If the problem persists, confirm that the system time in the EC2 node is correct. Amazon S3 rejects requests with a time stamp that is more than **15** minutes old compared to their server's times for security purposes.

**Example 27.1. Start the Red Hat JBoss Data Grid Server with a Private Bucket**

Run the following command from the top level of the server directory to start the Red Hat JBoss Data Grid server using a private S3 bucket:

```
bin/clustered.sh -Djboss.bind.address={server_ip_address} -
Djboss.bind.address.management={server_ip_address} -
Djboss.default.jgroups.stack=s3 -Djgroups.s3.bucket={s3_bucket_name} -
Djgroups.s3.access_key={access_key} -
Djgroups.s3.secret_access_key={secret_access_key}
```

1. Replace *{server_ip_address}* with the server's IP address.
2. Replace *{s3_bucket_name}* with the appropriate bucket name.
3. Replace *{access_key}* with the user's access key.
4. Replace *{secret_access_key}* with the user's secret access key.

Report a bug

## 27.2.2. Using Pre-Signed URLs

For this configuration, create a publically readable bucket in S3 by setting the **List** permissions to **Everyone** to allow public read access. Each node in the cluster generates a pre-signed URL for put and delete operations, as required by the **S3_PING** protocol. This URL points to a unique file and can include a folder path within the bucket.

**NOTE**

Longer paths will cause errors in **S3_PING**. For example, a path such as **my_bucket/DemoCluster/node1** works while a longer path such as **my_bucket/Demo/Cluster/node1** will not.

Report a bug

### 27.2.2.1. Generating Pre-Signed URLs

JGroup's **S3_PING** class includes a utility method to generate pre-signed URLs. The last argument for this method is the time when the URL expires expressed in the number of seconds since the Unix epoch (January 1, 1970).

The syntax to generate a pre-signed URL is as follows:

```
String Url = S3_PING.generatePreSignedUrl("{access_key}",
"{secret_access_key}", "{operation}", "{bucket_name}", "{path}",
{seconds});
```

1. Replace *{operation}* with either **PUT** or **DELETE**.
2. Replace *{access_key}* with the user's access key.
3. Replace *{secret_access_key}* with the user's secret access key.
4. Replace *{bucket_name}* with the name of the bucket.
5. Replace *{path}* with the desired path to the file within the bucket.
6. Replace *{seconds}* with the number of seconds since the Unix epoch (January 1, 1970) that the path remains valid.

**Example 27.2. Generate a Pre-Signed URL**

```
String putUrl = S3_PING.generatePreSignedUrl("access_key",
"secret_access_key", "put", "my_bucket", "DemoCluster/node1",
1234567890);
```

Ensure that the **S3_PING** configuration includes the ***pre_signed_put_url*** and ***pre_signed_delete_url*** properties generated by the call to **S3_PING.generatePreSignedUrl()**. This configuration is more secure than one using private S3 buckets, because the AWS credentials are not stored on each node in the cluster

**NOTE**

If a pre-signed URL is entered into an XML file, then the & characters in the URL must be replaced with its XML entity (**&amp;**).

Report a bug

#### 27.2.2.2. Set Pre-Signed URLs Using the Command Line

To set the pre-signed URLs using the command line, use the following guidelines:

- Enclose the URL in double quotation marks (**""**).
- In the URL, each occurrence of the ampersand (&) character must be escaped with a backslash (\)

**Example 27.3. Start a JBoss Data Grid Server with a Pre-Signed URL**

```
bin/clustered.sh -Djboss.bind.address={server_ip_address} -
Djboss.bind.address.management={server_ip_address} -
Djboss.default.jgroups.stack=s3 -
Djgroups.s3.pre_signed_put_url="http://{s3_bucket_name}.s3.amazonaws.com
/ node1?
AWSAccessKeyId={access_key}\&Expires={expiration_time}\&Signature={signa
ture}"-
Djgroups.s3.pre_signed_delete_url="http://{s3_bucket_name}.s3.amazonaws.
com/ node1?
AWSAccessKeyId={access_key}\&Expires={expiration_time}\&Signature={signa
ture}"
```

In the provided example, the ***{signatures}*** values are generated by the **`S3_PING.generatePreSignedUrl()`** method. Additionally, the *{expiration_time}* values are the expiration time for the URL that are passed into the **`S3_PING.generatePreSignedUrl()`** method.

Report a bug

### 27.2.3. Using Public S3 Buckets

This configuration involves an S3 bucket that has public read and write permissions, which means that **`Everyone`** has permissions to **`List`**, **`Upload/Delete`**, **`View Permissions`**, and **`Edit Permissions`** for the bucket.

The ***location*** property must be specified with the bucket name for this configuration. This configuration method is the least secure because any user who knows the name of the bucket can upload and store data in the bucket and the bucket creator's account is charged for this data.

To start the Red Hat JBoss Data Grid server, use the following command:

```
bin/clustered.sh -Djboss.bind.address={server_ip_address} -
Djboss.bind.address.management={server_ip_address} -
Djboss.default.jgroups.stack=s3 -Djgroups.s3.bucket={s3_bucket_name}
```

Report a bug

### 27.2.4. Troubleshooting S3_PING Warnings

Depending on the **`S3_PING`** configuration type used, the following warnings may appear when starting the JBoss Data Grid Server:

```
15:46:03,468 WARN  [org.jgroups.conf.ProtocolConfiguration] (MSC service
thread 1-7) variable "${jgroups.s3.pre_signed_put_url}" in S3_PING could
not be substituted; pre_signed_put_url is removed from properties
```

```
15:46:03,469 WARN  [org.jgroups.conf.ProtocolConfiguration] (MSC service
thread 1-7) variable "${jgroups.s3.prefix}" in S3_PING could not be
substituted; prefix is removed from properties
```

```
15:46:03,469 WARN  [org.jgroups.conf.ProtocolConfiguration] (MSC service
thread 1-7) variable "${jgroups.s3.pre_signed_delete_url}" in S3_PING
could not be substituted; pre_signed_delete_url is removed from properties
```

-

In each case, ensure that the property listed as missing in the warning is not needed by the **S3_PING** configuration.

Report a bug

# CHAPTER 28. HIGH AVAILABILITY USING SERVER HINTING

In Red Hat JBoss Data Grid, Server Hinting ensures that backed up 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.

Data distribution across nodes is controlled by the Consistent Hashing mechanism. JBoss Data Grid offers a pluggable policy to specify the consistent hashing algorithm. For details see Section 28.4, “ConsistentHashFactories”

Setting a ***`machineId`***, ***`rackId`***, or ***`siteId`*** in the transport configuration will trigger the use of **`TopologyAwareConsistentHashFactory`**, which is the equivalent of the **`DefaultConsistentHashFactory`** with Server Hinting enabled.

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

Report a bug

## 28.1. ESTABLISHING SERVER HINTING WITH JGROUPS

When setting up a clustered environment in Red Hat 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 26.2.2, “Pre-Configured JGroups Files”

Report a bug

## 28.2. CONFIGURE SERVER HINTING (REMOTE CLIENT-SERVER MODE)

In Red Hat JBoss Data Grid's Remote Client-Server mode, Server Hinting is configured in the JGroups subsystem on the `transport` element for the default stack, as follows:

**Procedure 28.1. Configure Server Hinting in Remote Client-Server Mode**

1. **Find the JGroups Subsystem Configuration**

```
<subsystem xmlns="urn:jboss:domain:jgroups:1.1"
    default-stack="${jboss.default.jgroups.stack:udp}">
 <stack name="udp">
```

2. **Enable Server Hinting via the `transport` Element**

   a. **Set the Site ID**

```
<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}">
```

b. **Set the Rack ID**

```
<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}">
```

c. **Set the Machine ID**

```
<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

## 28.3. CONFIGURE SERVER HINTING (LIBRARY MODE)

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

**Procedure 28.2. Configure Server Hinting for Library Mode**

The following configuration attributes are used to configure Server Hinting in JBoss Data Grid.

1. **Set the *clusterName* Attribute**
   The ***clusterName*** attribute specifies the name assigned to the cluster.

   ```
   <transport clusterName = "MyCluster" />
   ```

2. **Add the *machineId***
   The ***machineId*** attribute specifies the JVM instance that contains the original data. This is particularly useful for nodes with multiple JVMs and physical hosts with multiple virtual hosts.

   ```
   <transport clusterName = "MyCluster"
              machineId = "LinuxServer01" />
   ```

3. **Add the *rackId***

   The ***rackId*** parameter specifies the rack that contains the original data, so that other racks are used for backups.

   ```
   <transport clusterName = "MyCluster"
              machineId = "LinuxServer01"
              rackId = "Rack01" />
   ```

4. **Add the *siteId***

   The ***siteId*** parameter differentiates between nodes in different data centers replicating to each other.

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

The listed parameters are optional in a JBoss Data Grid configuration.

If ***machineId***, ***rackId***, or ***siteId*** are included in the configuration, **TopologyAwareConsistentHashFactory** is selected automatically, enabling Server Hinting. However, if Server Hinting is not configured, JBoss Data Grid's distribution algorithms are allowed to store replications in the same physical machine/rack/data center as the original data.

Report a bug

## 28.4. CONSISTENTHASHFACTORIES

Red Hat JBoss Data Grid offers a pluggable mechanism for selecting the consistent hashing algorithm. It is shipped with four implementations but a custom implementation can also be used.

JBoss Data Grid ships with four ConsistentHashFactory implementations:

- **DefaultConsistentHashFactory** - keeps segments balanced evenly across all the nodes, however the key mapping is not guaranteed to be same across caches,as this depends on the history of each cache.

- **SyncConsistentHashFactory** - guarantees that the key mapping is the same for each cache, provided the current membership is the same. This has a drawback in that a node joining the cache can cause the existing nodes to also exchange segments, resulting in either additional state transfer traffic, the distribution of the data becoming less even, or both.

- **TopologyAwareConsistentHashFactory** - equivalent of **DefaultConsistentHashFactory**, but with server hinting enabled.

- **TopologyAwareSyncConsistentHashFactory** - equivalent of **SyncConsistentHashFactory**, but with server hinting enabled.

The consistent hash implementation can be selected via the hash configuration:

