Red Hat JBoss Data Grid 6.2

Developer Guide

For use with Red Hat JBoss Data Grid 6.2.1

Edition 3

Last Updated: 2017-11-21

Red Hat JBoss Data Grid 6.2 Developer Guide

For use with Red Hat JBoss Data Grid 6.2.1

Edition 3

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

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

Mandar Joshi
Red Hat Engineering Content Services
majoshi@redhat.com

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

Legal Notice

Copyright © 2014 Red Hat, Inc.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

An advanced guide intended for developers using Red Hat JBoss Data Grid 6.2.1

Table of Contents

PART I. PROGRAMMABLE APIS 7

CHAPTER 1. THE CACHE API 8
1.1. USING THE CONFIGURATIONBUILDER API TO CONFIGURE THE CACHE API 8
1.2. PER-INVOCATION FLAGS 9
1.2.1. Per-Invocation Flag Functions 9
1.2.2. Configure Per-Invocation Flags 10
1.2.3. Per-Invocation Flags Example 10
1.3. THE ADVANCEDCACHE INTERFACE 10
1.3.1. Flag Usage with the AdvancedCache Interface 11
1.3.2. Custom Interceptors and the AdvancedCache Interface 11
1.3.3. Custom Interceptors 11
1.3.3.1. Custom Interceptor Design 11
1.3.3.2. Adding Custom Interceptors Declaratively 11
1.3.3.3. Adding Custom Interceptors Programmatically 13

CHAPTER 2. THE BATCHING API 15
2.1. ABOUT JAVA TRANSACTION API TRANSACTIONS 15
2.2. BATCHING AND THE JAVA TRANSACTION API (JTA) 15
2.3. USING THE BATCHING API 16
2.3.1. Enable the Batching API 16
2.3.2. Configure the Batching API 16
2.3.3. Use the Batching API 16
2.3.4. Batching API Usage Example 17

CHAPTER 3. THE GROUPING API 18
3.1. GROUPING API OPERATIONS 18
3.2. GROUPING API USE CASE 18
3.3. CONFIGURE THE GROUPING API 19
3.3.1. Enable Groups 19
3.3.2. Specify an Intrinsic Group 19
3.3.3. Specify an Extrinsic Group 20
3.3.4. Register Groupers 21

CHAPTER 4. THE PERSISTENCE SPI 22
4.1. PERSISTENCE SPI BENEFITS 22
4.2. PROGRAMMATICALLY CONFIGURE THE PERSISTENCE SPI 22

CHAPTER 5. THE CONFIGURATIONBUILDER API 24
5.1. USING THE CONFIGURATIONBUILDER API 24
5.1.1. Programmatically Create a CacheManager and Replicated Cache 24
5.1.2. Create a Customized Cache Using the Default Named Cache 25
5.1.3. Create a Customized Cache Using a Non-Default Named Cache 26
5.1.4. Using the Configuration Builder to Create Caches Programmatically 27
5.1.5. Global Configuration Examples 27
5.1.5.1. Globally Configure the Transport Layer 27
5.1.5.2. Globally Configure the Cache Manager Name 28
5.1.5.3. Globally Customize Thread Pool Executors 28
5.1.6. Cache Level Configuration Examples 28
5.1.6.1. Cache Level Configuration for the Cluster Mode 28
5.1.6.2. Cache Level Eviction and Expiration Configuration 28
5.1.6.3. Cache Level Configuration for JTA Transactions 29
5.1.6.4. Cache Level Configuration Using Chained Persistent Stores 29

5.1.6.5. Cache Level Configuration for Advanced Externalizers 29

CHAPTER 6. THE EXTERNALIZABLE API 30
6.1. CUSTOMIZE EXTERNALIZERS 30
6.2. ANNOTATING OBJECTS FOR MARSHALLING USING @SERIALIZEWITH 30
6.3. USING AN ADVANCED EXTERNALIZER 31
6.3.1. Implement the Methods 32
6.3.2. Link Externalizers with Marshaller Classes 32
6.3.3. Register the Advanced Externalizer (Declaratively) 33
6.3.4. Register the Advanced Externalizer (Programmatically) 34
6.3.5. Register Multiple Externalizers 34
6.4. CUSTOM EXTERNALIZER ID VALUES 35
6.4.1. Customize the Externalizer ID (Declaratively) 35
6.4.2. Customize the Externalizer ID (Programmatically) 36

CHAPTER 7. THE NOTIFICATION/LISTENER API 37
7.1. LISTENER EXAMPLE 37
7.2. CACHE ENTRY MODIFIED LISTENER CONFIGURATION 37
7.3. LISTENER NOTIFICATIONS 37
7.3.1. About Cache-level Notifications 37
7.3.2. Cache Manager-level Notifications 38
7.3.3. About Synchronous and Asynchronous Notifications 38
7.4. NOTIFYINGFUTURES 38
7.4.1. NotifyingFutures Example 38

PART II. REMOTE CLIENT-SERVER MODE INTERFACES 40

CHAPTER 8. THE ASYNCHRONOUS API 41
8.1. ASYNCHRONOUS API BENEFITS 41
8.2. ABOUT ASYNCHRONOUS PROCESSES 41
8.3. RETURN VALUES AND THE ASYNCHRONOUS API 42

CHAPTER 9. THE REST INTERFACE 43
9.1. RUBY CLIENT CODE 43
9.2. USING JSON WITH RUBY EXAMPLE 43
9.3. PYTHON CLIENT CODE 44
9.4. JAVA CLIENT CODE 44
9.5. CONFIGURE THE INTERFACE USING CONNECTORS 46
9.5.1. Configure REST Connectors 47
9.6. USING THE REST INTERFACE 48
9.6.1. Adding Data Using REST 48
9.6.1.1. About PUT /{cacheName}/{cacheKey} 49
9.6.1.2. About POST /{cacheName}/{cacheKey} 49
9.6.2. Retrieving Data Using REST 49
9.6.2.1. About GET /{cacheName}/{cacheKey} 50
9.6.2.2. About HEAD /{cacheName}/{cacheKey} 50
9.6.3. Removing Data Using REST 50
9.6.3.1. About DELETE /{cacheName}/{cacheKey} 50
9.6.3.2. About DELETE /{cacheName} 50
9.6.3.3. Background Delete Operations 50
9.6.4. REST Interface Operation Headers 51
9.7. REST INTERFACE SECURITY 53
9.7.1. Publish REST Endpoints as a Public Interface 54
9.7.2. Enable Security for the REST Endpoint 54

CHAPTER 10. THE MEMCACHED INTERFACE **56**
10.1. ABOUT MEMCACHED SERVERS 56
10.2. MEMCACHED STATISTICS 56
10.3. CONFIGURE THE INTERFACE USING CONNECTORS 58
10.3.1. Configure Memcached Connectors 58
10.4. MEMCACHED INTERFACE SECURITY 60
10.4.1. Publish Memcached Endpoints as a Public Interface 60

CHAPTER 11. THE HOT ROD INTERFACE **61**
11.1. ABOUT HOT ROD 61
11.2. THE BENEFITS OF USING HOT ROD OVER MEMCACHED 61
11.3. HOT ROD HASH FUNCTIONS 62
11.4. HOT ROD SERVER NODES 62
11.4.1. About Consistent Hashing Algorithms 62
11.5. HOT ROD HEADERS 63
11.5.1. Hot Rod Header Data Types 63
11.5.2. Request Header 63
11.5.3. Response Header 65
11.5.4. Topology Change Headers 65
11.5.4.1. Topology Change Marker Values 66
11.5.4.2. Topology Change Headers for Topology-Aware Clients 66
11.5.4.3. Topology Change Headers for Hash Distribution-Aware Clients 67
11.6. HOT ROD OPERATIONS 68
11.6.1. Hot Rod Operations 68
11.6.2. Hot Rod BulkGetKeys Operation 69
11.6.3. Hot Rod BulkGet Operation 71
11.6.4. Hot Rod Clear Operation 72
11.6.5. Hot Rod ContainsKey Operation 72
11.6.6. Hot Rod Get Operation 73
11.6.7. Hot Rod GetWithMetadata Operation 74
11.6.8. Hot Rod Ping Operation 76
11.6.9. Hot Rod PutIfAbsent Operation 76
11.6.10. Hot Rod Put Operation 78
11.6.11. Hot Rod Query Operation 78
11.6.12. Hot Rod RemoveIfUnmodified Operation 79
11.6.13. Hot Rod Remove Operation 80
11.6.14. Hot Rod ReplaceIfUnmodified Operation 81
11.6.15. Hot Rod Replace Operation 82
11.6.16. Hot Rod Stats Operation 83
11.7. HOT ROD OPERATION VALUES 85
11.7.1. Magic Values 85
11.7.2. Status Values 86
11.7.3. Transaction Type Values 86
11.7.4. Client Intelligence Values 87
11.7.5. Flag Values 87
11.7.6. Hot Rod Error Handling 87
11.8. PUT REQUEST EXAMPLE 88
11.9. HOT ROD JAVA CLIENT 90
11.9.1. Hot Rod Java Client Download 90
11.9.2. Hot Rod Java Client Configuration 90
11.9.3. Hot Rod Java Client Basic API 92
11.9.4. Hot Rod Java Client Versioned API 92
11.10. HOT ROD C ++ CLIENT 93

11.10.1. Hot Rod C ++ Client Formats 93
11.10.2. Hot Rod C ++ Client Download 94
11.10.3. Hot Rod C ++ Client Configuration 94
11.10.4. Hot Rod C ++ Client API 95
11.10.5. Hot Rod C ++ Client Requisites 95
11.11. INTEROPERABILITY BETWEEN C++ AND HOT ROD JAVA CLIENT 95
11.12. CONFIGURE THE INTERFACE USING CONNECTORS 96
11.12.1. Configure Hot Rod Connectors 96
11.13. HOT ROD INTERFACE SECURITY 100
11.13.1. Publish Hot Rod Endpoints as a Public Interface 100
11.13.2. SSL/TLS Authentication for Hot Rod 100

PART III. ADVANCED FEATURES IN RED HAT JBOSS DATA GRID **102**

CHAPTER 12. TRANSACTIONS **103**
12.1. ABOUT JAVA TRANSACTION API TRANSACTIONS 103
12.2. TRANSACTIONS SPANNING MULTIPLE CACHE INSTANCES 103
12.3. THE TRANSACTION MANAGER 103

CHAPTER 13. MARSHALLING **105**
13.1. ABOUT MARSHALLING FRAMEWORK 105
13.2. SUPPORT FOR NON-SERIALIZABLE OBJECTS 105
13.3. HOT ROD AND MARSHALLING 105
13.4. CONFIGURING THE MARSHALLER USING THE REMOTECACHEMANAGER 106
13.5. TROUBLESHOOTING 107
13.5.1. Marshalling Troubleshooting 107
13.5.2. Other Marshalling Related Issues 110

CHAPTER 14. LISTENERS AND NOTIFICATIONS **113**
14.1. THE NOTIFICATION/LISTENER API 113
14.2. LISTENER NOTIFICATIONS 113
14.2.1. About Cache-level Notifications 113
14.2.2. Cache Manager-level Notifications 113
14.2.3. About Synchronous and Asynchronous Notifications 113
14.3. MODIFYING CACHE ENTRIES 114
14.3.1. Cache Entry Modified Listener Configuration 114
14.3.2. Listener Example 114
14.3.3. Cache Entry Modified Listener Example 114
14.4. NOTIFYINGFUTURES 115
14.4.1. NotifyingFutures Example 115

CHAPTER 15. THE INFINSPAN CDI MODULE **116**
15.1. USING INFINISPAN CDI 116
15.1.1. Infinispan CDI Prerequisites 116
15.1.2. Set the CDI Maven Dependency 116
15.2. USING THE INFINISPAN CDI MODULE 116
15.2.1. Configure and Inject Infinispan Caches 117
15.2.1.1. Inject an Infinispan Cache 117
15.2.1.2. Inject a Remote Infinispan Cache 117
15.2.1.3. Set the Injection's Target Cache 117
15.2.1.3.1. Create a Qualifier Annotation 117
15.2.1.3.2. Add a Producer Class 118
15.2.1.3.3. Inject the Desired Class 118
15.2.2. Configure Cache Managers with CDI 118

15.2.2.1. Specify the Default Configuration 118
15.2.2.2. Override the Creation of the Embedded Cache Manager 119
15.2.2.3. Configure a Remote Cache Manager 120
15.2.2.4. Configure Multiple Cache Managers with a Single Class 120
15.2.3. Storage and Retrieval Using CDI Annotations 122
15.2.3.1. Configure Cache Annotations 122
15.2.3.2. Enable Cache Annotations 122
15.2.3.3. Caching the Result of a Method Invocation 122
15.2.3.3.1. Specify the Cache Used 123
15.2.3.3.2. Cache Keys for Cached Results 124
15.2.3.3.3. Generate a Custom Key 124
15.2.3.3.4. CacheKey Implementation Code 125
15.2.4. Cache Operations 125
15.2.4.1. Update a Cache Entry 125
15.2.4.2. Remove an Entry from the Cache 126
15.2.4.3. Clear the Cache 126

CHAPTER 16. ROLLING UPGRADES ... 127
16.1. ROLLING UPGRADES USING REST 127
16.2. ROLLING UPGRADES USING HOT ROD (REMOTE CLIENT-SERVER MODE) 128
16.3. ROLLINGUPGRADEMANAGER OPERATIONS 132
16.4. REMOTECACHESTORE PARAMETERS FOR ROLLING UPGRADES 132
16.4.1. rawValues and RemoteCacheStore 132
16.4.2. hotRodWrapping 132

CHAPTER 17. MAPREDUCE ... 134
17.1. THE MAPREDUCE API 134
17.1.1. MapReduceTask 136
17.1.2. Mapper and CDI 136
17.2. MAPREDUCETASK DISTRIBUTED EXECUTION 137
17.3. MAP REDUCE EXAMPLE 138

CHAPTER 18. DISTRIBUTED EXECUTION ... 141
18.1. DISTRIBUTEDCALLABLE API 141
18.2. CALLABLE AND CDI 142
18.3. DISTRIBUTED TASK FAILOVER 142
18.4. DISTRIBUTED TASK EXECUTION POLICY 143
18.5. DISTRIBUTED EXECUTION EXAMPLE 144

CHAPTER 19. DATA INTEROPERABILITY ... 147
19.1. INTEROPERABILITY BETWEEN LIBRARY AND REMOTE CLIENT-SERVER ENDPOINTS 147
19.2. USING COMPATIBILITY MODE 147
19.3. PROTOCOL INTEROPERABILITY 147
19.3.1. Use Cases and Requirements 148
19.3.2. Protocol Interoperability Over REST 149

APPENDIX A. REVISION HISTORY ... 150

PART I. PROGRAMMABLE APIS

Red Hat JBoss Data Grid provides the following programmable APIs:

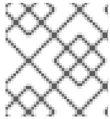
- Cache
- Batching
- Grouping
- Persistence (formerly CacheStore)
- ConfigurationBuilder
- Externalizable
- Notification (also known as the Listener API because it deals with Notifications and Listeners)

Report a bug

CHAPTER 1. THE CACHE API

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

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



NOTE

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

Report a bug

1.1. USING THE CONFIGURATIONBUILDER API TO CONFIGURE THE CACHE API

Red Hat JBoss Data Grid uses a ConfigurationBuilder API to configure caches.

Caches are configured programmatically using the ***ConfigurationBuilder*** helper object.

The following is an example of a synchronously replicated cache configured programmatically using the ConfigurationBuilder API:

```
Configuration c = new
ConfigurationBuilder().clustering().cacheMode(CacheMode.REPL_SYNC).build()
;

String newCacheName = "repl";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

Configuration Explanation:

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

1. ```
   Configuration c = new
   ConfigurationBuilder().clustering().cacheMode(CacheMode.REPL_SYNC).b
   uild();
   ```

   In the first line of the configuration, a new cache configuration object (named **c**) is created using the **ConfigurationBuilder**. Configuration **c** is assigned the default values for all cache configuration options except the cache mode, which is overridden and set to synchronous replication (**REPL_SYNC**).

2. ```
   String newCacheName = "repl";
   ```

   In the second line of the configuration, a new variable (of type **String**) is created and assigned the value **repl**.

3.
```
manager.defineConfiguration(newCacheName, c);
```

In the third line of the configuration, the cache manager is used to define a named cache configuration for itself. This named cache configuration is called **repl** and its configuration is based on the configuration provided for cache configuration **c** in the first line.

4.
```
Cache<String, String> cache = manager.getCache(newCacheName);
```

In the fourth line of the configuration, the cache manager is used to obtain a reference to the unique instance of the **repl** that is held by the cache manager. This cache instance is now ready to be used to perform operations to store and retrieve data.

NOTE

JBoss EAP includes its own underlying JMX. This can cause a collision when using the sample code with JBoss EAP and display an error such as **org.infinispan.jmx.JmxDomainConflictException: Domain already registered org.infinispan**.

To avoid this, configure global configuration as follows:

```
GlobalConfiguration glob = new GlobalConfigurationBuilder()
 .clusteredDefault()
        .globalJmxStatistics()
          .allowDuplicateDomains(true)
          .enable()
        .build();
```

Report a bug

1.2. PER-INVOCATION FLAGS

Per-invocation flags can be used with data grids in Red Hat JBoss Data Grid to specify behavior for each cache call. Per-invocation flags facilitate the implementation of potentially time saving optimizations.

Report a bug

1.2.1. Per-Invocation Flag Functions

The **putForExternalRead()** method in Red Hat JBoss Data Grid's Cache API uses flags internally. This method can load a JBoss Data Grid cache with data loaded from an external resource. To improve the efficiency of this call, JBoss Data Grid calls a normal **put** operation passing the following flags:

- The **ZERO_LOCK_ACQUISITION_TIMEOUT** flag: JBoss Data Grid uses an almost zero lock acquisition time when loading data from an external source into a cache.
- The **FAIL_SILENTLY** flag: If the locks cannot be acquired, JBoss Data Grid fails silently without throwing any lock acquisition exceptions.
- The **FORCE_ASYNCHRONOUS** flag: If clustered, the cache replicates asynchronously, irrespective of the cache mode set. As a result, a response from other nodes is not required.

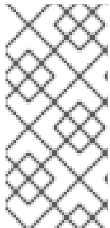
Combining the flags above significantly increases the efficiency of the operation. The basis for this efficiency is that **putForExternalRead** calls of this type are used because the client can retrieve the required data from a persistent store if the data cannot be found in memory. If the client encounters a cache miss, it retries the operation.

Report a bug

1.2.2. Configure Per-Invocation Flags

To use per-invocation flags in Red Hat JBoss Data Grid, add the required flags to the advanced cache via the **withFlags()** method call. For example:

```
Cache cache = ...
 cache.getAdvancedCache()
    .withFlags(Flag.SKIP_CACHE_STORE, Flag.CACHE_MODE_LOCAL)
    .put("local", "only");
```



NOTE

The called flags only remain active for the duration of the cache operation. To use the same flags in multiple invocations within the same transaction, use the **withFlags()** method for each invocation. If the cache operation must be replicated onto another node, the flags are also carried over to the remote nodes.

Report a bug

1.2.3. Per-Invocation Flags Example

In a use case for Red Hat JBoss Data Grid, where a write operation, such as **put()**, must not return the previous value, the ***IGNORE_RETURN_VALUES*** flag is used. This flag prevents a remote lookup (to get the previous value) in a distributed environment, which in turn prevents the retrieval of the undesired, potential, previous value. Additionally, if the cache is configured with a cache loader, this flag prevents the previous value from being loaded from its cache store.

An example of using the ***IGNORE_RETURN_VALUES*** flag is:

```
Cache cache = ...
 cache.getAdvancedCache()
    .withFlags(Flag.IGNORE_RETURN_VALUES)
    .put("local", "only")
```

Report a bug

1.3. THE ADVANCEDCACHE INTERFACE

Red Hat JBoss Data Grid offers an **AdvancedCache** interface, geared towards extending JBoss Data Grid, in addition to its simple Cache Interface. The **AdvancedCache** Interface can:

- Inject custom interceptors
- Access certain internal components
- Apply flags to alter the behavior of certain cache methods

The following code snippet presents an example of how to obtain an **AdvancedCache**:

```
AdvancedCache advancedCache = cache.getAdvancedCache();
```

Report a bug

1.3.1. Flag Usage with the AdvancedCache Interface

Flags, when applied to certain cache methods in Red Hat JBoss Data Grid, alter the behavior of the target method. Use **AdvancedCache.withFlags()** to apply any number of flags to a cache invocation, for example:

```
advancedCache.withFlags(Flag.CACHE_MODE_LOCAL, Flag.SKIP_LOCKING)
   .withFlags(Flag.FORCE_SYNCHRONOUS)
   .put("hello", "world");
```

Report a bug

1.3.2. Custom Interceptors and the AdvancedCache Interface

The **AdvancedCache** Interface provides a mechanism that allows advanced developers to attach custom interceptors. Custom interceptors can alter the behavior of the Cache API methods and the **AdvacedCache** Interface can be used to attach such interceptors programmatically at run time.

Report a bug

1.3.3. Custom Interceptors

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

Report a bug

1.3.3.1. Custom Interceptor Design

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

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

Report a bug

1.3.3.2. Adding Custom Interceptors Declaratively

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

A custom interceptor can be added using XML. Use the following procedure to add custom interceptors.

Procedure 1.1. Adding Custom Interceptors

1. **Define Custom Interceptors**
   All custom interceptors must extend org.jboss.cache.interceptors.base.CommandInterceptor. Use the customInterceptors method to add custom interceptors to the cache:

```
<namedCache name="cacheWithCustomInterceptors">
   <customInterceptors>
```

2. **Define the Position of the New Custom Interceptor**
   Interceptors must have a defined position. Valid options are:

   - **FIRST** - Specifies that the new interceptor is placed first in the chain.
   - **LAST** - Specifies that the new interceptor is placed last in the chain.
   - **OTHER_THAN_FIRST_OR_LAST** - Specifies that the new interceptor can be placed anywhere except first or last in the chain.

```
<namedCache name="cacheWithCustomInterceptors">
   <customInterceptors>
      <interceptor position="FIRST"
class="com.mycompany.CustomInterceptor1">
```

   - **Define Interceptor Properties**
     Define specific interceptor properties.

```
<namedCache name="cacheWithCustomInterceptors">
   <customInterceptors>
      <interceptor position="FIRST"
class="com.mycompany.CustomInterceptor1">
         <properties>
            <property name="attributeOne" value="value1" />
            <property name="attributeTwo" value="value2" />
         </properties>
      </interceptor>
```

3. **Apply Other Custom Interceptors**
   In this example, the next custom interceptor is called CustomInterceptor2.

```
<namedCache name="cacheWithCustomInterceptors">
   <customInterceptors>
      <interceptor position="FIRST"
class="com.mycompany.CustomInterceptor1">
         <properties>
            <property name="attributeOne" value="value1" />
            <property name="attributeTwo" value="value2" />
         </properties>
      </interceptor>
      <interceptor position="LAST"
class="com.mycompany.CustomInterceptor2"/>
```

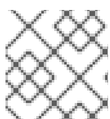
4. **Define the `index`, `before`, and `after` Attributes.**

   - The **`index`** identifies the position of this interceptor in the chain, with 0 being the first position. This attribute is mutually exclusive with **`position`**, **`before`**, and **`after`**.

   - The **`after`** method places the new interceptor directly after the instance of the named interceptor specified via its fully qualified class name. This attribute is mutually exclusive with **`position`**, **`before`**, and **`index`**.

   - The **`before`** method places the new interceptor directly before the instance of the named interceptor specified via its fully qualified class name. This attribute is mutually exclusive with **`position`**, **`after`**, and **`index`**.

```
<namedCache name="cacheWithCustomInterceptors">
   <customInterceptors>
      <interceptor position="FIRST"
class="com.mycompany.CustomInterceptor1">
         <properties>
            <property name="attributeOne" value="value1" />
            <property name="attributeTwo" value="value2" />
         </properties>
      </interceptor>
      <interceptor position="LAST"
class="com.mycompany.CustomInterceptor2"/>
      <interceptor index="3"
class="com.mycompany.CustomInterceptor1"/>
      <interceptor
before="org.infinispan.interceptors.CallInterceptor"
class="com.mycompany.CustomInterceptor2"/>
      <interceptor
after="org.infinispan.interceptors.CallInterceptor"
class="com.mycompany.CustomInterceptor1"/>
   </customInterceptors>
</namedCache>
```



NOTE

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

Report a bug

1.3.3.3. Adding Custom Interceptors Programmatically

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

For example:

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

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

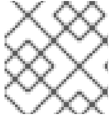
For example:

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

Report a bug

CHAPTER 2. THE BATCHING API

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



NOTE

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

Report a bug

2.1. ABOUT JAVA TRANSACTION API TRANSACTIONS

Red Hat JBoss Data Grid supports configuring, use of, and participation in Java Transaction API (JTA) compliant transactions.

JBoss Data Grid does the following for each cache operation:

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

Report a bug

2.2. BATCHING AND THE JAVA TRANSACTION API (JTA)

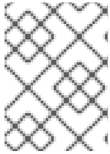
In Red Hat JBoss Data Grid, the batching functionality initiates a JTA transaction in the back end, causing all invocations within the scope to be associated with it. For this purpose, the batching functionality uses a simple Transaction Manager implementation at the back end. As a result, the following behavior is observed:

1. Locks acquired during an invocation are retained until the transaction commits or rolls back.
2. All changes are replicated in a batch on all nodes in the cluster as part of the transaction commit process. Ensuring that multiple changes occur within the single transaction, the replication traffic remains lower and improves performance.
3. When using synchronous replication or invalidation, a replication or invalidation failure causes the transaction to roll back.
4. If a `CacheLoader` that is compatible with a JTA resource, for example a JTA `DataSource`, is used for a transaction, the JTA resource can also participate in the transaction.
5. All configurations related to a transaction apply for batching as well.

An example of a transaction related configuration that can be applied for batching is as follows:

```
<transaction syncRollbackPhase="false"
      syncCommitPhase="false"
      useEagerLocking="true"
      eagerLockSingleNode="true" />
```

The configuration attributes can be used for both transactions and batching, using different values.



NOTE

Batching functionality and JTA transactions are only supported in JBoss Data Grid's Library Mode.

Report a bug

2.3. USING THE BATCHING API

2.3.1. Enable the Batching API

Red Hat JBoss Data Grid's Batching API uses the JBoss Enterprise Application Platform syntax to enable invocation batching in your cache configuration. An example of this is as follows:

```
<distributed-cache name="default" batching="true" statistics="true">
...
</distributed-cache>
```

In JBoss Data Grid, invocation batching is disabled by default and batching can be used without a defined Transaction Manager.

Report a bug

2.3.2. Configure the Batching API

To use the Batching API, enable invocation batching in the cache configuration.

XML Configuration

To configure the Batching API in the XML file:

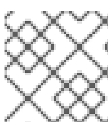
```
<invocationBatching enabled="true" />
```

Programmatic Configuration

To configure the Batching API programmatically use:

```
Configuration c = new
ConfigurationBuilder().invocationBatching().enable().build();
```

In Red Hat JBoss Data Grid, invocation batching is disabled by default and batching can be used without a defined Transaction Manager.



NOTE

Programmatic configurations can only be used with JBoss Data Grid's Library mode.

Report a bug

2.3.3. Use the Batching API

After the cache is configured to use batching, call **startBatch()** and **endBatch()** on the cache as follows to use batching:

```
Cache cache = cacheManager.getCache();
```

Example 2.1. Without Using Batch

```
cache.put("key", "value");
```

When the **cache.put(*key*, *value*);** line executes, the values are replaced immediately.

Example 2.2. Using Batch

```
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(true);
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(false);
```

When the line **cache.endBatch(true);** executes, all modifications made since the batch started are replicated.

When the line **cache.endBatch(false);** executes, changes made in the batch are discarded.

Report a bug

2.3.4. Batching API Usage Example

A simple use case that illustrates the Batching API usage is one that involves transferring money between two bank accounts.

Example 2.3. Batching API Usage Example

Red Hat JBoss Data Grid is used for a transaction that involves transferring money from one bank account to another. If both the source and destination bank accounts are located within JBoss Data Grid, a Batching API is used for this transaction. However, if one account is located within JBoss Data Grid and the other in a database, distributed transactions are required for the transaction.

Report a bug

CHAPTER 3. THE GROUPING API

The Grouping API can relocate groups of entries to a specified node or to a node selected using the hash of the group.

Report a bug

3.1. GROUPING API OPERATIONS

Normally, Red Hat JBoss Data Grid uses the hash of a specific key to determine an entry's destination node. However, when the Grouping API is used, a hash of the group associated with the key is used instead of the hash of the key to determine the destination node.

Each node can use an algorithm to determine the owner of each key. This removes the need to pass metadata (and metadata updates) about the location of entries between nodes. This approach is beneficial because:

- Every node can determine which node owns a particular key without expensive metadata updates across nodes.
- Redundancy is improved because ownership information does not need to be replicated if a node fails.

When using the Grouping API, each node must be able to calculate the owner of an entry. As a result, the group cannot be specified manually and must be either:

- Intrinsic to the entry, which means it was generated by the key class.
- Extrinsic to the entry, which means it was generated by an external function.

Report a bug

3.2. GROUPING API USE CASE

This feature allows logically related data to be stored on a single node. For example, if the cache contains user information, the information for all users in a single location can be stored on a single node.

The benefit of this approach is that when seeking specific (logically related) data, the Distributed Executor task is directed to run only on the relevant node rather than across all nodes in the cluster. Such directed operations result in optimized performance.

> **Example 3.1. Grouping API Example**
>
> Acme, Inc. is a home appliance company with over one hundred offices worldwide. Some offices house employees from various departments, while certain locations are occupied exclusively by the employees of one or two departments. The Human Resources (HR) department has employees in Bangkok, London, Chicago, Nice and Venice.
>
> Acme, Inc. uses Red Hat JBoss Data Grid's Grouping API to ensure that all the employee records for the HR department are moved to a single node (Node AB) in the cache. As a result, when attempting to retrieve a record for a HR employee, the `DistributedExecutor` only checks node AB and quickly and easily retrieves the required employee records.

Storing related entries on a single node as illustrated optimizes the data access and prevents time and resource wastage by seeking information on a single node (or a small subset of nodes) instead of all the nodes in the cluster.

Report a bug

3.3. CONFIGURE THE GROUPING API

Use the following steps to configure the Grouping API:

1. Enable groups using either the declarative or programmatic method.
2. Specify either an intrinsic or extrinsic group. For more information about these group types, see Section 3.1, “Grouping API Operations”
3. Register all specified groupers.

Report a bug

3.3.1. Enable Groups

The first step to set up the Grouping API is to enable groups. In Red Hat JBoss Data Grid, groups are enabled declaratively or programmatically as follows:

Declaratively Enable Groups

Use the following configuration to enable groups using XML:

```
<clustering>
  <hash>
     <groups enabled="true" />
  </hash>
</clustering>
```

Programmatically Enable Groups

Use the following to enable groups programmatically:

```
Configuration c = new
ConfigurationBuilder().clustering().hash().groups().enabled().build();
```

Report a bug

3.3.2. Specify an Intrinsic Group

Use an intrinsic group with the Grouping API if:

- the key class definition can be altered, that is if it is not part of an unmodifiable library.
- if the key class is not concerned with the determination of a key/value pair group.

Use the `@Group` annotation in the relevant method to specify an intrinsic group. The group must always be a String, as illustrated in the example:

Example 3.2. Specifying an Intrinsic Group Example

```
class User {

   ...
   String office;
   ...

   public int hashCode()) {
      // Defines the hash for the key, normally used to determine
location
      ...
   }

   // Override the location by specifying a group, all keys in the same
   // group end up with the same owner
   @Group
   String getOffice() {
      return office;
   }

}
```

Report a bug

3.3.3. Specify an Extrinsic Group

Specify an extrinsic group for the Grouping API if:

- the key class definition cannot be altered, that is if it is part of an unmodifiable library.
- if the key class is concerned with the determination of a key/value pair group.

An extrinsic group is specified using an implementation of the **Grouper** interface. This interface uses the **computeGroup** method to return the group.

In the process of specifying an extrinsic group, the **Grouper** interface acts as an interceptor by passing the computed value to **computeGroup**. If the **@Group** annotation is used, the group using it is passed to the first **Grouper**. As a result, using an intrinsic group provides even greater control.

Example 3.3. Specifying an Extrinsic Group Example

The following is an example that consists of a simple **Grouper** that uses the key class to extract the group from a key using a pattern. Any group information specified on the key class is ignored in such a situation.