```
<hash
consistentHashFactory="org.infinispan.distribution.ch.TopologyAwareSyncCon
sistentHashFactory"/>
```

This configuration guarantees caches with the same members have the same consistent hash, and if the ***machineId***, ***rackId***, or ***siteId*** attributes are specified in the transport configuration it also spreads backup copies across physical machines/racks/data centers.

It has a potential drawback in that it can move a greater number of segments than necessary during re-balancing. This can be mitigated by using a larger number of segments.

Another potential drawback is that the segments are not distributed as evenly as possible, and actually using a very large number of segments can make the distribution of segments worse.

Report a bug

### 28.4.1. Implementing a ConsistentHashFactory

A custom **ConsistentHashFactory** must implement the **org.infinispan.distribution.ch.ConsistenHashFactory** interface with the following methods (all of which return an implementation of **org.infinispan.distribution.ch.ConsistentHash**):

**Example 28.1. ConsistentHashFactory Methods**

```
create(Hash hashFunction, int numOwners, int numSegments, List<Address>
members,Map<Address, Float> capacityFactors)
updateMembers(ConsistentHash baseCH, List<Address> newMembers,
Map<Address,
Float> capacityFactors)
rebalance(ConsistentHash baseCH)
union(ConsistentHash ch1, ConsistentHash ch2)
```

Currently it is not possible to pass custom parameters to **ConsistentHashFactory** implementations.

Report a bug

## 28.5. KEY AFFINITY SERVICE

The key affinity service allows a value to be placed in a certain node in a distributed Red Hat JBoss Data Grid cluster. The service returns a key that is hashed to a particular node based on a supplied cluster address identifying it.

The keys returned by the key affinity service cannot hold any meaning, such as a username. These are only random identifiers that are used throughout the application for this record. The provided key generators do not guarantee that the keys returned by this service are unique. For custom key format, you can pass your own implementation of KeyGenerator.

The following is an example of how to obtain and use a reference to this service.

**Example 28.2. Key Affinity Service**

```
EmbeddedCacheManager cacheManager = getCacheManager();
Cache cache = cacheManager.getCache();
KeyAffinityService keyAffinityService =
        KeyAffinityServiceFactory.newLocalKeyAffinityService(
            cache,
            new RndKeyGenerator(),
            Executors.newSingleThreadExecutor(),
            100);
Object localKey =
keyAffinityService.getKeyForAddress(cacheManager.getAddress());
cache.put(localKey, "yourValue");
```

The following procedure is an explanation of the provided example.

**Procedure 28.3. Using the Key Affinity Service**

1. **Obtain a reference to a cache manager and cache**

   ```
   EmbeddedCacheManager cacheManager = getCacheManager();
   Cache cache = cacheManager.getCache();
   ```

2. **Create the affinity service**
   This starts the service, then uses the supplied ***Executor*** to generate and queue keys.

   ```
   KeyAffinityService keyAffinityService =
           KeyAffinityServiceFactory.newLocalKeyAffinityService(
               cache,
               new RndKeyGenerator(),
               Executors.newSingleThreadExecutor(),
               100);
   ```

3. **Obtain a key to be mapped to a certain address**
   Obtain a key from the service which will be mapped to the local node (**cacheManager.getAddress()** returns the local address).

   ```
   Object localKey =
   keyAffinityService.getKeyForAddress(cacheManager.getAddress());
   ```

4. **Put the value on the node**
   The entry with a key obtained from the **KeyAffinityService** is always stored on the node with the provided address. In this case, it is the local node.

   ```
   cache.put(localKey, "yourValue");
   ```

Report a bug

### 28.5.1. Lifecycle

**KeyAffinityService** extends ***Lifecycle***, which allows the key affinity service to be stopped, started, and restarted.

**Example 28.3. Key Affinity Service Lifecycle Parameter**

```
public interface Lifecycle {
     void start();
     void stop();
```

```
}
```

The service is instantiated through the **`KeyAffinityServiceFactory`**. All factory methods have an ***`Executor`***, that is used for asynchronous key generation, so that this does not occur in the caller's thread. The user controls the shutting down of this ***`Executor`***.

The **`KeyAffinityService`** must be explicitly stopped when it is no longer required. This stops the background key generation, and releases other held resources. The **`KeyAffinityServce`** will only stop itself when the cache manager with which it is registered is shut down.

Report a bug

### 28.5.2. Topology Changes

**`KeyAffinityService`** key ownership may change when a topology change occurs. The key affinity service monitors topology changes and updates so that it doesn't return stale keys, or keys that would map to a different node than the one specified. However, this does not guarantee that a node affinity hasn't changed when a key is used. For example:

1. Thread (**`T1`**) reads a key ( **`K1`**) that maps to a node ( **`A`**).
2. A topology change occurs, resulting in **`K1`** mapping to node **`B`**.
3. **`T1`** uses **`K1`** to add something to the cache. At this point, **`K1`** maps to **`B`**, a different node to the one requested at the time of read.

The above scenario is a not ideal, however it is a supported behavior for the application, as the keys that are already in use may be moved over during cluster change. The **`KeyAffinityService`** provides an access proximity optimization for stable clusters, which does not apply during the instability of topology changes.

Report a bug

# CHAPTER 29. SET UP CROSS-DATACENTER REPLICATION

In Red Hat JBoss Data Grid, Cross-Datacenter Replication allows the administrator to create data backups in multiple clusters. These clusters can be at the same physical location or different ones. JBoss Data Grid's Cross-Site Replication implementation is based on JGroups' **RELAY2** protocol.

Cross-Datacenter Replication ensures data redundancy across clusters. Ideally, each of these clusters must be in a different physical location than the others.

Report a bug

## 29.1. CROSS-DATACENTER REPLICATION OPERATIONS

Red Hat JBoss Data Grid's Cross-Datacenter Replication operation is explained through the use of an example, as follows:

**Example 29.1. Cross-Datacenter Replication Example**

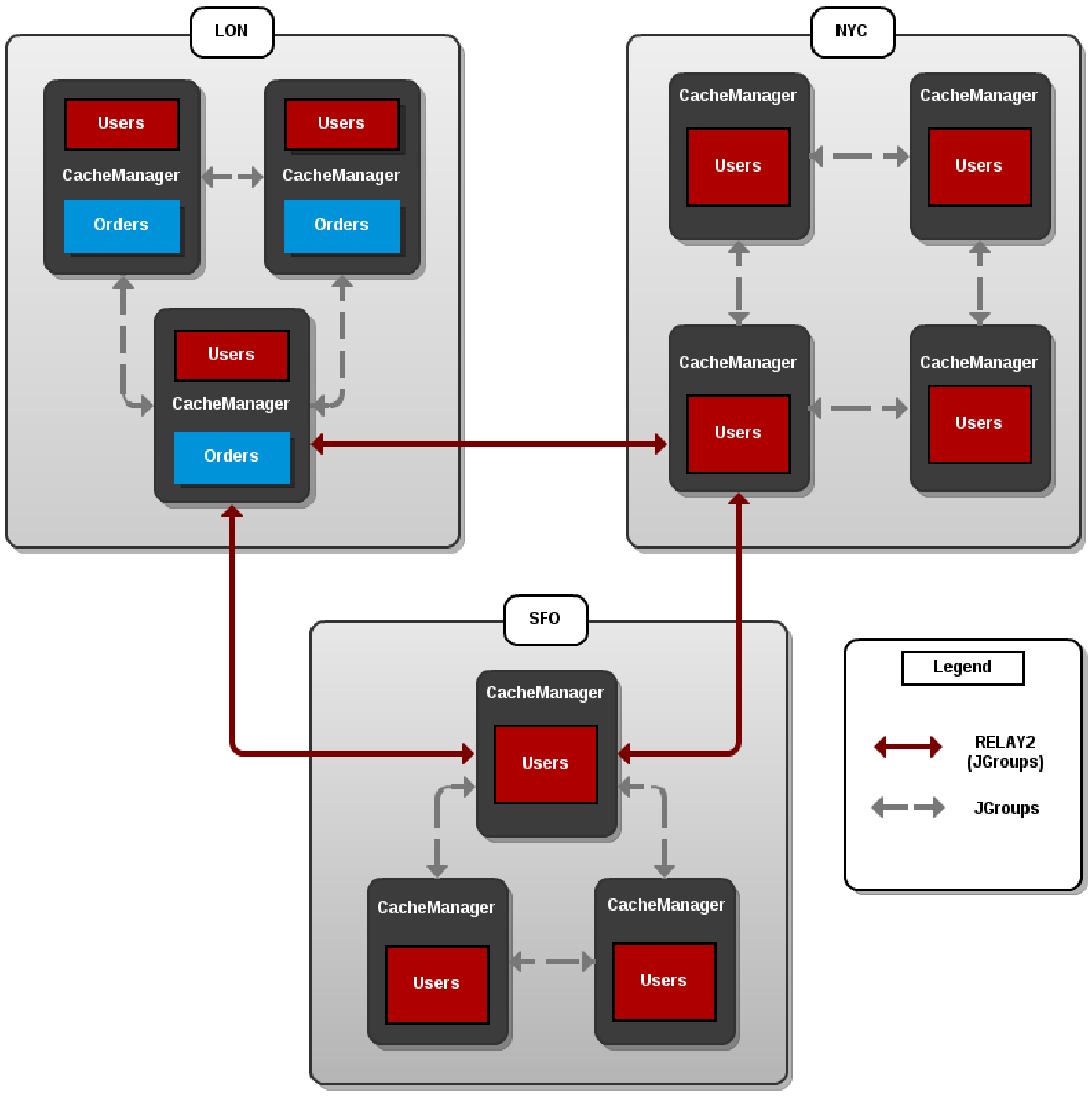


**Figure 29.1. Cross-Datacenter Replication Example**

Three sites are configured in this example: **LON**, **NYC** and **SFO**. Each site hosts a running JBoss Data Grid cluster made up of three to four physical nodes.

The `Users` cache is active in all three sites. Changes to the `Users` cache at the **LON** site is replicated at the other two sites. The `Orders` cache, however, is only available locally at the **LON** site because it is not replicated to the other sites.

The `Users` cache can use different replication mechanisms each site. For example, it can back up data synchronously to **SFO** and asynchronously to **NYC** and **LON**.

The `Users` cache can also have a different configuration from one site to another. For example, it can be configured as a distributed cache with *numOwners* set to **2** in the **LON** site, as a replicated cache in the **NYC** site and as a distributed cache with *numOwners* set to **1** in the **SFO** site.

JGroups is used for communication within each site as well as inter-site communication. Specifically, a JGroups protocol called **RELAY2** facilitates communication between sites. For more information, see Section C.4, "About RELAY2"

Report a bug

## 29.2. CONFIGURE CROSS-DATACENTER REPLICATION

### 29.2.1. Configure Cross-Datacenter Replication (Remote Client-Server Mode)

In Red Hat JBoss Data Grid's Remote Client-Server mode, cross-datacenter replication is set up as follows:

**Procedure 29.1. Set Up Cross-Datacenter Replication**

1. **Set Up RELAY**
   Add the following configuration to the `standalone.xml` file to set up **RELAY**:

```
<subsystem xmlns="urn:jboss:domain:jgroups:1.2"
    default-stack="udp">
 <stack name="udp">
  <transport type="UDP"
      socket-binding="jgroups-udp"/>
  ...