```
public class KXGrouper implements Grouper<String> {

   // A pattern that can extract from a "kX" (e.g. k1, k2) style key
   // The pattern requires a String key, of length 2, where the first
character is
   // "k" and the second character is a digit. We take that digit, and
perform
```

```
    // modular arithmetic on it to assign it to group "1" or group "2".

    private static Pattern kPattern = Pattern.compile("(^k)(\\d)$");

     public String computeGroup(String key, String group) {
         Matcher matcher = kPattern.matcher(key);
         if (matcher.matches()) {
             String g = Integer.parseInt(matcher.group(2)) % 2 + "";
             return g;
         } else
             return null;
     }

     public Class<String> getKeyType() {
         return String.class;
     }

}
```

Report a bug

3.3.4. Register Groupers

After creation, each grouper must be registered to be used.

Declaratively Register a Grouper

Use the following code to register a grouping using XML:

```
<clustering>
  <hash>
     <groups enabled="true">
        <grouper class="com.acme.KXGrouper" />
     </groups>
  </hash>
</clustering>
```

Programmatically Register a Grouper

Use the following code to register a grouper programmatically:

```
Configuration c = new
ConfigurationBuilder().clustering().hash().groups().addGrouper(new
KXGrouper()).enabled().build();
```

Report a bug

CHAPTER 4. THE PERSISTENCE SPI

In Red Hat JBoss Data Grid, persistence can configure external (persistent) storage engines. These storage engines complement JBoss Data Grid's default in-memory storage.

Persistent external storage provides several benefits:

- Memory is volatile and a cache store can increase the life span of the information in the cache, which results in improved durability.
- Using persistent external stores as a caching layer between an application and a custom storage engine provides improved Write-Through functionality.
- Using a combination of eviction and passivation, only the frequently required information is stored in-memory and other data is stored in the external storage.

Report a bug

4.1. PERSISTENCE SPI BENEFITS

The Red Hat JBoss Data Grid implementation of the Persistence SPI offers the following benefits:

- Alignment with JSR-107 (http://jcp.org/en/jsr/detail?id=107). JBoss Data Grid's **CacheWriter** and **CacheLoader** interfaces are similar to the JSR-107 writer and reader. As a result, alignment with JSR-107 provides improved portability for stores across JCache-compliant vendors.
- Simplified transaction integration. JBoss Data Grid handles locking automatically and so implementations do not have to coordinate concurrent access to the store. Depending on the locking mode, concurrent writes on the same key may not occur. However, implementors expect operations on the store to originate from multiple threads and add the implementation code accordingly.
- Reduced serialization, resulting in reduced CPU usage. The new SPI exposes stored entries in a serialized format. If an entry is fetched from persistent storage to be sent remotely, it does not need to be serialized (when reading from the store) and then serialized again (when writing to the wire). Instead, the entry is written to the wire in the serialized format as fetched from the storage.

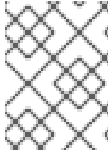
Report a bug

4.2. PROGRAMMATICALLY CONFIGURE THE PERSISTENCE SPI

The following is a sample programmatic configuration for a Single File Store using the Persistence SPI:

```
ConfigurationBuilder builder = new ConfigurationBuilder();
   builder.persistence()
         .passivation(false)
         .addSingleFileStore()
            .preload(true)
            .shared(false)
            .fetchPersistentState(true)
            .ignoreModifications(false)
            .purgeOnStartup(false)
            .location(System.getProperty("java.io.tmpdir"))
```

```
        .async()
            .enabled(true)
            .threadPoolSize(5)
        .singleton()
            .enabled(true)
            .pushStateWhenCoordinator(true)
            .pushStateTimeout(20000);
```



NOTE

Programmatic configurations can only be used with Red Hat JBoss Data Grid's Library mode.

Report a bug

CHAPTER 5. THE CONFIGURATIONBUILDER API

The ConfigurationBuilder API is a programmatic configuration API in Red Hat JBoss Data Grid.

The ConfigurationBuilder API is designed to assist with:

- Chain coding of configuration options in order to make the coding process more efficient
- Improve the readability of the configuration

In JBoss Data Grid, the ConfigurationBuilder API is also used to enable CacheLoaders and configure both global and cache level operations.

Report a bug

5.1. USING THE CONFIGURATIONBUILDER API

5.1.1. Programmatically Create a CacheManager and Replicated Cache

Programmatic configuration in Red Hat JBoss Data Grid almost exclusively involves the ConfigurationBuilder API and the CacheManager. The following is an example of a programmatic CacheManager configuration:

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-
file.xml");
Cache defaultCache = manager.getCache();
Configuration c = new
ConfigurationBuilder().clustering().cacheMode(CacheMode.REPL_SYNC)
.build();

String newCacheName = "repl";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

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

Procedure 5.1. Steps for Programmatic Configuration in JBoss Data Grid

1. Create a CacheManager as a starting point in an XML file. If required, this CacheManager can be programmed in runtime to the specification that meets the requirements of the use case. The following is an example of how to create a CacheManager:

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

2. Create a new synchronously replicated cache programmatically.

   a. Create a new configuration object instance using the ConfigurationBuilder helper object:

   ```
   Configuration c = new
   ConfigurationBuilder().clustering().cacheMode(CacheMode.REPL_SYNC
   )
   .build();
   ```

In the first line of the configuration, a new cache configuration object (named **c**) is created using the **ConfigurationBuilder**. Configuration **c** is assigned the default values for all cache configuration options except the cache mode, which is overridden and set to synchronous replication (**REPL_SYNC**).

b. Set the cache mode to synchronous replication:

```
String newCacheName = "repl";
```

In the second line of the configuration, a new variable (of type **String**) is created and assigned the value **repl**.

c. Define or register the configuration with a manager:

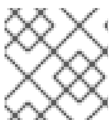
```
manager.defineConfiguration(newCacheName, c);
```

In the third line of the configuration, the cache manager is used to define a named cache configuration for itself. This named cache configuration is called **repl** and its configuration is based on the configuration provided for cache configuration **c** in the first line.

d.

```
Cache<String, String> cache = manager.getCache(newCacheName);
```

In the fourth line of the configuration, the cache manager is used to obtain a reference to the unique instance of the **repl** that is held by the cache manager. This cache instance is now ready to be used to perform operations to store and retrieve data.



NOTE

Programmatic configurations can only be used with JBoss Data Grid's Library mode.

Report a bug

5.1.2. Create a Customized Cache Using the Default Named Cache

The default cache configuration (or any customized configuration) can serve as a starting point to create a new cache.

As an example, if the **infinispan-config-file.xml** specifies the configuration for a replicated cache as a default and a distributed cache with a customized lifespan value is required. The required distributed cache must retain all aspects of the default cache specified in the **infinispan-config-file.xml** file except the mentioned aspects.

The following is an example of a customized default cache configuration:

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration dcc = cacheManager.getDefaultCacheConfiguration();
Configuration c = new ConfigurationBuilder().read(dcc).clustering()
.cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000L).enable()
```

```
.build();
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

An explanation of the provided configuration is as follows:

Procedure 5.2. Customize the Default Cache

1. Read an instance of a default Configuration object to get the default configuration:

   ```
   EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-
   config-file.xml");
   Configuration dcc = cacheManager.getDefaultCacheConfiguration();
   ```

2. Use the ConfigurationBuilder to construct and modify the cache mode and L1 cache lifespan on a new configuration object:

   ```
   Configuration c = new ConfigurationBuilder().read(dcc).clustering()
   .cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000L).enable()
   .build();
   ```

3. Register/define your cache configuration with a cache manager, where cacheName is name of cache specified in **`infinispan-config-file.xml`**:

   ```
   manager.defineConfiguration(newCacheName, c);
   ```

4. Get default cache with custom configuration changes:

   ```
   Cache<String, String> cache = manager.getCache(newCacheName);
   ```

Report a bug

5.1.3. Create a Customized Cache Using a Non-Default Named Cache

A situation can arise where a new customized cache must be created using a named cache that is not the default. The steps to accomplish this are similar to those used when using the default named cache for this purpose.

The difference in approach is due to taking a named cache called **`replicatedCache`** as the base instead of the default cache. The following is an example of a customized cache using a non-default named cache:

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-
file.xml");
Configuration rc = cacheManager.getCacheConfiguration("replicatedCache");
Configuration c = new ConfigurationBuilder().read(rc).clustering()
.cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000L).enable()
.build();
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

An explanation of the provided example is as follows:

Procedure 5.3. Create a Customized Cache Using a Non-Default Named Cache

1. Read the `replicatedCache` to get the default configuration:

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-
config-file.xml");
Configuration rc =
cacheManager.getCacheConfiguration("replicatedCache");
```

2. Use the ConfigurationBuilder to construct and modify the desired configuration on a new configuration object:

```
Configuration c = new ConfigurationBuilder().read(rc).clustering()
.cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000L).enable()
.build();
```

3. Register/define your cache configuration with a cache manager where *newCacheName* is the name of cache specified in `infinispan-config-file.xml`

```
manager.defineConfiguration(newCacheName, c);
```

4. Get a default cache with custom configuration changes:

```
Cache<String, String> cache = manager.getCache(newCacheName);
```

Report a bug

5.1.4. Using the Configuration Builder to Create Caches Programmatically

As an alternative to using an xml file with default cache values to create a new cache, use the ConfigurationBuilder API to create a new cache without any XML files. The ConfigurationBuilder API is intended to provide ease of use when creating chained code for configuration options.

The following new configuration is valid for global and cache level configuration. GlobalConfiguration objects are constructed using GlobalConfigurationBuilder while Configuration objects are built using ConfigurationBuilder.

Report a bug

5.1.5. Global Configuration Examples

5.1.5.1. Globally Configure the Transport Layer

A commonly used configuration option is to configure the transport layer. This informs Red Hat JBoss Data Grid how a node will discover other nodes:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .globalJmxStatistics().enable()
  .build();
```

Report a bug

5.1.5.2. Globally Configure the Cache Manager Name

The following sample configuration allows you to use options from the global JMX statistics level to configure the name for a cache manager. This name distinguishes a particular cache manager from other cache managers on the same system.

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .globalJmxStatistics()
    .cacheManagerName("SalesCacheManager")
    .mBeanServerLookup(new JBossMBeanServerLookup())
    .enable()
  .build();
```

Report a bug

5.1.5.3. Globally Customize Thread Pool Executors

Some Red Hat JBoss Data Grid features are powered by a group of thread pool executors. These executors can be customized at the global level as follows:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .replicationQueueScheduledExecutor()
    .factory(new DefaultScheduledExecutorFactory())
    .addProperty("threadNamePrefix", "RQThread")
  .build();
```

Report a bug

5.1.6. Cache Level Configuration Examples

5.1.6.1. Cache Level Configuration for the Cluster Mode

The following configuration allows the use of options such as the cluster mode for the cache at the cache level rather than globally:

```
Configuration config = new ConfigurationBuilder()
  .clustering()
    .cacheMode(CacheMode.DIST_SYNC)
    .sync()
    .l1().lifespan(25000L).enable()
    .hash().numOwners(3)
  .build();
```

Report a bug

5.1.6.2. Cache Level Eviction and Expiration Configuration

Use the following configuration to configure expiration or eviction options for a cache at the cache level:

```
Configuration config = new ConfigurationBuilder()
           .eviction()
```

```
.maxEntries(20000).strategy(EvictionStrategy.LIRS).expiration()
             .wakeUpInterval(5000L)
             .maxIdle(120000L)
           .build();
```

Report a bug

5.1.6.3. Cache Level Configuration for JTA Transactions

To interact with a cache for JTA transaction configuration, configure the transaction layer and optionally customize the locking settings. For transactional caches, it is recommended to enable transaction recovery to deal with unfinished transactions. Additionally, it is recommended that JMX management and statistics gathering is also enabled.

```
Configuration config = new ConfigurationBuilder()
  .locking()

.concurrencyLevel(10000).isolationLevel(IsolationLevel.REPEATABLE_READ)

.lockAcquisitionTimeout(12000L).useLockStriping(false).writeSkewCheck(true
)
  .transaction()
    .transactionManagerLookup(new GenericTransactionManagerLookup())
  .recovery().enable()
  .jmxStatistics().enable()
  .build();
```

Report a bug

5.1.6.4. Cache Level Configuration Using Chained Persistent Stores

The following configuration can be used to configure one or more chained persistent stores at the cache level:

```
Configuration config = new ConfigurationBuilder()
       .persistence()
       .passivation(false)
.addSingleFileStore().shared(false).preload(false).location("/tmp").async(
).enable().threadPoolSize(20).build();
```

Report a bug

5.1.6.5. Cache Level Configuration for Advanced Externalizers

An advanced option such as a cache level configuration for advanced externalizers can also be configured programmatically as follows:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .serialization()
    .addAdvancedExternalizer(new PersonExternalizer())
    .addAdvancedExternalizer(999, new AddressExternalizer())
  .build();
```

Report a bug

CHAPTER 6. THE EXTERNALIZABLE API

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.

The Externalizable interface uses and extends serialization. This interface is used to control serialization and deserialization in JBoss Data Grid.

Report a bug

6.1. CUSTOMIZE EXTERNALIZERS

As a default in Red Hat JBoss Data Grid, all objects used in a distributed or replicated cache must be serilizable. The default Java serialization mechanism can result in network and performance inefficiency. Additional concerns include serialization versioning and backwards compatibility.

For enhanced throughput, performance or to enforce specific object compatibility, use a customized externalizer. Customized externalizers for JBoss Data Grid can be used in one of two ways:

- Use an Externalizable Interface. For details, see the Red Hat JBoss Data Grid *Developer Guide*'s *The Externalizable API* chapter.
- Use an advanced externalizer.

Report a bug

6.2. ANNOTATING OBJECTS FOR MARSHALLING USING @SERIALIZEWITH

Objects can be marshalled by providing an Externalizer implementation for the type that needs to be marshalled or unmarshalled, then annotating the marshalled type class with **`@SerializeWith`** indicating the Externalizer class to use.

For example:

```
import org.infinispan.commons.marshall.Externalizer;
import org.infinispan.commons.marshall.SerializeWith;

@SerializeWith(Person.PersonExternalizer.class)
public class Person {

   final String name;
   final int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
```

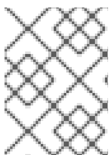
```
    }

    public static class PersonExternalizer implements Externalizer<Person>
{
        @Override
        public void writeObject(ObjectOutput output, Person person)
              throws IOException {
           output.writeObject(person.name);
           output.writeInt(person.age);
        }

        @Override
        public Person readObject(ObjectInput input)
              throws IOException, ClassNotFoundException {
           return new Person((String) input.readObject(), input.readInt());
        }
    }
}
```

In the provided example, the object has been defined as marshallable due to the **@SerializeWith** annotation. JBoss Marshalling will therefore marshall the object using the Externalizer class passed.

This method of defining externalizers is user friendly, however it has the following disadvantages:

- The payload sizes generated using this method are not the most efficient. This is due to some constraints in the model, such as support for different versions of the same class, or the need to marshall the Externalizer class.
- This model requires the marshalled class to be annotated with **@SerializeWith**, however an Externalizer may need to be provided for a class for which source code is not available, or for any other constraints, it cannot be modified.
- Annotations used in this model may be limiting for framework developers or service providers that attempt to abstract lower level details, such as the marshalling layer, away from the user.

Advanced Externalizers are available for users affected by these disadvantages.



NOTE

To make Externalizer implementations easier to code and more typesafe, define type <t> as the type of object that is being marshalled or unmarshalled.

Report a bug

6.3. USING AN ADVANCED EXTERNALIZER

Using a customized advanced externalizer helps optimize performance in Red Hat JBoss Data Grid.

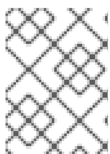
1. Define and implement the **readObject()** and **writeObject()** methods.
2. Link externalizers with marshaller classes.
3. Register the advanced externalizer.

Report a bug

6.3.1. Implement the Methods

To use advanced externalizers, define and implement the **`readObject()`** and **`writeObject()`** methods. The following is a sample definition:

```
import org.infinispan.commons.marshall.AdvancedExternalizer;

public class Person {

   final String name;
   final int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public static class PersonExternalizer implements
AdvancedExternalizer<Person> {
      @Override
      public void writeObject(ObjectOutput output, Person person)
            throws IOException {
         output.writeObject(person.name);
         output.writeInt(person.age);
      }

      @Override
      public Person readObject(ObjectInput input)
            throws IOException, ClassNotFoundException {
         return new Person((String) input.readObject(), input.readInt());
      }

      @Override
      public Set<Class<? extends Person>> getTypeClasses() {
         return Util.<Class<? extends Person>>asSet(Person.class);
      }

      @Override
      public Integer getId() {
         return 2345;
      }
   }
}
```



NOTE

This method does not require annotated user classes. As a result, this method is valid for classes where the source code is not available or cannot be modified.

Report a bug

6.3.2. Link Externalizers with Marshaller Classes

Use an implementation of **getTypeClasses()** to discover the classes that this externalizer can marshall and to link the **readObject()** and **writeObject()** classes.

The following is a sample implementation:

```
import org.infinispan.util.Util;
...
@Override
public Set<Class<? extends ReplicableCommand>> getTypeClasses() {
  return Util.asSet(LockControlCommand.class, GetKeyValueCommand.class,
        ClusteredGetCommand.class, MultipleRpcCommand.class,
        SingleRpcCommand.class, CommitCommand.class,
        PrepareCommand.class, RollbackCommand.class,
        ClearCommand.class, EvictCommand.class,
        InvalidateCommand.class, InvalidateL1Command.class,
        PutKeyValueCommand.class, PutMapCommand.class,
        RemoveCommand.class, ReplaceCommand.class);
}
```

In the provided sample, the **ReplicableCommandExternalizer** indicates that it can externalize several command types. This sample marshalls all commands that extend the **ReplicableCommand** interface but the framework only supports class equality comparison so it is not possible to indicate that the classes marshalled are all children of a particular class or interface.

In some cases, the class to be externalized is private and therefore the class instance is not accessible. In such a situation, look up the class with the provided fully qualified class name and pass it back. An example of this is as follows:

```
@Override
public Set<Class<? extends List>> getTypeClasses() {
 return Util.<Class<? extends List>>asSet(
       Util.<List>loadClass("java.util.Collections$SingletonList", null));
}
```

Report a bug

6.3.3. Register the Advanced Externalizer (Declaratively)

After the advanced externalizer is set up, register it for use with Red Hat JBoss Data Grid. This registration is done declaratively (via XML) as follows:

Procedure 6.1. Register the Advanced Externalizer

1. Add the **global** element to the **infinispan** element:

```
<infinispan>
  <global />
</infinispan>
```

2. Add the **serialization** element to the **global** element as follows:

```
<infinispan>
  <global>
    <serialization />
```

```
  </global>
</infinispan>
```

3. Add the **`advancedExternalizers`** element to add information about the new advanced externalizer as follows:

```
<infinispan>
  <global>
    <serialization>
      <advancedExternalizers />
    </serialization>
  </global>
</infinispan>
```

4. Define the externalizer class using the ***`externalizerClass`*** attribute as follows:

```
<infinispan>
    <global>
        <serialization>
            <advancedExternalizers>
                <advancedExternalizer
externalizerClass="org.infinispan.marshall.AdvancedExternalizerTest$
IdViaAnnotationObj$Externalizer"/>
            </advancedExternalizers>
        </serialization>
    </global>
</infinispan>
```

Replace the *$IdViaAnnotationObj* and *$AdvancedExternalizer* values as required.

Report a bug

6.3.4. Register the Advanced Externalizer (Programmatically)

After the advanced externalizer is set up, register it for use with Red Hat JBoss Data Grid. This registration is done programmatically as follows:

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
   .addAdvancedExternalizer(new Person.PersonExternalizer());
```

Enter the desired information for the GlobalConfigurationBuilder in the first line.

Report a bug

6.3.5. Register Multiple Externalizers

Alternatively, register multiple advanced externalizers because **`GlobalConfiguration.addExternalizer()`** accepts ***`varargs`***. Before registering the new externalizers, ensure that their IDs are already defined using the **`@Marshalls`** annotation.

```
builder.serialization()
   .addAdvancedExternalizer(new Person.PersonExternalizer(),
                            new Address.AddressExternalizer());
```

Report a bug

6.4. CUSTOM EXTERNALIZER ID VALUES

Advanced externalizers can be assigned custom IDs if desired. Some ID ranges are reserved for other modules or frameworks and must be avoided:

Table 6.1. Reserved Externalizer ID Ranges

ID Range	Reserved For
1000-1099	The Infinispan Tree Module
1100-1199	Red Hat JBoss Data Grid Server modules
1200-1299	Hibernate Infinispan Second Level Cache
1300-1399	JBoss Data Grid Lucene Directory
1400-1499	Hibernate OGM
1500-1599	Hibernate Search/Infinispan Query

Report a bug

6.4.1. Customize the Externalizer ID (Declaratively)

Customize the advanced externalizer ID declaratively (via XML) as follows:

Procedure 6.2. Customizing the Externalizer ID (Declaratively)

1. Add the **global** element to the **infinispan** element:

```
<infinispan>
  <global />
</infinispan>
```

2. Add the **serialization** element to the **global** element as follows:

```
<infinispan>
  <global>
    <serialization />
  </global>
</infinispan>
```

3. Add the **advancedExternalizer** element to add information about the new advanced externalizer as follows:

```
<infinispan>
```

```
  <global>
    <serialization>
      <advancedExternalizer />
    </serialization>
  </global>
</infinispan>
```

4. Define the externalizer ID using the ***id*** attribute as follows:

```
<infinispan>
  <global>
    <serialization>
      <advancedExternalizers>
       <advancedExternalizer id="$ID" />
      </advancedExternalizers>
    </serialization>
  </global>
</infinispan>
```

Ensure that the selected ID is not from the range of IDs reserved for other modules.

5. Define the externalizer class using the ***externalizerClass*** attribute as follows:

```
<infinispan>
  <global>
    <serialization>
      <advancedExternalizers>
       <advancedExternalizer id="$ID"
externalizerClass="org.infinispan.marshall.AdvancedExternalizerTest$
IdViaConfigObj$Externalizer"/>
       </advancedExternalizers>
    </serialization>
  </global>
</infinispan>
```

Replace the *$IdViaAnnotationObj* and *$AdvancedExternalizer* values as required.

Report a bug

6.4.2. Customize the Externalizer ID (Programmatically)

Use the following configuration to programmatically assign a specific ID to the externalizer:

```
GlobalConfiguration globalConfiguration = new GlobalConfigurationBuilder()
            .serialization()
               .addAdvancedExternalizer($ID, new
Person.PersonExternalizer())
            .build();
```

Replace the *$ID* with the desired ID.

Report a bug

CHAPTER 7. THE NOTIFICATION/LISTENER API

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

Report a bug

7.1. LISTENER EXAMPLE

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

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

Report a bug

7.2. CACHE ENTRY MODIFIED LISTENER CONFIGURATION

In a cache entry modified listener event, The ***`getValue()`*** method's behavior is specific to whether the callback is triggered before or after the actual operation has been performed. For example, if **`event.isPre()`** is true, then **`event.getValue()`** would return the old value, prior to modification. If **`event.isPre()`** is false, then **`event.getValue()`** would return new value. If the event is creating and inserting a new entry, the old value would be null. For more information about ***`isPre()`***, see the Red Hat JBoss Data Grid *API Documentation*'s listing for the **`org.infinispan.notifications.cachelistener.event`** package.

Listeners can only be configured programmatically by using the methods exposed by the Listenable and FilteringListenable interfaces (which the Cache object implements).

Report a bug

7.3. LISTENER NOTIFICATIONS

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

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

Report a bug

7.3.1. About Cache-level Notifications

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

Report a bug

7.3.2. Cache Manager-level Notifications

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

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

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

Report a bug

7.3.3. About Synchronous and Asynchronous Notifications

By default, notifications in Red Hat JBoss Data Grid are dispatched in the same thread that generates the event. Therefore the listener must be written in a way that does not block or prevent the thread's progression.

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

Annotate listeners using the following:

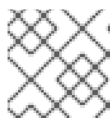
```
@Listener (sync = false)public class MyAsyncListener { .... }
```

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

Report a bug

7.4. NOTIFYINGFUTURES

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



NOTE

NotifyingFutures are only available in JBoss Data Grid Library mode.

Report a bug

7.4.1. NotifyingFutures Example

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

```
FutureListener futureListener = new FutureListener() {

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

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

Report a bug

PART II. 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 8. 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

8.1. 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:

```
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

8.2. 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)

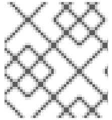
- Locking

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

Report a bug

8.3. 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 9. 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

9.1. 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.

```
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

9.2. 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

9.3. 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.

```
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

9.4. JAVA CLIENT CODE

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

Imports

Define imports as follows:

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

Add a String Value to a Cache

The following is an example of using Java to add 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();
   }
```

Get a String Value from a Cache

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.

```
   /**
    * 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();
    }
```

The following code is an example of 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

9.5. CONFIGURE THE INTERFACE USING CONNECTORS

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.

Report a bug

9.5.1. Configure REST Connectors

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

Procedure 9.1. 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:6.0">
      <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:6.0">
      <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:6.0">
      <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:6.0">
   <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 **DIGEST**, **CLIENT-CERT** and **SPNEGO**. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.0">
   <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:6.0">
   <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

9.6. 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

9.6.1. 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

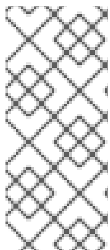
9.6.1.1. About PUT /{cacheName}/{cacheKey}

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

9.6.1.2. 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

9.6.2. 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

9.6.2.1. 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

9.6.2.2. 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

9.6.3. 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

9.6.3.1. 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

9.6.3.2. 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

9.6.3.3. 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

9.6.4. REST Interface Operation Headers

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

Table 9.1. 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.
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.

Headers	Mandatory/Optional	Values	Default Value	Details
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 9.2. 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.

Header	Algorithm	Example	Details
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.
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

9.7. REST INTERFACE SECURITY

9.7.1. Publish REST Endpoints as a Public Interface

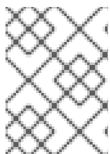
Red Hat JBoss Data Grid's REST server operates as a management interface by default. To extend its operations to a public interface, alter the value of the ***interface*** parameter in the **socket-binding** element from **management** to **public** as follows:

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

Report a bug

9.7.2. 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 9.2. 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:6.0">
               <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 WildFly 7 or 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:

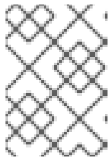
   ```
   user1=2dc3eacfed8cf95a4a31159167b936fc
   ```

   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 10. 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

10.1. 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

10.2. MEMCACHED STATISTICS

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

Table 10.1. 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

10.3. CONFIGURE THE INTERFACE USING CONNECTORS

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.

Report a bug

10.3.1. 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 10.1. 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:6.0">
<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:6.0">
<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:6.0">
<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:6.0">
<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:6.0">
<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:6.0">
<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:6.0">
<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

10.4. MEMCACHED INTERFACE SECURITY

10.4.1. 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 11. THE HOT ROD INTERFACE

11.1. 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

11.2. 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

11.3. HOT ROD HASH FUNCTIONS

Red Hat JBoss Data Grid uses a consistent hash function to place nodes and, subsequently, their corresponding keys on a hash wheel to determine where entries live.

The hash space value is constant and limited to the maximum 32-bit positive integer value (*Integer.MAX_INT*). This value is returned to the client using the Hot Rod protocol each time a hash-topology change is detected to prevent Hot Rod clients assuming a specific hash space as a default. The hash space can only contain positive numbers ranging from `0` to *Integer.MAX_INT*.

When the Hot Rod protocol is used to interact with JBoss Data Grid, the keys (and their values) must be byte arrays to ensure platform neutral behavior. Smart clients (which are aware of hash distribution in the background) must be able to recalculate hash codes of such byte array keys in this platform-neutral manner. To accommodate this, version information for hash functions used in JBoss Data Grid is saved for implementation by non-Java clients, if required.

Client Intelligence Levels

The following table describes the client intelligence levels:

Table 11.1. Client Intelligence Levels Definition

Intelligence Level	Description
Level 1	In this level, simple clients connect to the grid using a list of statically provided server addresses. These addresses are used based on round-robin scheduling.
Level 2	In this level, clients are aware of the topology of the grid and are notified when new servers are added or removed from the grid.
Level 3	In this level, clients use the grid topology and the key hashes to directly connect to the node that is the primary owner for a data item. Thus reducing the need for remote calls between the server nodes.

Report a bug

11.4. HOT ROD SERVER NODES

11.4.1. About Consistent Hashing Algorithms

Consistent hashing algorithms arrange the hash space as a circle. Owners are assigned for segments of the hash space. When a key is assigned to an owner, the hash of the key is used to determine in which segment the key should be stored. If an owner is removed, then its segment is allocated to its neighbors in the circle. This means that if an owner is removed, most of the hash space remains stable, resulting in less overhead.

Report a bug

11.5. HOT ROD HEADERS

11.5.1. 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 11.2. 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

11.5.2. Request Header

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

Table 11.3. 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.

Field Name	Data Type/Size	Details
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.
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`.

Field Name	Data Type/Size	Details
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

11.5.3. Response Header

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

Table 11.4. 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

11.5.4. 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

11.5.4.1. Topology Change Marker Values

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

Table 11.5. Topology Change Marker Field Values

Value	Details
0	No topology change information is added.
1	Topology change information is added.

Report a bug

11.5.4.2. 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 11.6. 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.

Response Header Fields	Data Type/Size	Details
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

11.5.4.3. 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 11.7. 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.

Field	Data Type/Size	Details
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.
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

11.6. HOT ROD OPERATIONS

11.6.1. 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

Report a bug

11.6.2. Hot Rod BulkGetKeys Operation

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

Table 11.8. BulkGetKeys Operation Request Format

Field	Data Type	Details
Header	variable	Request header.

Field	Data Type	Details
Scope	vInt	`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. `1` = Global Scope - This scope behaves the same to Default Scope. `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.

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

Table 11.9. 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

11.6.3. Hot Rod BulkGet Operation

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

Table 11.10. 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 11.11. BulkGet Operation Response Format

Field	Data Type	Details
Header	-	-

Field	Data Type	Details
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.
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

11.6.4. Hot Rod Clear Operation

The **clear** operation format includes only a header.

Valid response statuses for this operation are as follows:

Table 11.12. Clear Operation Response

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

Report a bug

11.6.5. Hot Rod ContainsKey Operation

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

Table 11.13. 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 11.14. 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

11.6.6. Hot Rod Get Operation

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

Table 11.15. 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 11.16. 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 11.17. 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

11.6.7. Hot Rod GetWithMetadata Operation

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

Table 11.18. 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 11.19. GetWithMetadata Operation Response Format

Field	Data Type	Details
Header	variable	Response header.
Response status	1 byte	`0x00` = success, if key retrieved. `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

11.6.8. 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 11.20. Ping Operation Response

Response Status	Details
0x00	Successful ping without any errors.

Report a bug

11.6.9. Hot Rod PutIfAbsent Operation

The `putIfAbsent` operation request format includes the following:

Table 11.21. 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 11.22.

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`.

Report a bug

11.6.10. Hot Rod Put Operation

The `put` operation request format includes the following:

Table 11.23.

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 11.24.

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`.

Report a bug

11.6.11. Hot Rod Query Operation

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

Table 11.25. 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 11.26. Query Operation Response

Response Status	Data	Details
Header	variable	Response header.
Response payload Length	vInt	The length of the Protobuf encoded response object.
Response payload	Byte array	Byte array containing the Protobuf encoded response object, having a length specified by previous field.

Report a bug

11.6.12. Hot Rod RemoveIfUnmodified Operation

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

Table 11.27. 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.

Field	Data Type	Details
Entry Version	8 bytes	Uses the value returned by the **GetWithVersion** operation.

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

Table 11.28. 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

11.6.13. Hot Rod Remove Operation

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

Table 11.29. 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 11.30. 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

11.6.14. Hot Rod ReplaceIfUnmodified Operation

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

Table 11.31. ReplaceIfUnmodified 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.

Field	Data Type	Details
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	Uses the value returned by the `GetWithVersion` operation.
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 11.32. ReplaceIfUnmodified 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

11.6.15. Hot Rod Replace Operation

The `replace` operation request format includes the following:

Table 11.33. Replace Operation Request Fields

Field	Data Type	Details
Header	-	-
Key Length	vInt	Contains the length of the key.

Field	Data Type	Details
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 11.34. 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**.

Report a bug

11.6.16. 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 11.35. 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.
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 11.36. 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

11.7. HOT ROD OPERATION VALUES

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

Table 11.37. Opcode Request and Response Header Values

Operation	Request Operation Code	Response Operation Code
put	0x01	0x02
get	0x03	0x04
putIfAbsent	0x05	0x06
replace	0x07	0x08
replaceIfUnmodified	0x09	0x0A
remove	0x0B	0x0C
removeIfUnmodified	0x0D	0x0E
containsKey	0x0F	0x10
getWithVersion	0x11	0x12
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

11.7.1. Magic Values

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

Table 11.38. Magic Field Values

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

Report a bug

11.7.2. Status Values

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

Table 11.39. 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

11.7.3. Transaction Type Values

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

Table 11.40. Transaction Type Field Values

Value	Details

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

11.7.4. Client Intelligence Values

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

Table 11.41. 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

11.7.5. Flag Values

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

Table 11.42. Flag Field Values

Value	Details
0x0001	ForceReturnPreviousValue

Report a bug

11.7.6. Hot Rod Error Handling

Table 11.43. 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

11.8. PUT REQUEST EXAMPLE

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

Table 11.44. 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 11.45. Example Request Field Names and Values

Field Name	Byte	Value
Magic	0	0xA0

Field Name	Byte	Value
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
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 11.46. 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 11.47. 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

11.9. 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

11.9.1. Hot Rod Java Client Download

The Hot Rod Java client is included under the **client/hotrod/java** directory in the Red Hat JBoss Data Grid 6.2 and better Server binary distributions on the Red Hat Customer Portal at https://access.redhat.com

Report a bug

11.9.2. 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 11.1. 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 11.2. 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



## below is connection pooling config

maxActive=-1

maxTotal = -1

maxIdle = -1

whenExhaustedAction = 1

timeBetweenEvictionRunsMillis=120000

minEvictableIdleTimeMillis=300000
```

```
testWhileIdle = true

minIdle = 1
```

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

11.9.3. 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 11.3. 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

11.9.4. Hot Rod Java Client Versioned API

To ensure data consistency, Hot Rod stores a version number that uniquely identifies each modification. Using a ***getVersioned*** or ***getWithVersion***, 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.

The following example shows the usage of 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");
RemoteCache.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");
```

Similarly for replace:

```
remoteCache.put("car", "ferrari");
RemoteCache.VersionedValue valueBinary = remoteCache.getVersioned("car");
assert remoteCache.replace("car", "lamborghini",
valueBinary.getVersion());
```

Report a bug

11.10. HOT ROD C ++ CLIENT

The Hot Rod C ++ client is a new addition to the Hot Rod client family which includes Java Hot Rod 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 connect and query or store data in JBoss Hot Rod remote caches. The Hot Rod C++ client supports all three levels of client intelligence. The Hot Rod C ++ client can be compiled for Linux, Unix, and Windows operating systems.

Report a bug

11.10.1. Hot Rod C ++ Client Formats

The Hot Rod C++ client is available in the following two library formats:

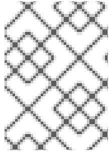
- Static library
- Shared/Dynamic library

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

11.10.2. 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 6.2 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

11.10.3. Hot Rod C ++ Client Configuration

The C++ Hot Rod 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 11.4. 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

11.10.4. 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 11.5. 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

11.10.5. Hot Rod C ++ Client Requisites

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

11.11. INTEROPERABILITY BETWEEN C++ AND HOT ROD JAVA CLIENT

Red Hat JBoss Data Grid provides interoperability between Hot Rod Java and C++ Hot Rod 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 C++ Hot Rod 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.

```
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

11.12. CONFIGURE THE INTERFACE USING CONNECTORS

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.

Report a bug

11.12.1. 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 11.1. 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:6.0">
      <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:6.0">
      <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 the number of cores available multiplied by two. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.0">
      <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:6.0">
      <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:6.0">
      <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:6.0">
      <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:6.0">
      <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:6.0">
      <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:6.0">
      <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 *update-timeout* Parameter**

The ***update-timeout*** parameter specifies the time (in milliseconds) after which the update operation times out. The default value for this parameter is **30** seconds. This is an optional parameter.

```
<subsystem xmlns="urn:infinispan:server:endpoint:6.0">
      <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="${MILLISECONDS}"
                               update-
timeout="${MILLISECONDS}"/>
```

d. **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:6.0">
      <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="${MILLISECONDS}"
                               update-timeout="${MILLISECONDS}"
                               external-host="${HOSTNAME}" />
```

e. **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:6.0">
      <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="${MILLISECONDS}"
                               update-timeout="${MILLISECONDS}"
                               external-host="${HOSTNAME}"
                               external-port="${PORT}" />
```

f. **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:6.0">
 <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="${MILLISECONDS}"
     update-timeout="${MILLISECONDS}"
     external-host="${HOSTNAME}"
     external-port="${PORT}"
     lazy-retrieval="${TRUE/FALSE}" />
</subsystem>
```

Report a bug

11.13. HOT ROD INTERFACE SECURITY

11.13.1. Publish Hot Rod Endpoints as a Public Interface

Red Hat JBoss Data Grid's Hot Rod server operates as a management interface as a 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="hotrod" interface="public" port="11222" />
```

Report a bug

11.13.2. SSL/TLS Authentication for Hot Rod

Hot Rod can be encrypted using SSL, and has the option to require client certification authentication.

Use the following procedure to secure the Hot Rod connector using SSL.

Procedure 11.2. Secure Hot Rod Using SSL/TLS

1. **Generate a Keystore**
   Create a Java Keystore using the keytool application distributed with the JDK and add your certificate to it. The certificate can be either self signed, or obtained from a trusted CA depending on your security policy.

2. **Place the Keystore in the Configuration Directory**
   Put the keystore in the **~/JDG_HOME/standalone/configuration** directory with the **standalone-hotrod-ssl.xml** file from the **~/JDG_HOME/docs/examples/configs** directory.

3. **Declare an SSL Server Identity**
   Declare an SSL server identity within a security realm in the management section of the configuration file. The SSL server identity must specify the path to a keystore and its secret key.