  <relay site="LON">
     <remote-site name="NYC" stack="tcp" cluster="global"/>
     <remote-site name="SFO" stack="tcp" cluster="global"/>
     <property name="relay_multicasts">false</property>
  </relay>
 </stack>
</subsystem>
```

   The **RELAY** protocol creates an additional stack (running parallel to the existing **TCP** stack) to communicate with the remote site. If a **TCP** based stack is used for the local cluster, two **TCP** based stack configurations are required: one for local communication and one to connect to the remote site. For an illustration, see Section 29.1, "Cross-Datacenter Replication Operations"

2. **Set Up Sites**
   Use the following configuration in the `standalone.xml` file to set up sites for each distributed cache in the cluster:

   ```
   <distributed-cache>
        ...
        <backups>
           <backup site="{FIRSTSITENAME}" strategy="{SYNC/ASYNC}" />
           <backup site="{SECONDSITENAME}" strategy="{SYNC/ASYNC}" />
        </backups>
   </distributed-cache>
   ```

3. **Configure Local Site Transport**
   Add the name of the local site in the `transport` element to configure transport:

   ```
   <transport executor="infinispan-transport"
              lock-timeout="60000"
              cluster="LON"
              stack="udp"/>
   ```

Report a bug

## 29.2.2. Configure Cross-Data Replication (Library Mode)

### 29.2.2.1. Configure Cross-Datacenter Replication Declaratively

When configuring Cross-Datacenter Replication, the `relay.RELAY2` protocol creates an additional stack (running parallel to the existing **TCP** stack) to communicate with the remote site. If a **TCP**-based stack is used for the local cluster, two **TCP** based stack configurations are required: one for local communication and one to connect to the remote site.

In JBoss Data Grid's Library mode, cross-datacenter replication is set up as follows:

**Procedure 29.2. Setting Up Cross-Datacenter Replication**

1. **Configure the Local Site**
   Add the `site` element to the `global` element to add the local site (in this example, the local site is named **LON**).

   ```
   <infinispan>
      <global>
         ...
         <site local="LON" />
         ...
      </global>
   </infinispan>
   ```

2. **Configure JGroups for the Local Site**
   Cross-site replication requires a non-default JGroups configuration. Add the `transport` element and set up the path to the configuration file as the ***`configurationFile`*** property. In this example, the JGroups configuration file is named `jgroups-with-relay.xml`.

```
<infinispan>
   <global>
      ...
      <site local="LON" />
      <transport clusterName="default">
         <properties>
             <property name="configurationFile" value="jgroups-
with-relay.xml" />
          </properties>
      </transport>
      ...
   </global>
</infinispan>
```

3. **Configure the LON Cache**
   Configure the cache in site **LON** to back up to the sites **NYC** and **SFO**:

```
<infinispan>
   <global>
      <site local="LON" />
      <transport clusterName="default">
         <properties>
             <property name="configurationFile" value="jgroups-
with-relay.xml" />
          </properties>
      </transport>
      ...
   </global>
   ...
   <namedCache name="lon">
      <sites>
         <backups>
            <backup site="NYC"
      strategy="SYNC"
      backupFailurePolicy="WARN" />
            <backup site="SFO"
      strategy="ASYNC"
      backupFailurePolicy="IGNORE"/>
         </backups>
      </sites>
   </namedCache>
</infinispan>
```

4. **Configure the Back Up Caches**

   a. Configure the cache in site **NYC** to receive back up data from **LON**:

```
<infinispan>
   <global>
      <site local="NYC" />
      <transport clusterName="default">
         <properties>
             <property name="configurationFile" value="jgroups-
with-relay.xml"/>
          </properties>
      </transport>
      ...
   </global>
   ...
   <namedCache name="lonBackup">
      <sites>
         <backupFor remoteSite="LON"
      remoteCache="lon" />
      </sites>
   </namedCache>
</infinispan>
```

b. Configure the cache in site **SFO** to receive back up data from **LON**:

```
<infinispan>
   <global>
      <site local="SFO" />
      <transport clusterName="default">
         <properties>
             <property name="configurationFile" value="jgroups-
with-relay.xml"/>
          </properties>
      </transport>
      ...
   </global>
   ...
   <namedCache name="lonBackup">
      <sites>
         <backupFor remoteSite="LON"
      remoteCache="lon" />
      </sites>
   </namedCache>
</infinispan>
```

5. **Add the Contents of the Configuration File**
   As a default, Red Hat JBoss Data Grid includes JGroups configuration files such as **`jgroups-tcp.xml`** and **`jgroups-udp.xml`** in the **`infinispan-core-{VERSION}.jar`** package.

   Copy the JGroups configuration to a new file (in this example, it is named **`jgroups-with-relay.xml`**) and add the provided configuration information to this file. Note that the **`relay.RELAY2`** protocol configuration must be the last protocol in the configuration stack.

```
<config>
    ...
    <relay.RELAY2 site="LON"
              config="relay.xml"
              relay_multicasts="false" />
</config>
```

6. **Configure the relay.xml File**
   Set up the **`relay.RELAY2`** configuration in the **`relay.xml`** file. This file describes the global cluster configuration.

```
<RelayConfiguration>
```

```
    <sites>
        <site name="LON"
              id="0">
            <bridges>
                <bridge config="jgroups-global.xml"
                        name="global"/>
                </bridges>
        </site>
        <site name="NYC"
              id="1">
            <bridges>
                <bridge config="jgroups-global.xml"
                        name="global"/>
                </bridges>
        </site>
        <site name="SFO"
              id="2">
            <bridges>
                <bridge config="jgroups-global.xml"
                        name="global"/>
            </bridges>
        </site>
    </sites>
</RelayConfiguration>
```

7. **Configure the Global Cluster**
   The file **jgroups-global.xml** referenced in **relay.xml** contains another JGroups configuration which is used for the global cluster: communication between sites.

   The global cluster configuration is usually **TCP**-based and uses the **TCPPING** protocol (instead of **PING** or **MPING**) to discover members. Copy the contents of **jgroups-tcp.xml** into **jgroups-global.xml** and add the following configuration in order to configure **TCPPING**:

```
<config>
    <TCP bind_port="7800" ... />
    <TCPPING
initial_hosts="lon.hostname[7800],nyc.hostname[7800],sfo.hostname[78
00]"
             num_initial_members="3"
             ergonomics="false" />
          <!-- Rest of the protocols -->
</config>
```

   Replace the hostnames (or IP addresses) in ***TCPPING.initial_hosts*** with those used for your site masters. The ports (**7800** in this example) must match the ***TCP.bind_port***.

   For more information about the **TCPPING** protocol, see Section 26.2.1.3, "Using the TCPPing Protocol"

Report a bug

#### 29.2.2.2. Configure Cross-Datacenter Replication Programmatically

The programmatic method to configure cross-datacenter replication in Red Hat JBoss Data Grid is as follows:

**Procedure 29.3. Configure Cross-Datacenter Replication Programmatically**

1. **Identify the Node Location**
   Declare the site the node resides in:

   ```
   globalConfiguration.site().localSite("LON");
   ```

2. **Configure JGroups**
   Configure JGroups to use the **RELAY** protocol:

   ```
   globalConfiguration.transport().addProperty("configurationFile",
   jgroups-with-relay.xml);
   ```

3. **Set Up the Remote Site**
   Set up JBoss Data Grid caches to replicate to the remote site:

   ```
   ConfigurationBuilder lon = new ConfigurationBuilder();
   lon.sites().addBackup()
         .site("NYC")
         .backupFailurePolicy(BackupFailurePolicy.WARN)
         .strategy(BackupConfiguration.BackupStrategy.SYNC)
         .replicationTimeout(12000)
         .sites().addInUseBackupSite("NYC")
       .sites().addBackup()
         .site("SFO")
         .backupFailurePolicy(BackupFailurePolicy.IGNORE)
         .strategy(BackupConfiguration.BackupStrategy.ASYNC)
         .sites().addInUseBackupSite("SFO")
   ```

4. **Optional: Configure the Backup Caches**
   JBoss Data Grid implicitly replicates data to a cache with same name as the remote site. If a backup cache on the remote site has a different name, users must specify a ***backupFor*** cache to ensure data is replicated to the correct cache.

   **NOTE**

   This step is optional and only required if the remote site's caches are named differently from the original caches.

   a. Configure the cache in site **NYC** to receive backup data from **LON**:

   ```
   ConfigurationBuilder NYCbackupOfLon = new ConfigurationBuilder();
   lonBackup.sites().backupFor().remoteCache("lon").remoteSite("LON"
   );
   ```

   b. Configure the cache in site **SFO** to receive backup data from **LON**:

   ```
   ConfigurationBuilder SFObackupOfLon = new ConfigurationBuilder();
   lonBackup.sites().backupFor().remoteCache("lon").remoteSite("LON"
   );
   ```

5. **Add the Contents of the Configuration File**

As a default, Red Hat JBoss Data Grid includes JGroups configuration files such as **jgroups-tcp.xml** and **jgroups-udp.xml** in the **infinispan-core-*{VERSION}*.jar** package.

Copy the JGroups configuration to a new file (in this example, it is named **jgroups-with-relay.xml**) and add the provided configuration information to this file. Note that the **relay.RELAY2** protocol configuration must be the last protocol in the configuration stack.