```
<server-identities>
  <ssl protocol="...">
    <keystore path="..." relative-to="..." keystore-password="..."
alias="..." key-password="..." />
  </ssl>
  <secret value="..." />
</server-identities>
```

4. **Add the Security Element**
   Add the security element to the Hot Rod connector as follows:

```
<hotrod-connector socket-binding="hotrod" cache-container="local">
    <security ssl="true" security-realm="ApplicationRealm" require-
ssl-client-auth="false" />
</hotrod-connector>
```

   a. **Server Authentication of Certificate**
      If you require the server to perform authentication of the client certificate, create a truststore that contains the valid client certificates and set the ***require-ssl-client-auth*** attribute to **true**.

5. **Start the Server**
   Start the server using the following:

```
bin/standalone.sh -c standalone-hotrod-ssl.xml
```

   This will start a server with a Hot Rod endpoint on port 11222. This endpoint will only accept SSL connections.

Report a bug

PART III. ADVANCED FEATURES IN RED HAT JBOSS DATA GRID

Red Hat JBoss Data Grid includes some advanced features. Some of these features are listed in the *JBoss Data Grid Administration and Configuration Guide* with instructions for a basic configuration. The *JBoss Data Grid Developer Guide* explores these features and others in more detail and with instructions for more advanced usage.

The advanced features explored in this guide are as follows:

- Transactions
- Marshalling
- Listeners and Notifications
- The Infinspan CDI Module
- Rolling Upgrades
- MapReduce
- Distributed Execution
- Interoperability and Compatibility Mode

Report a bug

CHAPTER 12. TRANSACTIONS

12.1. ABOUT JAVA TRANSACTION API TRANSACTIONS

Red Hat JBoss Data Grid supports configuring, use of, and participation in Java Transaction API (JTA) compliant transactions.

JBoss Data Grid does the following for each cache operation:

1. First, it retrieves the transactions currently associated with the thread.

2. If not already done, it registers an **XAResource** with the transaction manager to receive notifications when a transaction is committed or rolled back.

Report a bug

12.2. TRANSACTIONS SPANNING MULTIPLE CACHE INSTANCES

Each cache operates as a separate, standalone Java Transaction API (JTA) resource. However, components can be internally shared by 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

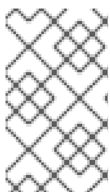
12.3. THE TRANSACTION MANAGER

Use the following to obtain the TransactionManager from the cache:

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

To execute a sequence of operations within transaction, wrap these with calls to methods begin() and commit() or rollback() on the TransactionManager:

```
tm.begin();
Object value = cache.get("A");
cache.remove("A");
Object prev = cache.put("B", value);
if (prev == null)
    tm.commit();
else
    tm.rollback();
```



NOTE

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

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

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

Report a bug

CHAPTER 13. MARSHALLING

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

Red Hat JBoss Data Grid uses marshalling and unmarshalling to:

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

Report a bug

13.1. ABOUT MARSHALLING FRAMEWORK

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

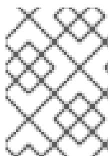
The Java Marshalling Framework uses high performance **`java.io.ObjectOutput`** and **`java.io.ObjectInput`** implementations compared to the standard **`java.io.ObjectOutputStream`** and **`java.io.ObjectInputStream`**.

Report a bug

13.2. SUPPORT FOR NON-SERIALIZABLE OBJECTS

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

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



NOTE

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

Report a bug

13.3. HOT ROD AND MARSHALLING

In Remote Client-Server mode, marshalling occurs both on the Red Hat JBoss Data Grid server and the client levels, but to varying degrees.

- All data stored by clients on the JBoss Data Grid server are provided either as a byte array, or in a primitive format that is marshalling compatible for JBoss Data Grid.

  On the server side of JBoss Data Grid, marshalling occurs where the data stored in primitive format are converted into byte array and replicated around the cluster or stored to a cache

store. No marshalling configuration is required on the server side of JBoss Data Grid.

- At the client level, marshalling must have a ***Marshaller*** configuration element specified in the RemoteCacheManager configuration in order to serialize and deserialize POJOs.

  Due to Hot Rod's binary nature, it relies on marshalling to transform POJOs, specifically keys or values, into byte array.

Report a bug

13.4. CONFIGURING THE MARSHALLER USING THE REMOTECACHEMANAGER

A Marshaller is specified using the ***marshaller*** configuration element in the RemoteCacheManager, the value of which must be the name of the class implementing the Marshaller interface. The default value for this property is ***org.infinispan.commons.marshall.jboss.GenericJBossMarshaller***.

The following example shows the default value the JBoss Marshaller.

```
ConfigurationBuilder builder = new ConfigurationBuilder();
//... (other configuration)
builder.marshaller(GenericJBossMarshaller.class);
// builder.marshaller(new GenericJBossMarshaller()); //(alternative)

Configuration configuration = builder.build();
RemoteCacheManager manager = new RemoteCacheManager(configuration);
```

The following procedure describes how to define a Marshaller to use with RemoteCacheManager.

Procedure 13.1. Define a Marshaller

1. **Create a ConfigurationBuilder**
   Create a ConfigurationBuilder and configure it with the required settings.

   ```
   ConfigurationBuilder builder = new ConfigurationBuilder();
   //... (other configuration)
   ```

2. **Add a Marshaller Class**
   Add a Marshaller class specification within the Marshaller method.

   ```
   builder.marshaller(GenericJBossMarshaller.class);
   ```

   - Alternatively, specify a custom Marshaller instance.

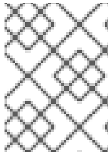
     ```
     builder.marshaller(new GenericJBossMarshaller());
     ```

3. **Start the RemoteCacheManager**
   Build the configuration containing the Marshaller, and start a new RemoteCacheManager with it.

   ```
   Configuration configuration = builder.build();
   RemoteCacheManager manager = new RemoteCacheManager(configuration);
   ```

-

At the client level, POJOs need to be either Serializable, Externalizable, or primitive types.



NOTE

The Java Hot Rod client does not support providing Externalizer instances to serialize POJOs. This is only available for JBoss Data Grid Library mode.

Report a bug

13.5. TROUBLESHOOTING

13.5.1. Marshalling Troubleshooting

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

The following is an example of such a stack trace:

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

```
in object java.lang.Object@b40ec4
in object org.infinispan.commands.write.PutKeyValueCommand@df661da7
... Removed 22 stack frames
```

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

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

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

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

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

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

```
VersionAwareMarshallerTest.java:431)
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.VersionAwareMarshallerTest$1
```

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

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

```
java.io.IOException: Injected failue!
...
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.VersionAwareMarshallerTest$1
-> classloader hierarchy:
-> type classloader = sun.misc.Launcher$AppClassLoader@198dfaf
-
>...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/ecl
ipse-testng.jar
-
>...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/lib
/testng-jdk15.jar
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/test-
classes/
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/classes/
->...file:/home/galder/.m2/repository/org/testng/testng/5.9/testng-5.9-
jdk15.jar
->...file:/home/galder/.m2/repository/net/jcip/jcip-annotations/1.0/jcip-
annotations-1.0.jar
-
>...file:/home/galder/.m2/repository/org/easymock/easymockclassextension/2
.4/easymockclassextension-2.4.jar
->...file:/home/galder/.m2/repository/org/easymock/easymock/2.4/easymock-
2.4.jar
->...file:/home/galder/.m2/repository/cglib/cglib-nodep/2.1_3/cglib-nodep-
2.1_3.jar
->...file:/home/galder/.m2/repository/javax/xml/bind/jaxb-api/2.1/jaxb-
api-2.1.jar
->...file:/home/galder/.m2/repository/javax/xml/stream/stax-api/1.0-
2/stax-api-1.0-2.jar
-
>...file:/home/galder/.m2/repository/javax/activation/activation/1.1/activ
ation-1.1.jar
->...file:/home/galder/.m2/repository/jgroups/jgroups/2.8.0.CR1/jgroups-
2.8.0.CR1.jar
->...file:/home/galder/.m2/repository/org/jboss/javaee/jboss-transaction-
api/1.0.1.GA/jboss-transaction-api-1.0.1.GA.jar
-
>...file:/home/galder/.m2/repository/org/jboss/marshalling/river/1.2.0.CR4
-SNAPSHOT/river-1.2.0.CR4-SNAPSHOT.jar
->...file:/home/galder/.m2/repository/org/jboss/marshalling/marshalling-
api/1.2.0.CR4-SNAPSHOT/marshalling-api-1.2.0.CR4-SNAPSHOT.jar
->...file:/home/galder/.m2/repository/org/jboss/jboss-common-
core/2.2.14.GA/jboss-common-core-2.2.14.GA.jar
->...file:/home/galder/.m2/repository/org/jboss/logging/jboss-logging-
```

```
spi/2.0.5.GA/jboss-logging-spi-2.0.5.GA.jar
->...file:/home/galder/.m2/repository/log4j/log4j/1.2.14/log4j-1.2.14.jar
-
>...file:/home/galder/.m2/repository/com/thoughtworks/xstream/xstream/1.2/
xstream-1.2.jar
->...file:/home/galder/.m2/repository/xpp3/xpp3_min/1.1.3.4.O/xpp3_min-
1.1.3.4.O.jar
->...file:/home/galder/.m2/repository/com/sun/xml/bind/jaxb-
impl/2.1.3/jaxb-impl-2.1.3.jar
-> parent classloader = sun.misc.Launcher$ExtClassLoader@1858610
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/localedata.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunpkcs11.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunjce_provider.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/dnsns.jar
... Removed 22 stack frames
```

Report a bug

13.5.2. Other Marshalling Related Issues

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

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

```
pawner$1.run(STREAMING_STATE_TRANSFER.java:582)
     at
java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.
java:886)
     at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java
:908)
     at java.lang.Thread.run(Thread.java:680)
Caused by: java.util.concurrent.TimeoutException: Could not obtain
exclusive processing lock
     at
org.infinispan.remoting.transport.jgroups.JGroupsDistSync.acquireProcessin
gLock(JGroupsDistSync.java:71)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.generateTransactionL
og(StateTransferManagerImpl.java:202)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.generateState(StateT
ransferManagerImpl.java:165)
     ... 12 more
```

In logs, you can also spot exceptions which seems to be related to marshaling. However, the root cause of the exception can be different. The implication of this exception is that the state generator was unable to generate the transaction log hence the output it was writing in now closed. In such a situation, the state receiver will often log an ***EOFException***, displayed as follows, when failing to read the transaction log that was not written by the sender:

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

```
org.infinispan.statetransfer.StateTransferManagerImpl.applyTransactionLog(
StateTransferManagerImpl.java:250)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.applyState(StateTran
sferManagerImpl.java:320)
     at
org.infinispan.remoting.InboundInvocationHandlerImpl.applyState(InboundInv
ocationHandlerImpl.java:102)
     at
org.infinispan.remoting.transport.jgroups.JGroupsTransport.setState(JGroup
sTransport.java:603)
        ...
```

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

Report a bug

CHAPTER 14. LISTENERS AND NOTIFICATIONS

14.1. THE NOTIFICATION/LISTENER API

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

Report a bug

14.2. LISTENER NOTIFICATIONS

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

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

Report a bug

14.2.1. About Cache-level Notifications

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

Report a bug

14.2.2. Cache Manager-level Notifications

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

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

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

Report a bug

14.2.3. About Synchronous and Asynchronous Notifications

By default, notifications in Red Hat JBoss Data Grid are dispatched in the same thread that generates the event. Therefore the listener must be written in a way that does not block or prevent the thread's progression.

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

Annotate listeners using the following:

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

–

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

Report a bug

14.3. MODIFYING CACHE ENTRIES

After the cache entry has been created, the cache entry can be modified programmatically.

Report a bug

14.3.1. Cache Entry Modified Listener Configuration

In a cache entry modified listener event, The ***getValue()*** method's behavior is specific to whether the callback is triggered before or after the actual operation has been performed. For example, if **event.isPre()** is true, then **event.getValue()** would return the old value, prior to modification. If **event.isPre()** is false, then **event.getValue()** would return new value. If the event is creating and inserting a new entry, the old value would be null. For more information about ***isPre()***, see the Red Hat JBoss Data Grid *API Documentation*'s listing for the **org.infinispan.notifications.cachelistener.event** package.

Listeners can only be configured programmatically by using the methods exposed by the Listenable and FilteringListenable interfaces (which the Cache object implements).

Report a bug

14.3.2. Listener Example

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

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

Report a bug

14.3.3. Cache Entry Modified Listener Example

The following example defines a listener in Red Hat JBoss Data Grid that prints some information each time a cache entry is modified:

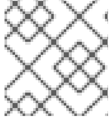
```
@Listener
public class PrintWhenModified {
    @CacheEntryModified
    public void print(CacheEntryModifiedEvent event) {
```

```
            System.out.println("Cache entry modified. Details = " + event");
          }
      }
```

Report a bug

14.4. NOTIFYINGFUTURES

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



NOTE

NotifyingFutures are only available in JBoss Data Grid Library mode.

Report a bug

14.4.1. NotifyingFutures Example

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

```
FutureListener futureListener = new FutureListener() {

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

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

Report a bug

CHAPTER 15. THE INFINSPAN CDI MODULE

Infinispan includes Context and Dependency Injection (CDI) in the **infinispan-cdi** module. The **infinispan-cdi** module offers:

- Configuration and injection using the Cache API.
- A bridge between the cache listeners and the CDI event system.
- Partial support for the JCACHE caching annotations.

Report a bug

15.1. USING INFINISPAN CDI

15.1.1. Infinispan CDI Prerequisites

The following is a list of prerequisites to use the Infinispan CDI module with Red Hat JBoss Data Grid:

- Ensure that the most recent version of the infinispan-cdi module is used.
- Ensure that the correct dependency information is set.

Report a bug

15.1.2. Set the CDI Maven Dependency

Add the following dependency information to the **pom.xml** file in your maven project:

```
<dependencies>
...
 <dependency>
     <groupId>org.infinispan</groupId>
     <artifactId>infinispan-cdi</artifactId>
     <version>${infinispan.version}</version>
 </dependency>
...
</dependencies>
```

If Maven is not in use, the **infinispan-cdi** jar file is available at **modules/infinispan-cdi** in the ZIP distribution.

Report a bug

15.2. USING THE INFINISPAN CDI MODULE

The Infinispan CDI module can be used for the following purposes:

- To configure and inject Infinispan caches into CDI Beans and Java EE components.
- To configure cache managers.
- To control storage and retrieval using CDI annotations.

Report a bug

15.2.1. Configure and Inject Infinispan Caches

15.2.1.1. Inject an Infinispan Cache

An Infinispan cache is one of the multiple components that can be injected into the project's CDI beans.

The following code snippet illustrates how to inject a cache instance into the CDI bean:

```
public class MyCDIBean {
    @Inject
    Cache<String, String> cache;
}
```

Report a bug

15.2.1.2. Inject a Remote Infinispan Cache

The code snippet to inject a normal cache is slightly modified to inject a remote Infinispan cache, as follows:

```
public class MyCDIBean {
    @Inject
    RemoteCache<String, String> remoteCache;
}
```

Report a bug

15.2.1.3. Set the Injection's Target Cache

The following are the three steps to set an injection's target cache:

1. Create a qualifier annotation.
2. Add a producer class.
3. Inject the desired class.

Report a bug

15.2.1.3.1. Create a Qualifier Annotation

To use CDI to return a specific cache, create custom cache qualifier annotations as follows:

```
@javax.inject.Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface SmallCache {}
```

Use the created **`@SmallCache`** qualifier to specify how to create specific caches.

Report a bug

15.2.1.3.2. Add a Producer Class

The following code snippet illustrates how the **@SmallCache** qualifier (created in the previous step) specifies a way to create a cache:

```
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.cdi.ConfigureCache;
import javax.enterprise.inject.Proces;

public class CacheCreator {
    @ConfigureCache("smallcache")
    @SmallCache
    @Produces
    public Configuration specialCacheCfg() {
        return new ConfigurationBuilder()
                   .eviction()
                       .strategy(EvictionStrategy.LRU)
                       .maxEntries(10)
                   .build();
    }
}
```

The elements in the code snippet are:

- **@ConfigureCache** specifies the name of the cache.
- **@SmallCache** is the cache qualifier.

Report a bug

15.2.1.3.3. Inject the Desired Class

Use the **@SmallCache** qualifier and the new producer class to inject a specific cache into the CDI bean as follows:

```
public class MyCDIBean {
    @Inject @SmallCache
    Cache<String, String> mySmallCache;
}
```

Report a bug

15.2.2. Configure Cache Managers with CDI

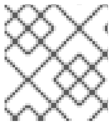
A Red Hat JBoss Data Grid Cache Manager (both embedded and remote) can be configured using CDI. Whether configuring an embedded or remote cache manager, the first step is to specify a default configuration that is annotated to act as a producer.

Report a bug

15.2.2.1. Specify the Default Configuration

Specify a method annotated as a producer for the Red Hat JBoss Data Grid configuration object to replace the default Infinispan Configuration. The following sample configuration illustrates this step:

```
public class Config {
    @Produces
    public Configuration defaultEmbeddedConfiguration () {
        return new ConfigurationBuilder()
                            .eviction()
                                    .strategy(EvictionStrategy.LRU)
                                    .maxEntries(100)
                            .build();
    }
}
```



NOTE

CDI adds a **@Default** qualifier if no other qualifiers are provided.

If a **@Produces** annotation is placed in a method that returns a Configuration instance, the method is invoked when a Configuration object is required.

In the provided example configuration, the method creates a new Configuration object which is subsequently configured and returned.

Report a bug

15.2.2.2. Override the Creation of the Embedded Cache Manager

Prerequisites

Section 15.2.2.1, "Specify the Default Configuration"

Creating Non Clustered Caches

After a producer method is annotated, this method will be called when creating an **EmbeddedCacheManager**, as follows:

```
public class Config {

    @Produces
    @ApplicationScoped
    public EmbeddedCacheManager defaultEmbeddedCacheManager() {
        Configuration cfg = new ConfigurationBuilder()
                                    .eviction()

.strategy(EvictionStrategy.LRU)
                                            .maxEntries(150)
                                    .build();
        return new DefaultCacheManager(cfg);
    }
}
```

The **@ApplicationScoped** annotation specifies that the method is only called once.

Creating Clustered Caches

The following configuration can be used to create an **EmbeddedCacheManager** that can create clustered caches.

```
public class Config {

 @Produces
 @ApplicationScoped
 public EmbeddedCacheManager defaultClusteredCacheManager() {
  GlobalConfiguration g = new GlobalConfigurationBuilder()
    .clusteredDefault()
    .transport()
    .clusterName("InfinispanCluster")
    .build();
  Configuration cfg = new ConfigurationBuilder()
    .eviction()
    .strategy(EvictionStrategy.LRU)
    .maxEntries(150)
    .build();
  return new DefaultCacheManager(g, cfg);
 }
}
```

Invoke the Method to Generate an EmbeddedCacheManager

The method annotated with **@Produces** in the non clustered method generates **Configuration** objects. The methods in the clustered cache example annonated with **@Produces** generate **EmbeddedCacheManager** objects.

Add an injection as follows in your CDI Bean to invoke the appropriate annotated method. This generates **EmbeddedCacheManager** and injects it into the code at runtime.

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

Report a bug

15.2.2.3. Configure a Remote Cache Manager

The **RemoteCacheManager** is configured in a manner similar to **EmbeddedCacheManagers**, as follows:

```
public class Config {
 @Produces
 @ApplicationScoped
 public RemoteCacheManager defaultRemoteCacheManager() {
           Configuration conf = new
ConfigurationBuilder().addServer().host(ADDRESS).port(PORT).build();
            return new RemoteCacheManager(conf);
 }
}}
```

Report a bug

15.2.2.4. Configure Multiple Cache Managers with a Single Class

A single class can be used to configure multiple cache managers and remote cache managers based on the created qualifiers. An example of this is as follows:

```
public class Config {
    @Produces
    @ApplicationScoped
    public org.infinispan.manager.EmbeddedCacheManager
    defaultEmbeddedCacheManager() {
        Configuration cfg = new ConfigurationBuilder()
                .eviction()
                .strategy(EvictionStrategy.LRU)
                .maxEntries(150)
                .build();
        return new DefaultCacheManager(cfg);
    }

    @Produces
    @ApplicationScoped
    @DefaultClustered
    public org.infinispan.manager.EmbeddedCacheManager
    defaultClusteredCacheManager() {
        GlobalConfiguration g = new GlobalConfigurationBuilder()
                .clusteredDefault()
                .transport()
                .clusterName("InfinispanCluster")
                .build();
        Configuration cfg = new ConfigurationBuilder()
                .eviction()
                .strategy(EvictionStrategy.LRU)
                .maxEntries(150)
                .build();
        return new DefaultCacheManager(g, cfg);
    }

    @Produces
    @ApplicationScoped
    @DefaultRemote
    public RemoteCacheManager
    defaultRemoteCacheManager() {
        org.infinispan.client.hotrod.configuration.Configuration conf =
new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder().addServe
r().host(ADDRESS).port(PORT).build();
        return new RemoteCacheManager(conf);
    }

    @Produces
    @ApplicationScoped
    @RemoteCacheInDifferentDataCentre
    public RemoteCacheManager newRemoteCacheManager() {
        org.infinispan.client.hotrod.configuration.Configuration confid =
new
org.infinispan.client.hotrod.configuration.ConfigurationBuilder().addServe
r().host(ADDRESS_FAR_AWAY).port(PORT).build();
```

```
            return new RemoteCacheManager(confid);
        }
    }
```

Report a bug

15.2.3. Storage and Retrieval Using CDI Annotations

15.2.3.1. Configure Cache Annotations

Specific CDI annotations are accepted for the JCache (JSR-107) specification. All included annotations are located in the **javax.cache** package.

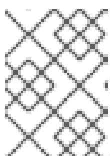
The annotations intercept method calls on CDI beans and perform storage and retrieval tasks as a result of these interceptions.

Report a bug

15.2.3.2. Enable Cache Annotations

Interceptors can be added to the CDI bean archive using the **beans.xml** file. Adding the following code adds interceptors such as the **CacheResultInterceptor**, **CachePutInterceptor**, **CacheRemoveEntryInterceptor** and the **CacheRemoveAllInterceptor**:

```
<beans xmlns="http://java.sun.som/xml/ns/javaee"
       xmlns:xsi="http://www/w3/org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/beans_1_0.xsd" >
 <interceptors>
  <class>
   org.infinispan.jcache.annotation.CacheResultInterceptor
  </class>
  <class>
   org.infinispan.jcache.annotation.CachePutInterceptor
  </class>
  <class>
   org.infinispan.jcache.annotation.CacheRemoveEntryInterceptor
  </class>
  <class>
   org.infinispan.jcache.annotation.CacheRemoveAllInterceptor
  </class>
 </interceptors>
</beans>
```



NOTE

The listed interceptors must appear in the **beans.xml** file for Red Hat JBoss Data Grid to use javax.cache annotations.

Report a bug

15.2.3.3. Caching the Result of a Method Invocation

A common practice for time or resource intensive operations is to save the results in a cache for future access. The following code is an example of such an operation:

```
public String toCelsiusFormatted(float fahrenheit) {
 return
  NumberFormat.getInstance()
  .format((fahrenheit * 5 / 9) - 32)
  + " degrees Celsius";
}
```

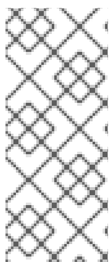
A common approach is to cache the results of this method call and to check the cache when the result is next required. The following is an example of a code snippet that looks up the result of such an operation in a cache. If the results are not found, the code snippet runs the **`toCelsiusFormatted`** method again and stores the result in the cache.

```
float f = getTemperatureInFahrenheit();
Cache<Float, String>
 fahrenheitToCelsiusCache = getCache();
String celsius = fahrenheitToCelsiusCache.get(f);
     if (celsius == null) {
      celsius = toCelsiusFormatted(f);
      fahrenheitToCelsiusCache.put(f, celsius);
     }
```

In such cases, the Infinispan CDI module can be used to eliminate all the extra code in the related examples. Annotate the method with the **`@CacheResult`** annotation from the javax.cache.annotation package instead, as follows:

```
@javax.cache.annotation.CacheResult
public String toCelsiusFormatted(float fahrenheit) {
 return NumberFormat.getInstance()
 .format((fahrenheit * 5 / 9) - 32)
 + " degrees Celsius";
}
```

Due to the annotation, Infinispan checks the cache and if the results are not found, it invokes the **`toCelsiusFormatted()`** method call.



NOTE

The Infinispan CDI module allows checking the cache for saved results, but this approach should be carefully considered before application. If the results of the call should always be fresh data, or if the cache reading requires a remote network lookup or deserialization from a cache loader, checking the cache before call method invocation can be counter productive.

Report a bug

15.2.3.3.1. Specify the Cache Used

Add the following optional attribute (***cacheName***) to the **`@CacheResult`** annotation to specify the cache to check for results of the method call:

```
@CacheResult(cacheName = "mySpecialCache")
public void doSomething(String parameter) {
...
}
```

Report a bug

15.2.3.3.2. Cache Keys for Cached Results

As a default, the **@CacheResult** annotation creates a key for the results fetched from a cache. The key consists of a combination of all parameters in the relevant method.

Create a custom key using the **@CacheKey** annotation as follows:

```
@CacheResult
public void doSomething
 (@CacheKey String p1,
 @CacheKey String p2,
 String dontCare) {
...
}
```

In the specified example, only the values of **p1** and **p2** are used to create the cache key. The value of **dontCare** is not used when determining the cache key.

Report a bug

15.2.3.3.3. Generate a Custom Key

Generate a custom key as follows:

```
import javax.cache.annotation.GeneratedCacheKey;
import javax.cache.annotation.CacheKeyGenerator;
import javax.cache.annotation.CacheKeyInvocationContext;
import java.lang.annotation.Annotation;

public class MyCacheKeyGenerator implements CacheKeyGenerator {

   @Override
   public GeneratedCacheKey generateCacheKey(CacheKeyInvocationContext<?
extends Annotation> ctx) {

      return new MyCacheKey(
            ctx.getAllParameters()[0].getValue()
      );
   }
}
```

The listed method constructs a custom key. This key is passed as part of the value generated by the first parameter of the invocation context.

To specify the custom key generation scheme, add the optional parameter ***cacheKeyGenerator*** to the **@CacheResult** annotation as follows:

```
@CacheResult(cacheKeyGenerator = MyCacheKeyGenerator.class)
public void doSomething(String p1, String p2) {
...
}
```

Using the provided method, **p1** contains the custom key.

Report a bug

15.2.3.3.4. CacheKey Implementation Code

A custom key generation scheme can be created to override the default key generation offered by the Infinispan CDI module.

Generate a custom key as follows:

```
import javax.cache.annotation.GeneratedCacheKey;

public class MyCacheKey implements GeneratedCacheKey {
   private Object p;

   public MyCacheKey(Object p) {
      this.p = p;
   }

   @Override
   public boolean equals(Object o) {
      ...
   }

   @Override
   public int hashCode() {
      ...
   }
}
```

The **equals()** and **hashCode()** methods must be correctly implemented for the **GeneratedCacheKey** to work as expected.

Report a bug

15.2.4. Cache Operations

15.2.4.1. Update a Cache Entry

When the method that contains the **@CachePut** annotation is invoked, a parameter (normally passed to the method annotated with **@CacheValue**) is stored in the cache.

The following is a sample of the **@CachePut** annotated method:

```
import javax.cache.annotation.CacheResult
@CachePut (cacheName = "personCache")
public void updatePerson
 (@CacheKey long personId,
```

```
 @CacheValue Person newPerson) {
...
 }
```

Further customization is possible using **cacheName** and **cacheKeyGenerator** in the **@CachePut** method. Additionally, some parameters in the invoked method may be annotated with **@CacheKey** to control key generation.

See Also:

- Section 15.2.3.3.2, "Cache Keys for Cached Results"

Report a bug

15.2.4.2. Remove an Entry from the Cache

The following is an example of a **@CacheRemoveEntry** annotated method and is used to remove an entry from the cache:

```
import javax.cache.annotation.CacheResult
@CacheRemoveEntry (cacheName = "cacheOfPeople")
public void changePersonName
 (@CacheKey long personId,
 string newName {
...
 }
```

The annotation accepts the optional **cacheName** and **cacheKeyGenerator** attributes.

Report a bug

15.2.4.3. Clear the Cache

Invoke the **@CacheRemoveAll** method to clear all entries from the cache. An example of a method annotated with **@CacheRemoveAll** is as follows

```
import javax.cache.annotation.CacheResult
@CacheRemoveAll (cacheName = "statisticsCache")
public void resetStatistics() {
...
}
```

As displayed in the example, this annotation accepts an optional **cacheName** attribute.

Report a bug

CHAPTER 16. 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 using Hot Rod.

Report a bug

16.1. 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 16.1. 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. **Create a Target Cluster**
   Start a new cluster (the target cluster) with the new version of JBoss Data Grid. Use either different network settings of a different JGroups cluster name to set it apart from the source cluster.

2. **Migrate Caches to the Target Cluster**
   To migrate multiple caches from the source and target cluster, 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.

3. **Restart Each Node**
   Configure each client to point to the target cluster instead of the source cluster. One by one, restart each node in the cluster to apply the new configuration. Eventually, the target cluster will handle all requests instead of the source cluster. The target cluster then lazily loads data from the source cluster on demand using the `RestCacheStore`.

4. **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.

5. **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.

6. **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 for all caches to be migrated.

   b. **Using the CLI**
      Run the `upgrade --disconnectsource=rest` command on the target cluster for all caches to be migrated. Optionally, use the `--all` switch to disconnect all caches in the cluster.

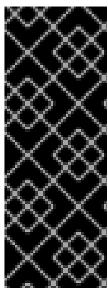
Result

Migration to the target cluster is complete. The source cluster can now be decommissioned

Report a bug

16.2. ROLLING UPGRADES USING HOT ROD (REMOTE CLIENT-SERVER MODE)

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

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. **Begin a New Cluster (Target Cluster)**
   Start the Target Cluster with the new version of JBoss Data Grid.

   Use either different network settings or JGroups cluster name to avoid overlap with the Source Cluster.

2. **Configure the Target Cluster with a `RemoteCacheStore`**

The Target Cluster is configured with a **RemoteCacheStore** with the following settings for each cache to be migrated:

1. ***servers*** must point to the Source Cluster.
2. ***cache name*** must coincide with the name of the cache on the Source Cluster.
3. ***hotrod-wrapping*** must be enabled (**"true"**).
4. ***purge*** must be disabled (**"false"**).
5. ***passivation*** must be disabled (**"false"**).

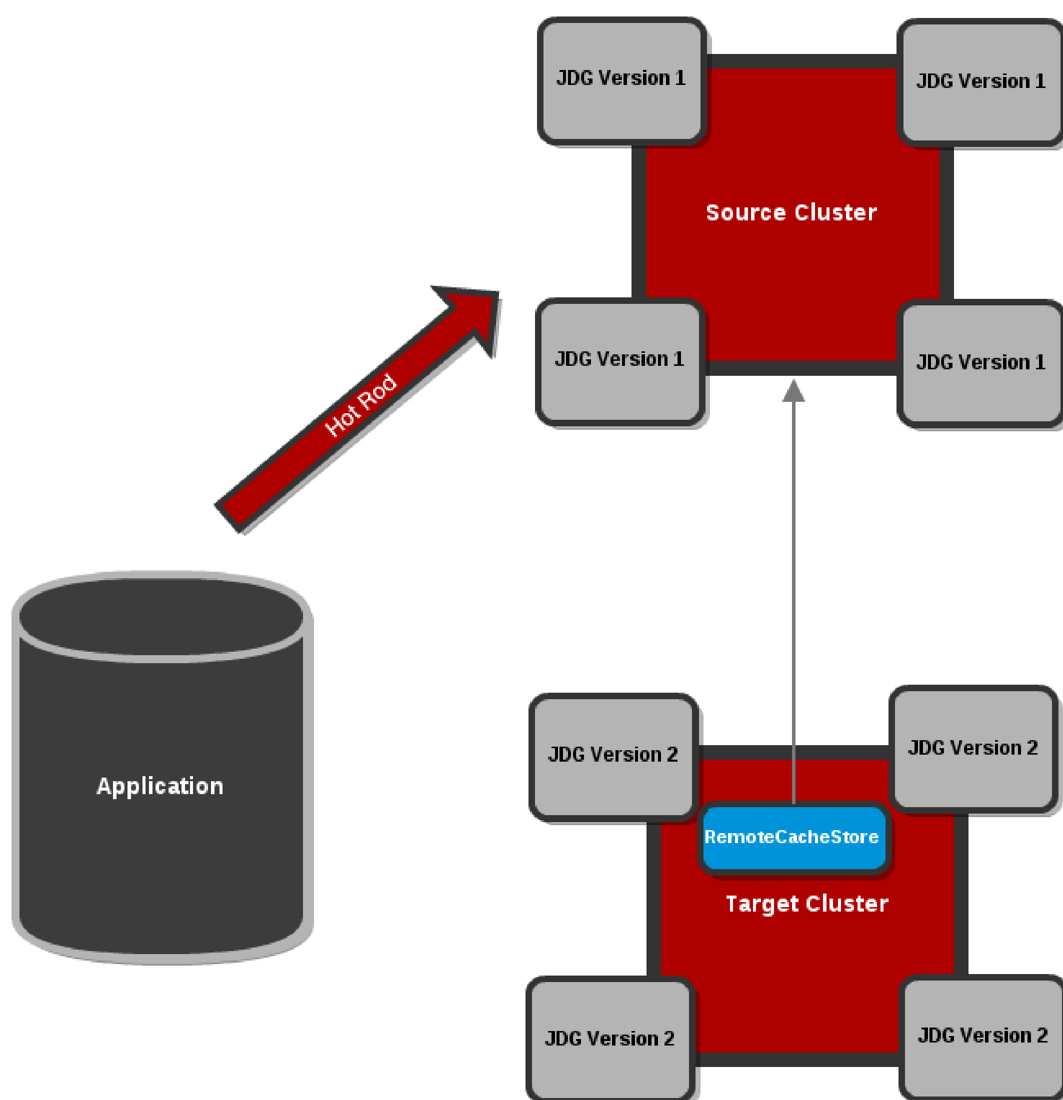
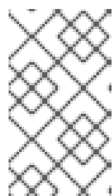


Figure 16.1. Configure the Target Cluster with a RemoteCacheStore



NOTE

See the **$JDG_HOME/server/docs/examples/configs/standalone-hotrod-rolling-upgrade.xml** file for a full example of the Target Cluster configuration for performing Rolling Upgrades.