```
<config>
    ...
    <relay.RELAY2 site="LON"
              config="relay.xml"
              relay_multicasts="false" />
</config>
```

6. **Configure the relay.xml File**
   Set up the **relay.RELAY2** configuration in the **relay.xml** file. This file describes the global cluster configuration.

```
<RelayConfiguration>
    <sites>
        <site name="LON"
              id="0">
            <bridges>
                <bridge config="jgroups-global.xml"
                        name="global"/>
                </bridges>
        </site>
        <site name="NYC"
              id="1">
            <bridges>
                <bridge config="jgroups-global.xml"
                        name="global"/>
                </bridges>
        </site>
        <site name="SFO"
              id="2">
            <bridges>
                <bridge config="jgroups-global.xml"
                        name="global"/>
            </bridges>
        </site>
    </sites>
</RelayConfiguration>
```

7. **Configure the Global Cluster**
   The file **jgroups-global.xml** referenced in **relay.xml** contains another JGroups configuration which is used for the global cluster: communication between sites.

   The global cluster configuration is usually **TCP**-based and uses the **TCPPING** protocol (instead of **PING** or **MPING**) to discover members. Copy the contents of **jgroups-tcp.xml** into **jgroups-global.xml** and add the following configuration in order to configure **TCPPING**:

```
<config>
    <TCP bind_port="7800" ... />
    <TCPPING
initial_hosts="lon.hostname[7800],nyc.hostname[7800],sfo.hostname[78
00]"
             num_initial_members="3"
             ergonomics="false" />
          <!-- Rest of the protocols -->
</config>
```

Replace the hostnames (or IP addresses) in ***`TCPPING.initial_hosts`*** with those used for your site masters. The ports (**`7800`** in this example) must match the ***`TCP.bind_port`***.

For more information about the **`TCPPING`** protocol, see Section 26.2.1.3, "Using the TCPPing Protocol"

Report a bug

## 29.3. TAKING A SITE OFFLINE

In Red Hat JBoss Data Grid's Cross-datacenter replication configuration, if backing up to one site fails a certain number of times during a time interval, that site can be marked as offline automatically. This feature removes the need for manual intervention by an administrator to mark the site as offline.

It is possible to configure JBoss Data Grid to take down a site automatically when specified conditions are met, or for an administrator to manually take down a site:

- Configure automatically taking a site offline:
  - Declaratively in Remote Client-Server mode.
  - Declaratively in Library mode.
  - Using the programmatic method.
- Manually taking a site offline:
  - Using JBoss Operations Network (JON).
  - Using the JBoss Data Grid Command Line Interface (CLI).

Report a bug

### 29.3.1. Taking a Site Offline (Remote Client-Server Mode)

In Red Hat JBoss Data Grid's Remote Client-Server mode, the **`take-offline`** element is added to the **`backup`** element to configure when a site is automatically taken offline.

**Example 29.2. Taking a Site Offline in Remote Client-Server Mode**

```
<backup>
 <take-offline after-failures="${NUMBER}"
        min-wait="${PERIOD}" />
</backup>
```

The **`take-offline`** element use the following parameters to configure when to take a site offline:

- The ***after-failures*** parameter specifies the number of times attempts to contact a site can fail before the site is taken offline.
- The ***min-wait*** parameter specifies the number (in milliseconds) to wait to mark an unresponsive site as offline. The site is offline when the ***min-wait*** period elapses after the first attempt, and the number of failed attempts specified in the ***after-failures*** parameter occur.

Report a bug

### 29.3.2. Taking a Site Offline (Library Mode)

In Red Hat JBoss Data Grid's Library mode, use the **backupFor** element after defining all back up sites within the **backups** element:

**Example 29.3. Taking a Site Offline in Library Mode**

```
<backup>
        <takeOffline afterFailures="${NUM}"
                     minTimeToWait="${PERIOD}"/>
</backup>
```

Add the **takeOffline** element to the **backup** element to configure automatically taking a site offline.

- The ***afterFailures*** parameter specifies the number of times attempts to contact a site can fail before the site is taken offline. The default value (`0`) allows an infinite number of failures if ***minTimeToWait*** is less than `0`. If the ***minTimeToWait*** is not less than `0`, ***afterFailures*** behaves as if the value is negative. A negative value for this parameter indicates that the site is taken offline after the time specified by ***minTimeToWait*** elapses.
- The ***minTimeToWait*** parameter specifies the number (in milliseconds) to wait to mark an unresponsive site as offline. The site is taken offline after the number attempts specified in the ***afterFailures*** parameter conclude and the time specified by ***minTimeToWait*** after the first failure has elapsed. If this parameter is set to a value smaller than or equal to `0`, this parameter is disregarded and the site is taken offline based solely on the ***afterFailures*** parameter.

Report a bug

### 29.3.3. Taking a Site Offline (Programmatically)

To configure taking a Cross-datacenter replication site offline automatically in Red Hat JBoss Data Grid programmatically:

**Example 29.4. Taking a Site Offline Programmatically**

```
lon.sites().addBackup()
      .site("NYC")
      .backupFailurePolicy(BackupFailurePolicy.FAIL)
      .strategy(BackupConfiguration.BackupStrategy.SYNC)
```

```
        .takeOffline()
           .afterFailures(500)
           .minTimeToWait(10000);
```

Report a bug

### 29.3.4. Taking a Site Offline via JBoss Operations Network (JON)

A site can be taken offline in Red Hat JBoss Data Grid using the JBoss Operations Network operations. For a list of the metrics, see Section 22.6.2, "JBoss Operations Network Plugin Operations"

Report a bug

### 29.3.5. Taking a Site Offline via the CLI

Use Red Hat JBoss Data Grid's Command Line Interface (CLI) to manually take a site from a cross-datacenter replication configuration down if it is unresponsive using the ***site*** command.

The **site** command can be used to check the status of a site as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> site --status
${SITENAME}
```

The result of this command would either be **online** or **offline** according to the current status of the named site.

The command can be used to bring a site online or offline by name as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> site --offline
${SITENAME}
```

```
[jmx://localhost:12000/MyCacheManager/namedCache]> site --online
${SITENAME}
```

If the command is successful, the output **ok** displays after the command.

For more information about the JBoss Data Grid CLI and its commands, see the *Developer Guide*'s chapter on the JBoss Data Grid Command Line Interface (CLI)

Report a bug

### 29.3.6. Bring a Site Back Online

After a site is taken offline, currently the only way to bring the site back online is using the JMX console to invoke the **bringSiteOnline(*siteName*)** operation on the **XSiteAdmin** MBean. For details about this MBean, see Section A.21, "XSiteAdmin"

Report a bug

## 29.4. CONFIGURE MULTIPLE SITE MASTERS

A standard Red Hat JBoss Data Grid cross-datacenter replication configuration includes one master node for each site. The master node is a gateway for other nodes to communicate with the master nodes at other sites.

This standard configuration works for a simple cross-datacenter replication configuration. However, with a larger volume of traffic between the sites, passing traffic through a single master node can create a bottleneck, which slows communication across nodes.

In JBoss Data Grid, configure multiple master nodes for each site to optimize traffic across multiple sites.

Report a bug

### 29.4.1. Multiple Site Master Operations

When multiple site masters are enabled and configured, the master nodes in each site joins the local cluster (i.e. the local site) as well as the global cluster (which includes nodes that are members of multiple sites).

Each node that acts as a site master and maintains a routing table that consists of a list of target sites and site masters. When a message arrives, a random master node for the destination site is selected. The message is then forwarded to the random master node, where it is sent to the destination node (unless the randomly selected node was the destination).

Report a bug

### 29.4.2. Configure Multiple Site Masters (Remote Client-Server Mode)

**Prerequisites**

Configure Cross-Datacenter Replication for Red Hat JBoss Data Grid's Remote Client-Server Mode.

**Procedure 29.4. Set Multiple Site Masters in Remote Client-Server Mode**

1. **Locate the Target Configuration**
   Locate the target site's configuration in the `clustered-xsite.xml` example configuration file. The sample configuration looks like the following example:

```
<relay site="LON">
 <remote-site name="NYC" stack="tcp" cluster="global"/>
 <remote-site name="SFO" stack="tcp" cluster="global"/>
 <property name="relay_multicasts">false</property>
</relay>
```

2. **Configure Maximum Sites**
   Use the ***`max_site_masters`*** property to determine the maximum number of master nodes within the site. Set this value to the number of nodes in the site to make every node a master.

```
<relay site="LON">
 <remote-site name="NYC" stack="tcp" cluster="global"/>
 <remote-site name="SFO" stack="tcp" cluster="global"/>
 <property name="relay_multicasts">false</property>
 <property name="max_site_masters">16</property>
</relay>
```

3. **Configure Site Master**
   Use the ***can_become_site_master*** property to allow the node to become the site master. This flag is set to **true** as a default. Setting this flag to **false** prevents the node from becoming a site master. This is required in situations where the node does not have a network interface connected to the external network.