Target Cluster configuration example assumes the Source Cluster is running with **`standalone.xml`** configuration:

```
<subsystem xmlns="urn:infinispan:server:core:6.0" default-cache-
container="local">
            <cache-container name="local" default-cache="default">
                <local-cache name="default" start="EAGER">
                    <locking isolation="NONE" acquire-
timeout="30000" concurrency-level="1000" striping="false"/>
                    <transaction mode="NONE"/>
                    <remote-store cache="default" socket-
timeout="60000" tcp-no-delay="true" shared="true" hotrod-
wrapping="true" purge="false" passivation="false">
                        <remote-server outbound-socket-
binding="remote-store-hotrod-server"/>
                    </remote-store>
                </local-cache>
            </cache-container>
            <cache-container name="security"/>
        </subsystem>
```

3. **Configure clients to point to the Target Cluster**
   Configure clients to point to the Target Cluster instead of the Source Cluster, restarting each client node one by one.

   Eventually all requests will be handled by the Target Cluster, which will lazily load data from the Source Cluster on demand. The Source Cluster is now the **`RemoteCacheStore`** for the Target Cluster.

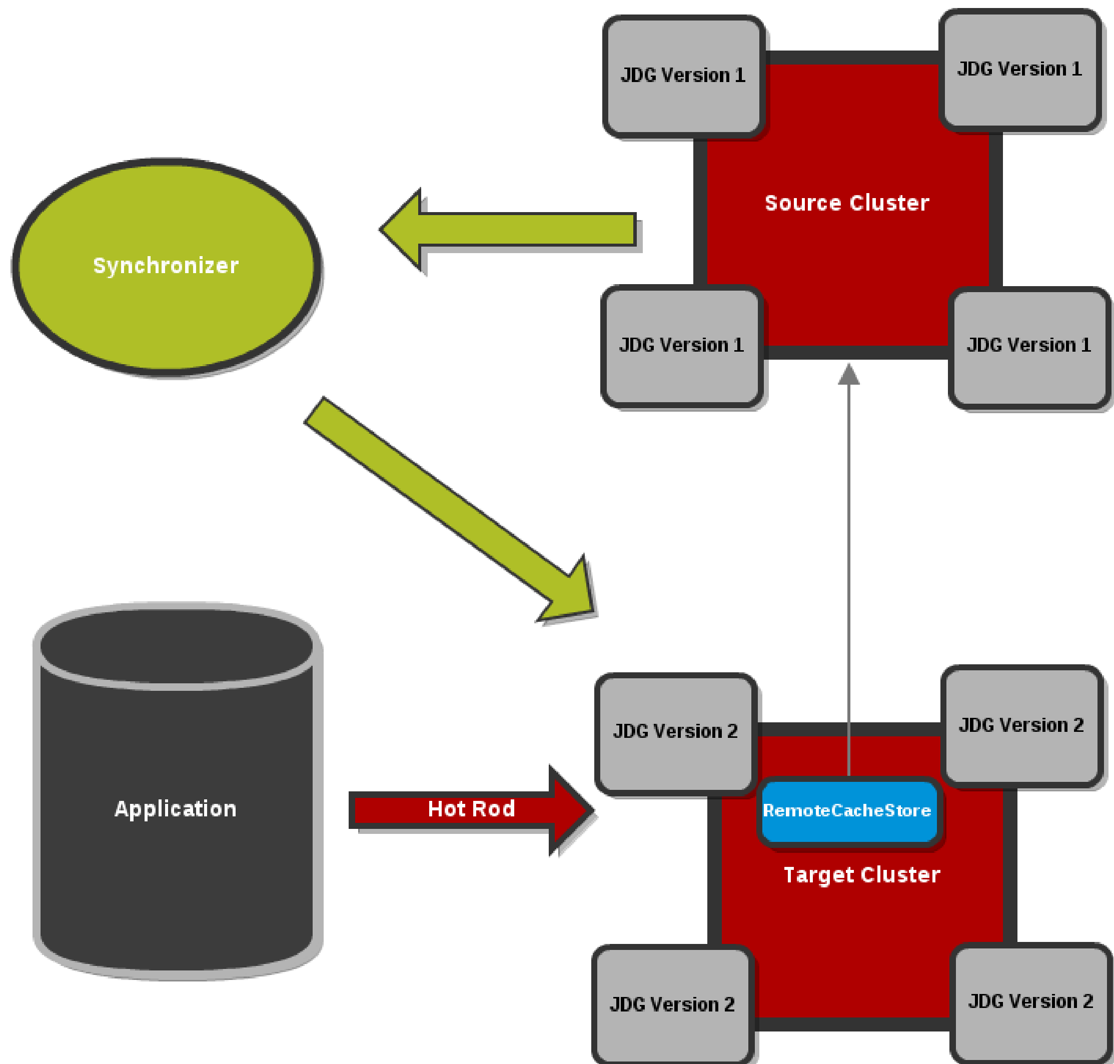


Figure 16.2. Clients point to the Target Cluster with the Source Cluster as `RemoteCacheStore` for the Target Cluster.

4. **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.

5. **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.

6. **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.

7. Decommission the Source Cluster.

Report a bug

16.3. 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

16.4. REMOTECACHESTORE PARAMETERS FOR ROLLING UPGRADES

16.4.1. 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

16.4.2. 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

CHAPTER 17. MAPREDUCE

The Red Hat JBoss Data Grid MapReduce model is an adaptation of Google's **MapReduce** model.

MapReduce is a programming model used to process and generate large data sets. It is typically used in distributed computing environments where nodes are clustered. In JBoss Data Grid, MapReduce allows transparent distributed processing of large amounts of data across the grid. It does this by performing computations locally where the data is stored whenever possible.

MapReduce uses the two distinct computational phases of map and reduce to process information requests through the data grid. The process occurs as follows:

1. The user initiates a task on a cache instance, which runs on a cluster node (the master node).
2. The master node receives the task input, divides the task, and sends tasks for map phase execution on the grid.
3. Each node executes a `Mapper` function on its input, and returns intermediate results back to the master node.
   - If the ***`distributedReducePhase`*** parameter is set to `"true"`, the map results are inserted in an intermediary cache, rather than being returned to the master node.
   - If a `Combiner` has been specified with ***`task.combinedWith(combiner)`***, the `Combiner` is called on the `Mapper` results and the combiner's results are returned to the master node or inserted in the intermediary cache.
4. The master node collects all intermediate results from the map phase and merges all intermediate values associated with the same intermediate key.
   - If the ***`distributedReducePhase`*** parameter is set to `true`, the merging of the intermediate values is done on each node, as the `Mapper` or `Combiner` results are inserted in the intermediary cache.The master node only receives the intermediate keys.
5. The master node sends intermediate key/value pairs for reduction on the grid.
   - If the ***`distributedReducePhase`*** parameter is set to `"false"`, the reduction phase is executed only on the master node.
6. The final results of the reduction phase are returned.
   - If the ***`distributedReducePhase`*** parameter is set to `"true"`, the master node running the task receives all results from the reduction phase and returns the final result to the MapReduce task initiator.
   - If a `Collator` has been specified with ***`task.execute(Collator)`***, the `Collator` is executed on the reduction phase results, and the `Collator` result is returned to the task initiator.

Report a bug

17.1. THE MAPREDUCE API

In Red Hat JBoss Data Grid, each MapReduce task has four main components:

- `Mapper`

- **Reducer**
- **Collator**
- **MapReduceTask**

The **Mapper** class implementation is a component of **MapReduceTask**, which is invoked once per input cache entry key/value pair. **Map** is a process of applying a given function to each element of a list, returning a list of results.

Each node in the JBoss Data Grid executes the **Mapper** on a given cache entry key/value input pair. It then transforms this cache entry key/value pair into an intermediate key/value pair, which is emitted into the provided **Collator** instance.

```
public interface Mapper<KIn, VIn, KOut, VOut> extends Serializable {

   /**
    * Invoked once for each input cache entry KIn,VOut pair.
    */
   void map(KIn key, VIn value, Collector<KOut, VOut> collector);
```

At this stage, for each output key there may be multiple output values. The multiple values must be reduced to a single value, and this is the task of the **Reducer**. JBoss Data Grid's distributed execution environment creates one instance of **Reducer** per execution node.

```
public interface Reducer<KOut, VOut> extends Serializable {

   /**
    * Combines/reduces all intermediate values for a particular
intermediate key to a single value.
    * <p>
    *
    */
   VOut reduce(KOut reducedKey, Iterator<VOut> iter);

}
```

The same **Reducer** interface is used for **Combiners**. A **Combiner** is similar to a **Reducer**, except that it must be able to work on partial results. The **Combiner** is executed on the results of the **Mapper**, on the same node, without considering the other nodes that might have generated values for the same intermediate key.

As **Combiners** only see a part of the intermediate values, they cannot be used in all scenarios, however when used they can reduce network traffic and memory consumption in the intermediate cache significantly.

The **Collator** coordinates results from **Reducers** that have been executed on JBoss Data Grid, and assembles a final result that is delivered to the initiator of the **MapReduceTask**. The **Collator** is applied to the final map key/value result of **MapReduceTask**.

```
public interface Reducer<KOut, VOut> extends Serializable {

   /**
    * Combines/reduces all intermediate values for a particular
intermediate key to a single value.
```

```
     * <p>
     *
     */
    VOut reduce(KOut reducedKey, Iterator<VOut> iter);

}
```

Report a bug

17.1.1. MapReduceTask

In Red Hat JBoss Data Grid, **MapReduceTask** is a distributed task, which unifies the **Mapper**, **Combiner**, **Reducer**, and **Collator** components into a cohesive computation, which can be parallelized and executed across a large-scale cluster.

These components can be specified with a fluent API. However,as most of them are serialized and executed on other nodes, using inner classes is not recommended.

For example:

```
new MapReduceTask(cache)
       .mappedWith(new MyMapper())
       .combinedWith(new MyCombiner())
       .reducedWith(new MyReducer())
       .execute(new MyCollator());
```

MapReduceTask requires a cache containing data that will be used as input for the task. The JBoss Data Grid execution environment will instantiate and migrate instances of provided **Mappers** and **Reducers** seamlessly across the nodes.

By default, all available key/value pairs of a specified cache will be used as input data for the task. This can be modified by using the ***onKeys*** method as an input key filter.

There are two **MapReduceTask** constructor parameters that determine how the intermediate values are processed:

- ***distributedReducePhase*** - When set to **"false"**, the default setting, the reducers are only executed on the master node. If set to **"true"**, the reducers are executed on every node in the cluster.
- ***useIntermediateSharedCache*** - Only important if ***distributedReducePhase*** is set to **"true"**. If **"true"**, which is the default setting, this task will share intermediate value cache with other executing MapReduceTasks on the grid. If set to **"false"**, this task will use its own dedicated cache for intermediate values.

Report a bug

17.1.2. Mapper and CDI

The **Mapper** is invoked with appropriate input key/value pairs on an executing node, however Red Hat JBoss Data Grid also provides a CDI injection for an input cache. The CDI injection can be used where additional data from the input cache is required in order to complete map transformation.

When the **Mapper** is executed on a JBoss Data Grid executing node, the JBoss Data Grid CDI module provides an appropriate cache reference, which is injected to the executing **Mapper**. To use the JBoss Data Grid CDI module with **Mapper**:

1. Declare a cache field in **Mapper**.
2. Annotate the cache field **Mapper** with **@org.infinispan.cdi.Input**.
3. Annotate with mandatory **@Inject annotation**.

For example:

```
public class WordCountCacheInjecterMapper implements Mapper<String,
String, String, Integer> {

   @Inject
   @Input
   private Cache<String, String> cache;

   @Override
   public void map(String key, String value, Collector<String, Integer>
collector) {

      //use injected cache if needed
      StringTokenizer tokens = new StringTokenizer(value);
      while (tokens.hasMoreElements()) {
         for(String token : value.split("\\w")) {
            collector.emit(token, 1);
         }
      }
   }
}
```

Report a bug

17.2. MAPREDUCETASK DISTRIBUTED EXECUTION

Distributed Execution of the MapReduceTask occurs in three phases:

- Mapping phase.
- Outgoing Key and Outgoing Value Migration.
- Reduce phase.

The Mapping phase occurs as follows:

Procedure 17.1. Mapping Phase

1. The input keys are grouped according to their owner nodes.
2. On each node the **Mapper** function processes all key/value pairs local to that node.
3. The results of the mapping process are collected using a **Collector**.

4. If a **Reducer** is specified, it is applied to all intermediate values collected for each outgoing key (***KOut***, ***VOut***).

The Outgoing Key and Outgoing Value migration phase occurs as follows:

Procedure 17.2. Outgoing Key and Outgoing Value Migration Phase

1. Intermediate keys exposed by the **Mapper** are grouped by the intermediate outgoing key (***KOut*** values. This grouping preserves the keys, as the mapping phase, when applied to other nodes in the cluster, may generate identical intermediate keys.

2. Once the Reduce phase has been invoked, (as described in Step 4 of the Mapping Phase above), an underlying hashing mechanism, a temporary distributed cache, and a DeltaAware cache insertion mechanism are used to:

   - hash the intermediate key ***KOut*** using its owner node.

   - migrate the hashed ***KOut*** key and its corresponding ***VOut*** value to the same owner node.

3. The list of ***KOut*** keys are returned to the master node. ***VOut*** values are not returned as they are not required by the master node.

The Reduce phase occurs as follows:

Procedure 17.3. Reduce Phase

1. ***KOut*** keys are grouped according to owner node.

2. The reduce operation applies to each node and its input (grouped ***KOut*** keys) as follows:

   - The reduce operation locates the temporary distributed cache created during the migration phase on the target node.

   - For each ***KOut*** key, a list of ***VOut*** values is taken from the temporary cache.

   - The ***KOut*** and ***VOut*** values are wrapped in an **Iterator** and the reduce operation is applied to the result.

3. The reduce operation generates a map where each key is ***KOut*** and each value is ***VOut***.

   Each node has its own map.

4. The maps from each node in the cluster are combined into a single map (**M**).

5. The MapReduce task returns map (**M**) to the node that initiated the **MapReduce** task.

Report a bug

17.3. MAP REDUCE EXAMPLE

The following example uses a word count application to demonstrate MapReduce and its distributed task abilities.

This example assumes we have a mapping of the key **sentence stored on JBoss Data Grid nodes**.

- Key is a String.
- Each sentence is a String.

All words that appear in all sentences must be counted.

The following defines the implementation of this distributed task:

```
public class WordCountExample {

   /**
    * In this example replace c1 and c2 with
    * real Cache references
    *
    * @param args
    */
   public static void main(String[] args) {
      Cache c1 = null;
      Cache c2 = null;

      c1.put("1", "Hello world here I am");
      c2.put("2", "Infinispan rules the world");
      c1.put("3", "JUDCon is in Boston");
      c2.put("4", "JBoss World is in Boston as well");
      c1.put("12","WildFly");
      c2.put("15", "Hello world");
      c1.put("14", "Infinispan community");
      c2.put("15", "Hello world");

      c1.put("111", "Infinispan open source");
      c2.put("112", "Boston is close to Toronto");
      c1.put("113", "Toronto is a capital of Ontario");
      c2.put("114", "JUDCon is cool");
      c1.put("211", "JBoss World is awesome");
      c2.put("212", "JBoss rules");
      c1.put("213", "JBoss division of RedHat ");
      c2.put("214", "RedHat community");

      MapReduceTask<String, String, String, Integer> t =
         new MapReduceTask<String, String, String, Integer>(c1);
      t.mappedWith(new WordCountMapper())
         .reducedWith(new WordCountReducer());
      Map<String, Integer> wordCountMap = t.execute();
   }

   static class WordCountMapper implements
Mapper<String,String,String,Integer> {
      /** The serialVersionUID */
      private static final long serialVersionUID = -5943370243108735560L;

      @Override
      public void map(String key, String value, Collector<String, Integer>
collector) {
         StringTokenizer tokens = new StringTokenizer(value);
   for(String token : value.split("\\w")) {
            collector.emit(token, 1);
```

```
                }
            }
    }

        static class WordCountReducer implements Reducer<String, Integer> {
            /** The serialVersionUID */
            private static final long serialVersionUID = 1901016598354633256L;

            @Override
            public Integer reduce(String key, Iterator<Integer> iter) {
                int sum = 0;
                while (iter.hasNext()) {
                    Integer i = (Integer) iter.next();
                    sum += i;
                }
                return sum;
            }
        }
    }
```

In this second example, a ***Collator*** is defined, which will transform the result of MapReduceTask Map<KOut,VOut> into a String that is returned to a task invoker. The ***Collator*** is a transformation function applied to a final result of MapReduceTask.

```
MapReduceTask<String, String, String, Integer> t = new
MapReduceTask<String, String, String, Integer>(cache);
t.mappedWith(new WordCountMapper()).reducedWith(new WordCountReducer());
String mostFrequentWord = t.execute(
        new Collator<String,Integer,String>() {

            @Override
            public String collate(Map<String, Integer> reducedResults) {
                String mostFrequent = "";
                int maxCount = 0;
                for (Entry<String, Integer> e : reducedResults.entrySet()) {
                    Integer count = e.getValue();
                    if(count > maxCount) {
                        maxCount = count;
                        mostFrequent = e.getKey();
                    }
                }
            return mostFrequent;
            }

        });
System.out.println("The most frequent word is " + mostFrequentWord);
```

Report a bug

CHAPTER 18. DISTRIBUTED EXECUTION

Red Hat JBoss Data Grid provides distributed execution through a standard JDK **ExecutorService** interface. Tasks submitted for execution are executed on an entire cluster of JBoss Data Grid nodes, rather than being executed in a local JVM.