```
<relay site="LON">
 <remote-site name="NYC" stack="tcp" cluster="global"/>
 <remote-site name="SFO" stack="tcp" cluster="global"/>
 <property name="relay_multicasts">false</property>
 <property name="max_site_masters">16</property>
 <property name="can_become_site_master">true</property>
</relay>
```

Report a bug

### 29.4.3. Configure Multiple Site Masters (Library Mode)

To configure multiple site masters in Red Hat JBoss Data Grid's Library Mode:

**Procedure 29.5. Configure Multiple Site Masters (Library Mode)**

1. **Configure Cross-Datacenter Replication**
   Configure Cross-Datacenter Replication in JBoss Data Grid. Use the instructions in Section 29.2.2.1, "Configure Cross-Datacenter Replication Declaratively" for an XML configuration or the instructions in Section 29.2.2.2, "Configure Cross-Datacenter Replication Programmatically" for a programmatic configuration.

2. **Add the Contents of the Configuration File**
   Add the ***can_become_site_master*** and ***max_site_masters*** parameters to the configuration as follows:

```
<config>
    ...
    <relay.RELAY2 site="LON"
              config="relay.xml"
              relay_multicasts="false"
              can_become_site_master="true"
              max_site_masters="16"/>
</config>
```

   Set the ***max_site_masters*** value to the number of nodes in the cluster to make all nodes masters.

Report a bug

# CHAPTER 30. ROLLING UPGRADES

In Red Hat JBoss Data Grid, rolling upgrades permit a cluster to be upgraded from one version to a new version without experiencing any downtime. This allows nodes to be upgraded without the need to restart the application or risk losing data.

In JBoss Data Grid, rolling upgrades can only be performed in Remote Client-Server mode.

Report a bug

## 30.1. ROLLING UPGRADES USING REST

The following procedure outlines using Red Hat JBoss Data Grid installations as a remote grid using the REST protocol. This procedure applies to rolling upgrades for the grid, not the client application.

**Procedure 30.1. Perform Rolling Upgrades Using REST**

In the instructions, the Source Cluster refers to the old cluster that is currently in use and the Target Cluster refers to the destination cluster for our data.

1. **Configure the Target Cluster**
   Use either different network settings or a different JGroups cluster name to set the Target Cluster (consisting of nodes with new JBoss Data Grid) apart from the Source Cluster. For each cache, configure a `RestCacheStore` with the following settings:

   a. Ensure that the host and port values point to the Source Cluster.

   b. Ensure that the path value points to the Source Cluster's REST endpoint.

2. **Start the Target Cluster**
   Start the Target Cluster's nodes. Configure each client to point to the Target Cluster instead of the Source Cluster. Eventually, the Target Cluster handles all requests instead of the Source Cluster. The Target Cluster then lazily loads data from the Source Cluster on demand using the `RestCacheStore`.

3. **Dump the Key Set**
   When all connections have shifted to the Target Cluster, remove the Source Cluster key set. This is done either using JMX or the CLI as follows:

   a. **Using JMX**
      Invoke the `recordKnownGlobalKeyset` operation on the `RollingUpgradeManager` MBean on the Source Cluster for all caches to be migrated.

   b. **Using the CLI**
      Run the `upgrade --dumpkeys` command on the Source Cluster for all caches to be migrated. Optionally, use the `--all` switch to dump all the caches in the cluster.

4. **Fetch the Remaining Data**
   The Target Cluster must fetch all the remaining data from the Source Cluster. This is done either using JMX or the CLI as follows:

   a. **Using JMX**
      Invoke the `synchronizeData` operation with the `rest` parameter specified on the `RollingUpgradeManager` MBean on the Target Cluster for all caches to be migrated.

b. **Using the CLI**
   Run the `upgrade --synchronize=rest` on the Target Cluster for all caches to be migrated. Optionally, use the *`--all`* switch to synchronize all caches in the cluster.

5. **Disable the RestCacheStore**
   Disable the `RestCacheStore` on the Target Cluster using either JMX or the CLI as follows:

   a. **Using JMX**
      Invoke the `disconnectSource` operation with the `rest` parameter specified on the `RollingUpgradeManager` MBean on the Target Cluster.

   b. **Using the CLI**
      Run the `upgrade --disconnectsource=rest` command on the Target Cluster. Optionally, use the *`--all`* switch to disconnect all caches in the cluster.

**Result**

Migration to the Target Cluster is complete. The Source Cluster can now be decommissioned.

Report a bug

## 30.2. ROLLING UPGRADES USING HOT ROD

The following process is used to perform rolling upgrades on Red Hat JBoss Data Grid running in Remote Client-Server mode, using Hot Rod. This procedure is designed to upgrade the data grid itself, and does not upgrade the client application.

**IMPORTANT**

Ensure that the correct version of the Hot Rod protocol is used with your JBoss Data Grid version:

- For JBoss Data Grid 6.1, use Hot Rod protocol version 1.2
- For JBoss Data Grid 6.2, use Hot Rod protocol version 1.3
- For JBoss Data Grid 6.3, use Hot Rod protocol version 2.0

**Prerequisite**

This procedure assumes that a cluster is already configured and running, and that it is using an older version of JBoss Data Grid. This cluster is referred to below as the Source Cluster and the Target Cluster refers to the new cluster to which data will be migrated.

1. **Configure the Target Cluster**
   Use either different network settings or a different JGroups cluster name to set the Target Cluster (consisting of nodes with new JBoss Data Grid) apart from the Source Cluster. For each cache, configure a `RemoteCacheStore` with the following settings:

   a. Ensure that *`remote-server`* points to the Source Cluster.

   b. Ensure that the cache name matches the name of the cache on the Source Cluster.

   c. Ensure that *`hotrod-wrapping`* is enabled (set to `true`).

   d. Ensure that *`purge`* is disabled (set to `false`).

e. Ensure that ***passivation*** is disabled (set to `false`).

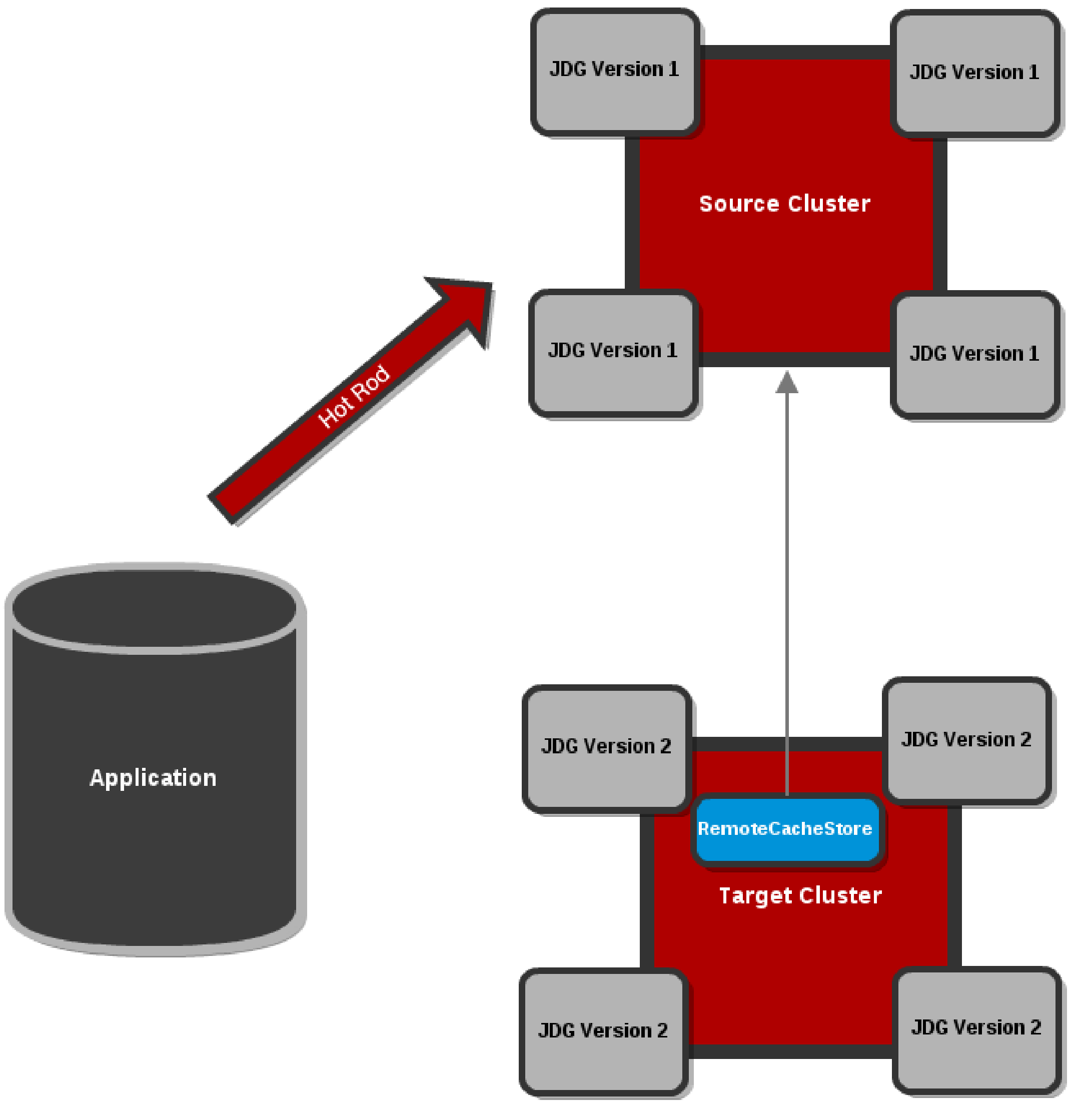


**Figure 30.1. Configure the Target Cluster with a RemoteCacheStore**

**NOTE**

See the **`$JDG_HOME/docs/examples/configs/standalone-hotrod-rolling-upgrade.xml`** file for a full example of the Target Cluster configuration for performing Rolling Upgrades.

2. **Start the Target Cluster**
   Start the Target Cluster's nodes. Configure each client to point to the Target Cluster instead of the Source Cluster. Eventually, the Target Cluster handles all requests instead of the Source Cluster. The Target Cluster then lazily loads data from the Source Cluster on demand using the **`RemoteCacheStore`**.

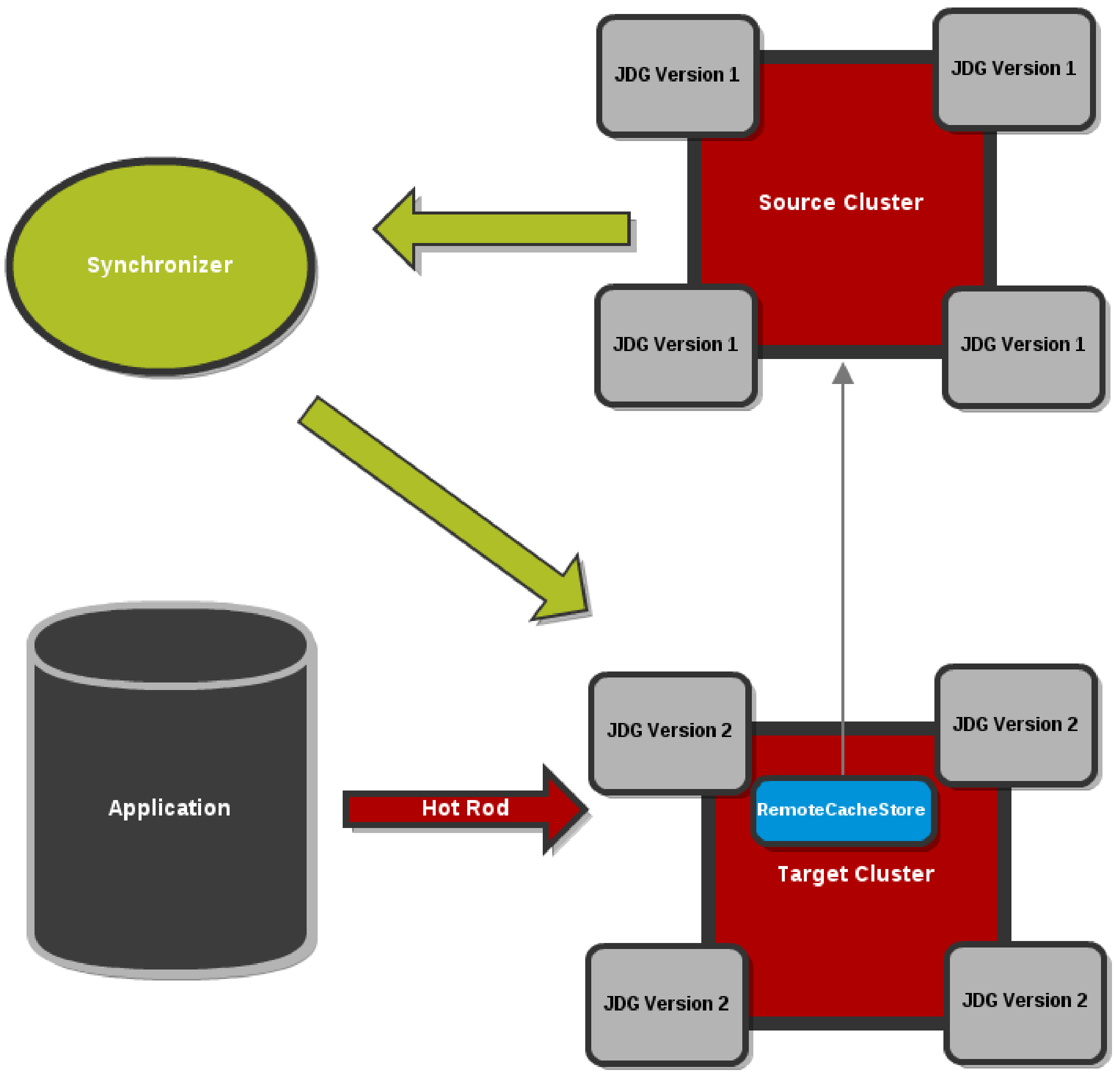


**Figure 30.2. Clients point to the Target Cluster with the Source Cluster as `RemoteCacheStore` for the Target Cluster.**

3. **Dump the Source Cluster keyset**
   When all connections are using the Target Cluster, the keyset on the Source Cluster must be dumped. This can be done using either JMX or the CLI:

   - **JMX**
     Invoke the ***`recordKnownGlobalKeyset`*** operation on the **`RollingUpgradeManager`** MBean on the Source Cluster for every cache that must be migrated.

   - **CLI**
     Invoke the **`upgrade --dumpkeys`** command on the Source Cluster for every cache that must be migrated, or use the **`--all`** switch to dump all caches in the cluster.

4. **Fetch remaining data from the Source Cluster**
   The Target Cluster fetches all remaining data from the Source Cluster. Again, this can be done using either JMX or CLI:

   - **JMX**
     Invoke the ***`synchronizeData`*** operation and specify the ***`hotrod`*** parameter on the **`RollingUpgradeManager`** MBean on the Target Cluster for every cache that must be migrated.

- **CLI**
  Invoke the `upgrade --synchronize=hotrod` command on the Target Cluster for every cache that must be migrated, or use the `--all` switch to synchronize all caches in the cluster.

5. **Disabling the `RemoteCacheStore`**
   Once the Target Cluster has obtained all data from the Source Cluster, the `RemoteCacheStore` on the Target Cluster must be disabled. This can be done as follows:

   - **JMX**
     Invoke the `disconnectSource` operation specifying the *`hotrod`* parameter on the `RollingUpgradeManager` MBean on the Target Cluster.

   - **CLI**
     Invoke the `upgrade --disconnectsource=hotrod` command on the Target Cluster.

6. **Decommission the Source Cluster**
   As a final step, decommission the Source Cluster.

Report a bug

## 30.3. ROLLINGUPGRADEMANAGER OPERATIONS

The `RollingUpgradeManager` Mbean handles the operations that allow data to be migrated from one version of Red Hat JBoss Data Grid to another when performing rolling upgrades. The `RollingUpgradeManager` operations are:

- *`recordKnownGlobalKeyset`* retrieves the entire keyset from the cluster running on the old version of JBoss Data Grid.

- *`synchronizeData`* performs the migration of data from the Source Cluster to the Target Cluster, which is running the new version of JBoss Data Grid.

- *`disconnectSource`* disables the Source Cluster, the older version of JBoss Data Grid, once data migration to the Target Cluster is complete.

Report a bug

## 30.4. REMOTECACHESTORE PARAMETERS FOR ROLLING UPGRADES

### 30.4.1. rawValues and RemoteCacheStore

By default, the RemoteCacheStore store's values are wrapped in InternalCacheEntry. Enabling the *`rawValues`* parameter causes the raw values to be stored instead for interoperability with direct access by RemoteCacheManagers.

*`rawValues`* must be enabled in order to interact with a Hot Rod cache via both RemoteCacheStore and RemoteCacheManager.

Report a bug

### 30.4.2. hotRodWrapping

The *`hotRodWrapping`* parameter is a shortcut that enables rawValues and sets an appropriate marshaller and entry wrapper for performing Rolling Upgrades.

Report a bug

# APPENDIX A. JMX MBEANS IN REDHAT JBOSS DATA GRID

## A.1. ACTIVATION

### `org.infinispan.eviction.ActivationManagerImpl`

Activates entries that have been passivated to the CacheStore by loading the entries into memory.

**Table A.1. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| activations | Number of activation events. | String | No |
| statisticsEnabled | Enables or disables the gathering of statistics by this component. | boolean | Yes |

**Table A.2. Operations**

| Name | Description | Signature |
|---|---|---|
| resetStatistics | Resets statistics gathered by this component. | void resetStatistics() |

Report a bug

## A.2. CACHE

### `org.infinispan.CacheImpl`

The Cache component represents an individual cache instance.

**Table A.3. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| cacheName | Returns the cache name. | String | No |
| cacheStatus | Returns the cache status. | String | No |