JBoss Data Grid's distributed task executors can use data from JBoss Data Grid cache nodes as input for execution tasks. As a result, there is no need to configure the cache store for intermediate or final results. As input data in JBoss Data Grid is already load balanced, tasks are also automatically balanced, therefore there is no need to explicitly assign tasks to specific nodes.

In JBoss Data Grid's distributed execution framework:

- Each **DistributedExecutorService** is bound to a single cache. Tasks submitted have access to key/value pairs from that particular cache if the task submitted is an instance of **DistributedCallable**.

- Every **Callable**, **Runnable**, and/or **DistributedCallable** submitted must be either **Serializable** or **Externalizable**, in order to prevent task migration to other nodes each time one of these tasks is performed. The value returned from a **Callable** must also be **Serializable** or **Externalizable**.

Report a bug

18.1. DISTRIBUTEDCALLABLE API

The **DistributedCallable** interface is a subtype of the existing **Callable** from java.util.concurrent.package, and can be executed in a remote JVM and receive input from Red Hat JBoss Data Grid. The **DistributedCallable** interface is used to facilitate tasks that require access to JBoss Data Grid cache data.

When using the **DistributedCallable** API to execute a task, the task's main algorithm remains unchanged, however the input source is changed.

Users who have already implemented **Callable** interface to describe task units must extend **DistributedCallable** and use keys from JBoss Data Grid execution environment as input for the task. For example:

```
public interface DistributedCallable<K, V, T> extends Callable<T> {

   /**
    * Invoked by execution environment after DistributedCallable
    * has been migrated for execution to a specific Infinispan node.
    *
    * @param cache
    *            cache whose keys are used as input data for this
    *            DistributedCallable task
    * @param inputKeys
    *            keys used as input for this DistributedCallable task
    */
   public void setEnvironment(Cache<K, V> cache, Set<K> inputKeys);

}
```

Report a bug

18.2. CALLABLE AND CDI

Where **DistributedCallable** cannot be implemented or is not appropriate, and a reference to input cache used in **DistributedExecutorService** is still required, there is an option to inject the input cache by CDI mechanism.

When the **Callable** task arrives at a Red Hat JBoss Data Grid executing node, JBoss Data Grid's CDI mechanism provides an appropriate cache reference, and injects it to the executing **Callable**.

To use the JBoss Data Grid CDI with **Callable**:

1. Declare a ***Cache*** field in **Callable** and annotated with ***org.infinispan.cdi.Input***

2. Include the mandatory ***@Inject*** annotation.

For example:

```
public class CallableWithInjectedCache implements Callable<Integer>,
Serializable {

      @Inject
      @Input
      private Cache<String, String> cache;


      @Override
      public Integer call() throws Exception {
        //use injected cache reference
        return 1;
      }
}
```

Report a bug

18.3. DISTRIBUTED TASK FAILOVER

Red Hat JBoss Data Grid's distributed execution framework supports task failover in the following cases:

- Failover due to a node failure where a task is executing.

- Failover due to a task failure; for example, if a **Callable** task throws an exception.

The failover policy is disabled by default, and **Runnable**, **Callable**, and **DistributedCallable** tasks fail without invoking any failover mechanism.

JBoss Data Grid provides a random node failover policy, which will attempt to execute a part of a **Distributed** task on another random node if one is available.

A random failover execution policy can be specified using the following as an example:

```
DistributedExecutorService des = new DefaultExecutorService(cache);
DistributedTaskBuilder<Boolean> taskBuilder =
des.createDistributedTaskBuilder(new SomeCallable());
taskBuilder.failoverPolicy(DefaultExecutorService.RANDOM_NODE_FAILOVER);
```

```
DistributedTask<Boolean> distributedTask = taskBuilder.build();
Future<Boolean> future = des.submit(distributedTask);
Boolean r = future.get();
```

The **DistributedTaskFailoverPolicy** interface can also be implemented to provide failover management. For example:

```
/**
 * DistributedTaskFailoverPolicy allows pluggable fail over target
selection for a failed remotely
 * executed distributed task.
 *
 */
public interface DistributedTaskFailoverPolicy {

   /**
    * As parts of distributively executed task can fail due to the task
itself throwing an exception
    * or it can be an Infinispan system caused failure (e.g node failed or
left cluster during task
    * execution etc).
    *
    * @param failoverContext
    *            the FailoverContext of the failed execution
    * @return result the Address of the Infinispan node selected for fail
over execution
    */
   Address failover(FailoverContext context);

   /**
    * Maximum number of fail over attempts permitted by this
DistributedTaskFailoverPolicy
    *
    * @return max number of fail over attempts
    */
   int maxFailoverAttempts();
}
```

Report a bug

18.4. DISTRIBUTED TASK EXECUTION POLICY

The **DistributedTaskExecutionPolicy** allows tasks to specify a custom execution policy across the Red Hat JBoss Data Grid cluster, by scoping execution of tasks to a subset of nodes.

For example, **DistributedTaskExecutionPolicy** can be used to manage task execution in the following cases:

- where a task is to be exclusively executed on a local network site instead of a backup remote network center.
- where only a dedicated subset of a certain JBoss Data Grid rack nodes are required for specific task execution.

The following is an example of the second use case:

```
DistributedExecutorService des = new DefaultExecutorService(cache);
DistributedTaskBuilder<Boolean> taskBuilder =
des.createDistributedTaskBuilder(new SomeCallable());
taskBuilder.executionPolicy(DistributedTaskExecutionPolicy.SAME_RACK);
DistributedTask<Boolean> distributedTask = taskBuilder.build();
Future<Boolean> future = des.submit(distributedTask);
Boolean r = future.get();
```

Report a bug

18.5. DISTRIBUTED EXECUTION EXAMPLE

In this example, parallel distributed execution is used to approximate the value of Pi (π)

1. As shown below, the area of a square is:
   Area of a Square (S) = $4r^2$

2. The following is an equation for the area of a circle:
   Area of a Circle (C) = $\pi \times r^2$

3. Isolate r^2 from the first equation:
   $r^2 = S/4$

4. Inject this value of r^2 into the second equation to find a value for Pi:
   $C = S\pi/4$

5. Isolating π in the equation results in:
   $C = S\pi/4$

   $4C = S\pi$

   $4C/S = \pi$

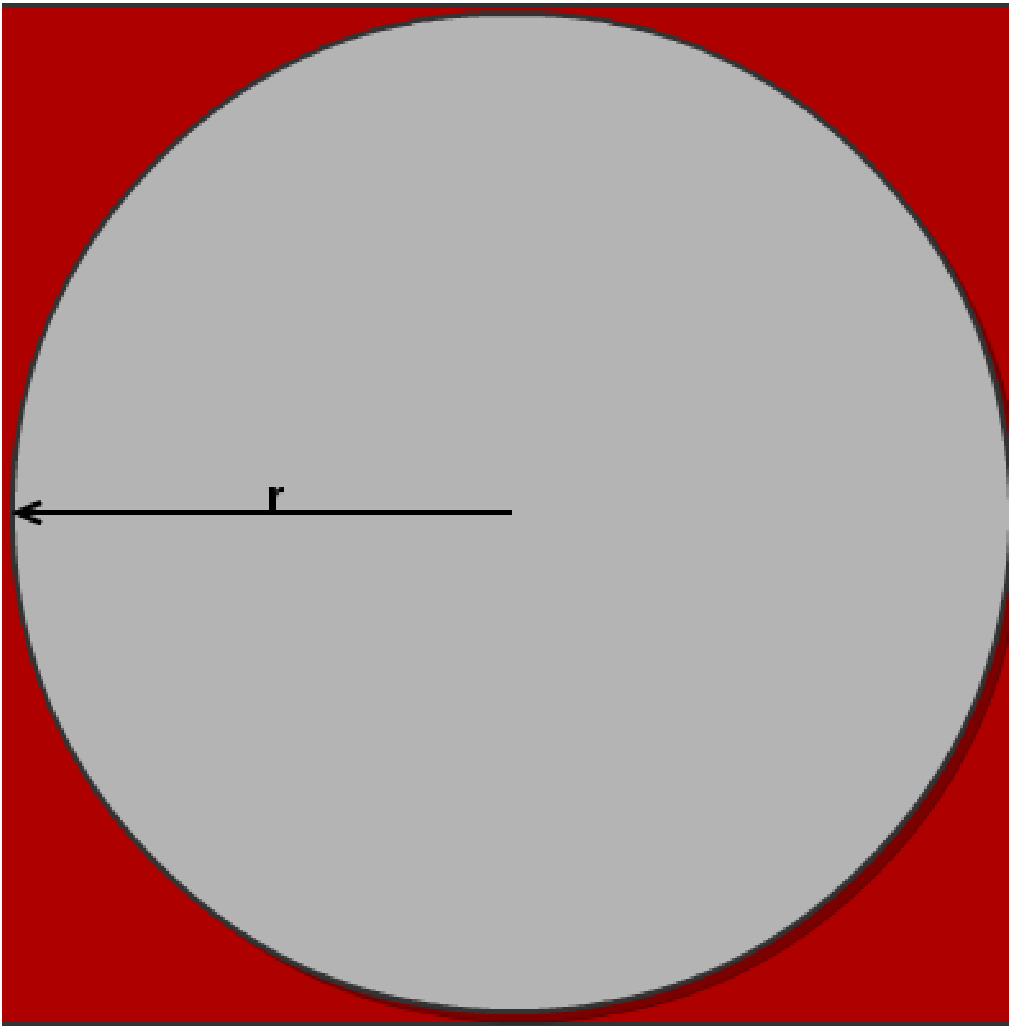


Figure 18.1. Distributed Execution Example

If we now throw a large number of darts into the square, then draw a circle inside the square, and discard all dart throws that landed outside the circle, we can approximate the C/S value.

The value of π is previously worked out to 4C/S. We can use this to derive the approximate value of π. By maximizing the amount of darts thrown, we can derive an improved approximation of π.

In the following example, we throw 10 million darts by parallelizing the dart tossing across the cluster:

```
public class PiAppx {

   public static void main (String [] arg){
      List<Cache> caches = ...;
      Cache cache = ...;

      int numPoints = 10000000;
      int numServers = caches.size();
      int numberPerWorker = numPoints / numServers;

      DistributedExecutorService des = new DefaultExecutorService(cache);
      long start = System.currentTimeMillis();
      CircleTest ct = new CircleTest(numberPerWorker);
      List<Future<Integer>> results = des.submitEverywhere(ct);
      int countCircle = 0;
      for (Future<Integer> f : results) {
         countCircle += f.get();
```

```
        }
        double appxPi = 4.0 * countCircle / numPoints;

        System.out.println("Distributed PI appx is " + appxPi +
        " completed in " + (System.currentTimeMillis() - start) + " ms");
    }

    private static class CircleTest implements Callable<Integer>,
Serializable {

        /** The serialVersionUID */
        private static final long serialVersionUID = 3496135215525904755L;

        private final int loopCount;

        public CircleTest(int loopCount) {
            this.loopCount = loopCount;
        }

        @Override
        public Integer call() throws Exception {
            int insideCircleCount = 0;
            for (int i = 0; i < loopCount; i++) {
                double x = Math.random();
                double y = Math.random();
                if (insideCircle(x, y))
                    insideCircleCount++;
            }
            return insideCircleCount;
        }

        private boolean insideCircle(double x, double y) {
            return (Math.pow(x - 0.5, 2) + Math.pow(y - 0.5, 2))
            <= Math.pow(0.5, 2);
        }
    }
}
```

Report a bug

CHAPTER 19. DATA INTEROPERABILITY

19.1. INTEROPERABILITY BETWEEN LIBRARY AND REMOTE CLIENT-SERVER ENDPOINTS

Red Hat JBoss Data Grid offers multiple ways to interact with data, for example:

- store and retrieve data in a local (embedded) way
- store and retrieve data remotely using various endpoints

In previous versions of JBoss Data Grid, selecting one of these methods ensured that if one method was used to store data, the same method was required to retrieve it. For example, storing data in embedded mode required retrieval via embedded mode instead of a different method, for example using a REST endpoint.

JBoss Data Grid now offers a compatibility mode to allow users to access data with any interface regardless of the method used to store the data. In compatibility mode, the data format is automatically converted for each endpoint, which allows information to be stored and retrieved using different interfaces.

JBoss Data Grid's compatibility mode makes the following assumption about data: the most common use for compatibility mode is to store Java objects. As a result, when enabled, compatibility mode unmarshalls or deserializes data when storing it. This results in increased efficiency when using Java objects with an embedded cache.

Compatibility mode comes at a higher performance cost than non-compatibility mode. As a result, this feature is disabled by default in JBoss Data Grid.

Report a bug

19.2. USING COMPATIBILITY MODE

Red Hat JBoss Data Grid's compatibility mode requires the following to work as expected:

- all endpoints configurations specify the same cache manager
- all endpoints can interact with the same target cache

Using JBoss Data Grid's Remote Client-Server distribution ensures that these requirements are configured by default.

Report a bug

19.3. PROTOCOL INTEROPERABILITY

Red Hat JBoss Data Grid protocol interoperability allows data in the form of raw bytes to be read/write accessed by different protocols, such as REST, Memcached, library, and Hot Rod, that are written in various programming languages, such as C++ or Java.

By default, each protocol stores data in the most efficient format for that protocol, ensuring transformations are not required when retrieving entries. When this data is required to be accessed from multiple protocols, compatibility mode must be enabled on caches that are being shared.

Enable Compatibility Mode declaratively in Client-Server mode

The **compatibility** element's ***enabled*** parameter is set to **true** or **false** to determine whether compatibility mode is in use.

```
<cache-container name="local" default-cache="default" statistics="true">
    <local-cache name="default" start="EAGER" statistics="true">
        <transaction mode="NONE"/>
        <compatibility enabled="true"/>
    </local-cache>
</cache-container>
```

Enable Compatibility Mode programmatically in Library mode

Use a configurationBuilder with the compatibility mode enabled as follows:

```
ConfigurationBuilder builder = ...
builder.compatibility().enable();
```

Enable Compatibility Mode declaratively in Library mode

The **compatibility** element's ***enabled*** parameter is set to **true** or **false** to determine whether compatibility mode is in use.

```
<namedCache name="compatcache">
 <compatibility enabled="true"/>
</namedCache>
```

Report a bug

19.3.1. Use Cases and Requirements

The following table outlines typical use cases for data interoperability in Red Hat JBoss Data Grid.

Table 19.1. Compatibility Mode Use Cases

Use Case	Client A (Reader or Writer)	Client B (Write/Read Counterpart of Client A)
1	Memcached	Hot Rod Java
2	REST	Hot Rod Java
3	Memcached	REST
4	Hot Rod Java	Hot Rod C++
5	Embedded	Hot Rod Java
6	REST	Hot Rod C++
7	Memcached	Hot Rod C++

In the provided use cases, marshalling is entirely up to the user. JBoss Data Grid stores a byte[], while the user marshalls or unmarshalls this into meaningful data.

For example, in Use Case 1, interoperability is between a Memcached client (A), and a Hot Rod Java client (B). If Client A wishes to serialize an application-specific Object, such as a **Person** instance, it would use a String as a key.

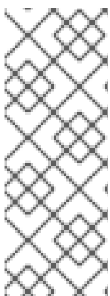
The following steps apply to all use cases:

Client A Side

1. A uses a third-party marshaller, such as Protobuf or Avro, to serialize the **Person** value into a byte[]. A UTF-8 encoded string must be used as the key (according to Memcached protocol requirements).
2. A writes a key-value pair to the server (key as UTF-8 string, the value as byte arrays).

Client B Side

1. B must read a **Person** for a specific key (String).
2. B serializes the same UTF-8 key into the corresponding byte[].
3. B invokes **get(byte[])**
4. B obtains a byte[] representing the serialized object.
5. B uses the same marshaller as A to unmarshall the byte[] into the corresponding **Person** object.



NOTE

- In Use Case 4, the Protostream Marshaller, which is included with the Hot Rod Java client, is recommended. For the Hot Rod C++ client, the Protobuf Marshaller from Google (https://developers.google.com/protocol-buffers/docs/overview) is recommended.
- In Use Case 5, the default Hot Rod marshaller can be used.

Report a bug

19.3.2. Protocol Interoperability Over REST

When data is stored via the REST interface the values are interpreted by embedded, Hot Rod or Memcached clients as a sequence of bytes. Meaning is given to this byte-sequence using the MIME type specified via the "Content-Type" HTTP header, but the content type information is only available to REST clients. No specific interoperability configuration is required for this to occur.

When retrieving data via REST, primitive types stored are read in their primitive format. If a UTF-8 String has been stored via Hot Rod, Embedded, or Memcached, it will be retrieved as String from REST. If custom objects have been serialized and stored via the embedded or remote cache, these can be retrieved as **application/x-java-serialized-object**, **application/xml**, or **application/json**. Any other byte arrays are treated as **application/octet-stream**.

Report a bug

APPENDIX A. REVISION HISTORY

Revision 6.2.1-3	Tue Mar 18 2014	Rakesh Ghatvisave

BZ-1076437: Added Note admonition box.

BZ-1076327: Standardized text in Note admonition boxes.

Revision 6.2.1-2	Wed Mar 12 2014	Rakesh Ghatvisave

BZ-1075586: Made changes as part of the review bug.

BZ-1074839: Added non breaking spaces for product names.

Revision 6.2.1-1	Fri Mar 07 2014	Rakesh Ghatvisave

BZ-1073639: Corrected misspelled word.

BZ-1073638: Corrected misspelled word.

Revision 6.2.1-0	Thu Feb 27 2014	Gemma Sheldon

BZ-1049678: Updated descriptions of Hot Rod operations.

BZ-1068769: Removed CLI tools part. This part is now included in the Administration and Configuration Guide.