| configurationAsProperties | Returns the cache configuration in form of properties. | java.util.Properties | No |

| Name | Description | Type | Writable |
|---|---|---|---|
| version | Returns the version of Infinispan | java.lang.String | No |

Table A.4. Operations

| Name | Description | Signature |
|---|---|---|
| start | Starts the cache. | void start() |
| stop | Stops the cache. | void stop() |
| clear | Clears the cache. | void clear() |

Report a bug

## A.3. CACHELOADER

### `org.infinispan.interceptors.CacheLoaderInterceptor`

This component loads entries from a CacheStore into memory.

Table A.5. Attributes

| Name | Description | Type | Writable |
|---|---|---|---|
| cacheLoaderLoads | Number of entries loaded from the cache store. | long | No |
| cacheLoaderMisses | Number of entries that did not exist in cache store. | long | No |
| stores | Returns a collection of cache loader types which are configured and enabled. | Collection | No |
| statisticsEnabled | Enables or disables the gathering of statistics by this component. | boolean | Yes |

Table A.6. Operations

| Name | Description | Signature |
|---|---|---|
| disableStore | Disable all cache loaders of a given type, where type is a fully qualified class name of the cache loader to disable. | void disableStore(String storeType) |
| resetStatistics | Resets statistics gathered by this component. | void resetStatistics() |

Report a bug

## A.4. CACHEMANAGER

### `org.infinispan.manager.DefaultCacheManager`

The CacheManager component acts as a manager, factory, and container for caches in the system.

**Table A.7. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| cacheManagerStatus | The status of the cache manager instance. | String | No |
| clusterMembers | Lists members in the cluster. | String | No |
| clusterName | Cluster name. | String | No |
| clusterSize | Size of the cluster in the number of nodes. | int | No |
| createdCacheCount | The total number of created caches, including the default cache. | String | No |
| definedCacheCount | The total number of defined caches, excluding the default cache. | String | No |
| definedCacheNames | The defined cache names and their statuses. The default cache is not included in this representation. | String | No |

| Name | Description | Type | Writable |
|---|---|---|---|
| name | The name of this cache manager. | String | No |
| nodeAddress | The network address associated with this instance. | String | No |
| physicalAddresses | The physical network addresses associated with this instance. | String | No |
| runningCacheCount | The total number of running caches, including the default cache. | String | No |
| version | Infinispan version. | String | No. |

**Table A.8. Operations**

| Name | Description | Signature |
|---|---|---|
| startCache | Starts the default cache associated with this cache manager. | void startCache() |
| startCache | Starts a named cache from this cache manager. | void startCache (String p0) |

Report a bug

## A.5. CACHESTORE

### `org.infinispan.interceptors.CacheWriterInterceptor`

The CacheStore component stores entries to a CacheStore from memory.

**Table A.9. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| writesToTheStores | Number of writes to the store. | long | No |
| statisticsEnabled | Enables or disables the gathering of statistics by this component. | boolean | Yes |

**Table A.10. Operations**

| Name | Description | Signature |
|---|---|---|
| resetStatistics | Resets statistics gathered by this component. | void resetStatistics() |

Report a bug

## A.6. DEADLOCKDETECTINGLOCKMANAGER

`org.infinispan.util.concurrent.locks.DeadlockDetectingLockManager`

This component provides information about the number of deadlocks that were detected.

**Table A.11. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| detectedLocalDeadlocks | Number of local transactions that were rolled back due to deadlocks. | long | No |
| detectedRemoteDeadlocks | Number of remote transactions that were rolled back due to deadlocks. | long | No |
| overlapWithNotDeadlockAwareLockOwners | Number of situations when we try to determine a deadlock and the other lock owner is NOT a transaction. In this scenario we cannot run the deadlock detection mechanism. | long | No |
| totalNumberOfDetectedDeadlocks | Total number of local detected deadlocks. | long | No |
| concurrencyLevel | The concurrency level that the MVCC Lock Manager has been configured with. | int | No |
| numberOfLocksAvailable | The number of exclusive locks that are available. | int | No |

| Name | Description | Type | Writable |
|---|---|---|---|
| numberOfLocksHeld | The number of exclusive locks that are held. | int | No |

**Table A.12. Operations**

| Name | Description | Signature |
|---|---|---|
| resetStatistics | Resets statistics gathered by this component. | void resetStatistics() |

Report a bug

## A.7. DISTRIBUTIONMANAGER

### `org.infinispan.distribution.DistributionManagerImpl`

The DistributionManager component handles the distribution of content across a cluster.

**NOTE**

The DistrubutionManager component is only available in clustered mode.

**Table A.13. Operations**

| Name | Description | Signature |
|---|---|---|
| isAffectedByRehash | Determines whether a given key is affected by an ongoing rehash. | boolean isAffectedByRehash(Object p0) |
| isLocatedLocally | Indicates whether a given key is local to this instance of the cache. Only works with String keys. | boolean isLocatedLocally(String p0) |
| locateKey | Locates an object in a cluster. Only works with String keys. | List locateKey(String p0) |

Report a bug

## A.8. INTERPRETER

### `org.infinispan.cli.interpreter.Interpreter`

The Interpreter component executes command line interface (CLI operations).

**Table A.14. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| cacheNames | Retrieves a list of caches for the cache manager. | String[] | No |

**Table A.15. Operations**

| Name | Description | Signature |
|---|---|---|
| createSessionId | Creates a new interpreter session. | String createSessionId(String cacheName) |
| execute | Parses and executes IspnCliQL statements. | String execute(String p0, String p1) |

Report a bug

## A.9. INVALIDATION

### `org.infinispan.interceptors.InvalidationInterceptor`

The Invalidation component invalidates entries on remote caches when entries are written locally.

**Table A.16. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| invalidations | Number of invalidations. | long | No |
| statisticsEnabled | Enables or disables the gathering of statistics by this component. | boolean | Yes |

**Table A.17. Operations**

| Name | Description | Signature |
|---|---|---|
| resetStatistics | Resets statistics gathered by this component. | void resetStatistics() |

Report a bug

## A.10. LOCKMANAGER

### `org.infinispan.util.concurrent.locks.LockManagerImpl`

The LockManager component handles MVCC locks for entries.

**Table A.18. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| concurrencyLevel | The concurrency level that the MVCC Lock Manager has been configured with. | int | No |
| numberOfLocksAvailable | The number of exclusive locks that are available. | int | No |
| numberOfLocksHeld | The number of exclusive locks that are held. | int | No |

Report a bug

## A.11. LOCALTOPOLOGYMANAGER

**`org.infinispan.topology.LocalTopologyManagerImpl`**

The LocalTopologyManager component controls the cache membership and state transfer in Red Hat JBoss Data Grid.

**NOTE**

The LocalTopologyManager component is only available in clustered mode.

**Table A.19. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| rebalancingEnabled | If false, newly started nodes will not join the existing cluster nor will the state be transferred to them. If any of the current cluster members are stopped when rebalancing is disabled, the nodes will leave the cluster but the state will not be rebalanced among the remaining nodes. This will result in fewer copies than specified by the **`numOwners`** attribute until rebalancing is enabled again. | boolean | Yes |

Report a bug

## A.12. MASSINDEXER

### `org.infinispan.query.MassIndexer`

The MassIndexer component rebuilds the index using cached data.

**Table A.20. Operations**

| Name | Description | Signature |
|---|---|---|
| start | Starts rebuilding the index. | void start() |

**NOTE**

This operation is available only for caches with indexing enabled. For more information, see the Red Hat JBoss Data Grid *Infinispan Query Guide*

Report a bug

## A.13. PASSIVATION

### `org.infinispan.interceptors.PassivationInterceptor`

The Passivation component handles the passivation of entries to a CacheStore on eviction.

**Table A.21. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| passivations | Number of passivation events. | String | No |
| statisticsEnabled | Enables or disables the gathering of statistics by this component | boolean | Yes |

**Table A.22. Operations**

| Name | Description | Signature |
|---|---|---|
| resetStatistics | Resets statistics gathered by this component. | void resetStatistics() |

Report a bug

## A.14. RECOVERYADMIN

### `org.infinispan.transaction.xa.recovery.RecoveryAdminOperations`

The RecoveryAdmin component exposes tooling for handling transaction recovery.

**Table A.23. Operations**

| Name | Description | Signature |
|---|---|---|
| forceCommit | Forces the commit of an in-doubt transaction. | String forceCommit(long p0) |
| forceCommit | Forces the commit of an in-doubt transaction | String forceCommit(int p0, byte[] p1, byte[] p2) |
| forceRollback | Forces the rollback of an in-doubt transaction. | String forceRollback(long p0) |
| forceRollback | Forces the rollback of an in-doubt transaction | String forceRollback(int p0, byte[] p1, byte[] p2) |
| forget | Removes recovery info for the given transaction. | String forget(long p0) |
| forget | Removes recovery info for the given transaction. | String forget(int p0, byte[] p1, byte[] p2) |
| showInDoubtTransactions | Shows all the prepared transactions for which the originating node crashed. | String showInDoubtTransactions() |

Report a bug

## A.15. ROLLINGUPGRADEMANAGER

### `org.infinispan.upgrade.RollingUpgradeManager`

The RollingUpgradeManager component handles the control hooks in order to migrate data from one version of Red Hat JBoss Data Grid to another.

**Table A.24. Operations**

| Name | Description | Signature |
|---|---|---|
| disconnectSource | Disconnects the target cluster from the source cluster according to the specified migrator. | void disconnectSource(String p0) |
| recordKnownGlobalKeyset | Dumps the global known keyset to a well-known key for retrieval by the upgrade process. | void recordKnownGlobalKeyset() |

| Name | Description | Signature |
|---|---|---|
| synchronizeData | Synchronizes data from the old cluster to this using the specified migrator. | long synchronizeData(String p0) |

Report a bug

## A.16. RPCMANAGER

**`org.infinispan.remoting.rpc.RpcManagerImpl`**

The RpcManager component manages all remote calls to remote cache instances in the cluster.

**NOTE**

The RpcManager component is only available in clustered mode.

**Table A.25. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| averageReplicationTime | The average time spent in the transport layer, in milliseconds. | long | No |
| committedViewAsString | Retrieves the committed view. | String | No |
| pendingViewAsString | Retrieves the pending view. | String | No |
| replicationCount | Number of successful replications. | long | No |
| replicationFailures | Number of failed replications. | long | No |
| successRatio | Successful replications as a ratio of total replications. | String | No |
| successRatioFloatingPoint | Successful replications as a ratio of total replications in numeric double format. | double | No |

| Name | Description | Type | Writable |
|---|---|---|---|
| statisticsEnabled | Enables or disables the gathering of statistics by this component. | boolean | Yes |

**Table A.26. Operations**

| Name | Description | Signature |
|---|---|---|
| resetStatistics | Resets statistics gathered by this component. | void resetStatistics() |
| setStatisticsEnabled | Whether statistics should be enabled or disabled (true/false) | void setStatisticsEnabled(boolean enabled) |

Report a bug

## A.17. STATETRANSFERMANAGER

### `org.infinispan.statetransfer.StateTransferManager`

The StateTransferManager component handles state transfer in Red Hat JBoss Data Grid.

**NOTE**

The StateTransferManager component is only available in clustered mode.

**Table A.27. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| joinComplete | If true, the node has successfully joined the grid and is considered to hold state. If false, the join process is still in progress.. | boolean | No |
| stateTransferInProgres s | Checks whether there is a pending inbound state transfer on this cluster member. | boolean | No |

Report a bug

## A.18. STATISTICS

### `org.infinispan.interceptors.CacheMgmtInterceptor`

This component handles general statistics such as timings, hit/miss ratio, etc.

**Table A.28. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| averageReadTime | Average number of milliseconds for a read operation on the cache. | long | No |
| averageWriteTime | Average number of milliseconds for a write operation in the cache. | long | No |
| elapsedTime | Number of seconds since cache started. | long | No |
| evictions | Number of cache eviction operations. | long | No |
| hitRatio | Percentage hit/(hit+miss) ratio for the cache. | double | No |
| hits | Number of cache attribute hits. | long | No |
| misses | Number of cache attribute misses. | long | No |
| numberOfEntries | Number of entries currently in the cache. | int | No |
| readWriteRatio | Read/writes ratio for the cache. | double | No |
| removeHits | Number of cache removal hits. | long | No |
| removeMisses | Number of cache removals where keys were not found. | long | No |
| stores | Number of cache attribute PUT operations. | long | No |

| Name | Description | Type | Writable |
|---|---|---|---|
| timeSinceReset | Number of seconds since the cache statistics were last reset. | long | No |

**Table A.29. Operations**

| Name | Description | Signature |
|---|---|---|
| resetStatistics | Resets statistics gathered by this component. | void resetStatistics() |

Report a bug

## A.19. TRANSACTIONS

### org.infinispan.interceptors.TxInterceptor

The Transactions component manages the cache's participation in JTA transactions.

**Table A.30. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| commits | Number of transaction commits performed since last reset. | long | No |
| prepares | Number of transaction prepares performed since last reset. | long | No |
| rollbacks | Number of transaction rollbacks performed since last reset. | long | No |
| statisticsEnabled | Enables or disables the gathering of statistics by this component. | boolean | Yes |

**Table A.31. Operations**

| Name | Description | Signature |
|---|---|---|
| resetStatistics | Resets statistics gathered by this component. | void resetStatistics() |

Report a bug

## A.20. TRANSPORT

### `org.infinispan.server.core.transport.NettyTransport`

The Transport component manages read and write operations to and from the server.

**Table A.32. Attributes**

| Name | Description | Type | Writable |
|---|---|---|---|
| hostName | Returns the host to which the transport binds. | String | No |
| idleTimeout | Returns the idle timeout. | String | No |
| numberOfGlobalConnections | Returns a count of active connections in the cluster. This operation will make remote calls to aggregate results, so latency may have an impact on the speed of calculation for this attribute. | Integer | false |
| numberOfLocalConnections | Returns a count of active connections this server. | Integer | No |
| numberWorkerThreads | Returns the number of worker threads. | String | No |
| port | Returns the port to which the transport binds. | String | |
| receiveBufferSize | Returns the receive buffer size. | String | No |
| sendBufferSize | Returns the send buffer size. | String | No |
| totalBytesRead | Returns the total number of bytes read by the server from clients, including both protocol and user information. | String | No |

| Name | Description | Type | Writable |
|---|---|---|---|
| totalBytesWritten | Returns the total number of bytes written by the server back to clients, including both protocol and user information. | String | No |
| tcpNoDelay | Returns whether TCP no delay was configured or not. | String | No |

Report a bug

## A.21. XSITEADMIN

**`org.infinispan.xsite.XSiteAdminOperations`**

The XSiteAdmin component exposes tooling for backing up data to remote sites.

**Table A.33. Operations**

| Name | Description | Signature |
|---|---|---|
| bringSiteOnline | Brings the given site back online on all the cluster. | String bringSiteOnline(String p0) |
| amendTakeOffline | Amends the values for 'TakeOffline' functionality on all the nodes in the cluster. | String amendTakeOffline(String p0, int p1, long p2) |
| getTakeOfflineAfterFailures | Returns the value of the 'afterFailures' for the 'TakeOffline' functionality. | String getTakeOfflineAfterFailures(String p0) |
| getTakeOfflineMinTimeToWait | Returns the value of the 'minTimeToWait' for the 'TakeOffline' functionality. | String getTakeOfflineMinTimeToWait(String p0) |
| setTakeOfflineAfterFailures | Amends the values for 'afterFailures' for the 'TakeOffline' functionality on all the nodes in the cluster. | String setTakeOfflineAfterFailures(String p0, int p1) |
| setTakeOfflineMinTimeToWait | Amends the values for 'minTimeToWait' for the 'TakeOffline' functionality on all the nodes in the cluster. | String setTakeOfflineMinTimeToWait(String p0, long p1) |

| Name | Description | Signature |
|---|---|---|
| siteStatus | Check whether the given backup site is offline or not. | String siteStatus(String p0) |
| status | Returns the the status(offline/online) of all the configured backup sites. | String status() |
| takeSiteOffline | Takes this site offline in all nodes in the cluster. | String takeSiteOffline(String p0) |

Report a bug

# APPENDIX B. CONFIGURATION RECOMMENDATIONS

## B.1. TIMEOUT VALUES

**Table B.1. Timeout Value Recommendations for JBoss Data Grid**

| Timeout Value | Parent Element | Default Value | Recommended Value |
|---|---|---|---|
| distributedSyncTimeout | transport | 240,000 (4 minutes) | Same as default |
| lockAcquisitionTimeout | locking | 10,000 (10 seconds) | Same as default |
| cacheStopTimeout | transaction | 30,000 (30 seconds) | Same as default |
| completedTxTimeout | transaction | 60,000 (60 seconds) | Same as default |
| replTimeout | sync | 15,000 (15 seconds) | Same as default |
| timeout | stateTransfer | 240,000 (4 minutes) | Same as default |
| timeout | backup | 10,000 (10 seconds) | Same as default |
| flushLockTimeout | async | 1 (1 millisecond) | Same as default. Note that this value applies to asynchronous cache stores, but not asynchronous caches. |
| shutdownTimeout | async | 25,000 (25 seconds) | Same as default. Note that this value applies to asynchronous cache stores, but not asynchronous caches. |
| pushStateTimeout | singletonStore | 10,000 (10 seconds) | Same as default. |
| backup | replicationTimeout | 10,000 (10 seconds) | |
| remoteCallTimeout | clusterLoader | 0 | For most requirements, same as default. This value is usually set to the same as the ***`sync.replTimeout`*** value. |

Report a bug

# APPENDIX C. REFERENCES

## C.1. 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

## C.2. ABOUT CONSISTENCY GUARANTEE

Despite the locking of a single owner instead of all owners, Red Hat JBoss Data Grid's consistency guarantee remains intact. Consider the following situation:

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

2. If another cache access occurs on node **B**, or any other node, and **TX2** attempts to lock **K**, this access attempt 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

## C.3. ABOUT JBOSS CACHE

Red Hat 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 Red Hat JBoss Data Grid.

Report a bug

## C.4. ABOUT RELAY2

The **RELAY** protocol bridges two remote clusters by creating a connection between one node in each site. This allows multicast messages sent out in one site to be relayed to the other and vice versa.

JGroups includes the **RELAY2** protocol, which is used for communication between sites in Red Hat JBoss Data Grid's Cross-Site Replication.

The **RELAY2** protocol works similarly to **RELAY** but with slight differences. Unlike **RELAY**, the **RELAY2** protocol:

- connects more than two sites.
- connects sites that operate autonomously and are unaware of each other.

- offers both unicasts and multicast routing between sites.

Report a bug

## C.5. ABOUT RETURN VALUES

Values returned by cache operations are referred to as return values. In Red Hat 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

## C.6. 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

## C.7. 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

## C.8. 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. It consists of entry data attributes from the related entry.
- A value is the data assigned to and identified by the key.

Report a bug

## C.9. THE EXTERNALIZER

### C.9.1. 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 Red Hat 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

### C.9.2. Internal Externalizer Implementation Access

Externalizable objects should not access Red Hat JBoss Data Grids Externalizer implementations. The following is an example of incorrect usage:

```
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 correct usage:

```
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

## C.10. HASH SPACE ALLOCATION

### C.10.1. About Hash Space Allocation

Red Hat 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

### C.10.2. Locating a Key in the Hash Space

Red Hat JBoss Data Grid always uses an algorithm to locate a key in the hash space. As a result, the node that stores 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

### C.10.3. Requesting a Full Byte Array

**How can I request the Red Hat 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 C.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 D. REVISION HISTORY

**Revision 6.3.0-27** **Tue Feb 24 2015** **Rakesh Ghatvisave**

BZ-1180947: Updated chunksize definitions.

**Revision 6.3.0-26** **Mon Jan 12 2015** **Misha Husnain Ali**

BZ-1180266: Updated default information for eviction strategy.

**Revision 6.3.0-25** **Tue Jan 06 2015** **Misha Husnain Ali**

BZ-1173689: Removed appendix C due to non-relevance.

**Revision 6.3.0-24** **Fri Dec 05 2014** **Misha Husnain Ali**

Updated for 6.3.2

**Revision 6.3.0-23** **Mon Nov 17 2014** **Misha Husnain Ali**

BZ-1162244: Added a chapter that is relevant to administrators.

**Revision 6.3.0-22** **Fri Oct 31 2014** **Misha Husnain Ali**

BZ-1158083: Removed the singleton element.

**Revision 6.3.0-21** **Tue Oct 28 2014** **Misha Husnain Ali**

BZ-1158083: Removed an irrelevant configuration element.

**Revision 6.3.0-21** **Tue Oct 28 2014** **Misha Husnain Ali**

BZ-1157886: Removed incorrect configuration description from async element details.

**Revision 6.3.0-20** **Fri Sep 26 2014** **Misha Husnain Ali**

BZ-1145759: Updated programmatic configuration.

**Revision 6.3.0-19** **Mon Sep 15 2014** **Misha Husnain Ali**

BZ-1122567: Added and updated timeout recommended values.

BZ-1112040: Updated and expanded state transfer topic.

**Revision 6.3.0-18** **Wed Sep 03 2014** **Rakesh Ghatvisave**

BZ-1122298: Added multiple XML schemas in JDBC cache store configurations.

BZ-1138877: Updated parameter information.

**Revision 6.3.0-16** **Wed Aug 27 2014** **Misha Husnain Ali**

BZ-1118181: Consolidated the cache store configuration information.

BZ-1134591: Updated JGroups and EAP versions.

**Revision 6.3.0-15** **Thu Aug 07 2014** **Gemma Sheldon**

BZ-118700: Updated Consistent Hash information.

BZ-921589: Added steps for creating new custom cache store.

**Revision 6.3.0-14** **Fri Aug 01 2014** **Rakesh Ghatvisave**

BZ-1122948: Added metrics and operations in the topic.

BZ-1122877: Restructuring of JON sections.

BZ-1122561: Added and updated content about supported Hot Rod operations.

**Revision 6.3.0-13** **Fri Jul 25 2014** **Rakesh Ghatvisave**

BZ-1122289: Added JMX information to JON topic.

BZ-1122284: Added transport element in cross site transfer example.

BZ-1118181: Removed duplicate config information for cache store chapter.

**Revision 6.3.0-12** **Tue Jul 15 2014** **Rakesh Ghatvisave**

BZ-1119624: Incorporated QE review changes.