Red Hat JBoss Data Grid 6.1

Developer Guide

For use with Red Hat JBoss Data Grid 6.1

Edition 2

Last Updated: 2017-11-21

Red Hat JBoss Data Grid 6.1 Developer Guide

For use with Red Hat JBoss Data Grid 6.1
Edition 2

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

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

Legal Notice

Abstract

An advanced guide intended for developers using Red Hat JBoss Data Grid 6.1

Table of Contents

PREFACE

CHAPTER 1. JBOSS DATA GRID

1.1. ABOUT JBOSS DATA GRID

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

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

JBoss Data Grid 6.1 and JBoss Data Grid 6.1 Beta is based on Infinispan version 5.2.

Report a bug

1.2. JBOSS DATA GRID SUPPORTED CONFIGURATIONS

The set of supported features, configurations, and integrations for JBoss Data Grid (6.0 and 6.1) are available at the Supported Configurations page at https://access.redhat.com/knowledge/articles/115883.

Report a bug

1.3. JBOSS DATA GRID USAGE MODES

1.3.1. JBoss Data Grid Usage Modes

JBoss Data Grid offers two usage modes:

- Remote Client-Server mode
- Library mode

Report a bug

1.3.2. Remote Client-Server Mode

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

All JBoss Data Grid operations in Remote Client-Server mode are non-transactional. As a result, a number of features cannot be performed when running JBoss Data Grid in Remote Client-Server mode.

However, there are a number of benefits to running JBoss Data Grid in Remote Client-Server mode if you do not require any features that require Library mode. Remote Client-Server mode is client language agnostic, provided there is a client library for your chosen protocol. As a result, Remote

Client-Server mode provides:

- easier scaling of the data grid.
- easier upgrades of the data grid without impact on client applications.

Report a bug

1.3.3. Library Mode

Library mode allows the user to build and deploy a custom runtime environment. The Library usage mode hosts a single data grid node in the applications process, with remote access to nodes hosted in other JVMs. Tested containers for JBoss Data Grid 6 Library mode includes Tomcat 7 and JBoss Enterprise Application Platform 6. Library mode is supported in JBoss Data Grid 6.0.1.

A number of features in JBoss Data Grid can be used in Library mode, but not Remote Client-Server mode.

Use Library mode if you require:

- transactions.
- listeners and notifications.

Report a bug

1.4. JBOSS DATA GRID BENEFITS

JBoss Data Grid provides the following benefits:

Benefits of JBoss Data Grid

Performance

Accessing objects from local memory is faster than accessing objects from remote data stores (such as a database). JBoss Data Grid provides an efficient way to store in-memory objects coming from a slower data source, resulting in faster performance than a remote data store. JBoss Data Grid also offers optimization for both clustered and non clustered caches to further improve performance.

Consistency

Storing data in a cache carried the inherent risk: at the time it is accessed, the data may be outdated (stale). To address this risk, JBoss Data Grid uses mechanisms such as cache invalidation and expiration to remove stale data entries from the cache. Additionally, JBoss Data Grid supports JTA, distributed (XA) and two-phase commit transactions along with transaction recovery and a version API to remove or replace data according to saved versions.

Massive Heap and High Availability

In JBoss Data Grid, applications no longer need to delegate the majority of their data lookup processes to a large single server database for performance benefits. JBoss Data Grid employs techniques such as replication and distribution to completely remove the bottleneck that exists in the majority of current enterprise applications.

> **Example 1.1. Massive Heap and High Availability Example**

> In a sample grid with 16 blade servers, each node has 2 GB storage space dedicated for a replicated cache. In this case, all the data in the grid is copies of the 2 GB data. In contrast, using a distributed grid (assuming the requirement of one copy per data item, resulting in the capacity of the overall heap being divided by two) the resulting memory backed virtual heap contains 16 GB data. This data can now be effectively accessed from anywhere in the grid. In case of a server failure, the grid promptly creates new copies of the lost data and places them on operational servers in the grid.

Scalability

A significant benefit of a distributed data grid over a replicated clustered cache is that a data grid is scalable in terms of both capacity and performance. Add a node to JBoss Data Grid to increase throughput and capacity for the entire grid. JBoss Data Grid uses a consistent hashing algorithm that limits the impact of adding or removing a node to a subset of the nodes instead of every node in the grid.

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

Data Distribution

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

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

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

Persistence

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

Language bindings

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

Management

In a grid environment of several hundred or more servers, management is an important feature.

JBoss Operations Network, the enterprise network management software, is the best tool to manage multiple JBoss Data Grid instances. JBoss Operations Network's features allow easy and effective monitoring of the Cache Manager and cache instances.

Remote Data Grids

Rather than scale up the entire application server architecture to scale up your data grid, JBoss Data Grid provides a Remote Client-Server mode which allows the data grid infrastructure to be upgraded independently from the application server architecture. Additionally, the data grid server can be assigned different resources than the application server and also allow independent data grid upgrades and application redeployment within the data grid.

Report a bug

1.5. JBOSS DATA GRID PREREQUISITES

The only prerequisites to set up JBoss Data Grid is a Java Virtual Machine (compatible with Java 6.0 or better) and that the most recent supported version of the product is installed on your system.

Report a bug

1.6. JBOSS DATA GRID VERSION INFORMATION

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

Report a bug

1.7. JBOSS DATA GRID CACHE ARCHITECTURE

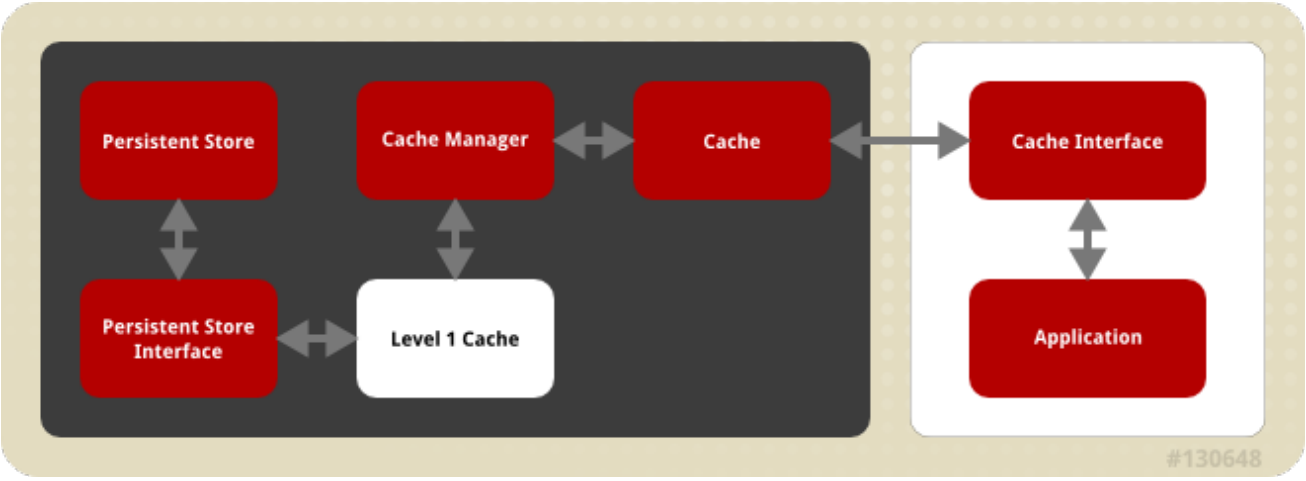


Figure 1.1. JBoss Data Grid Cache Architecture

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

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

- Elements that a user can interact directly with (depicted within a white box), which includes Cache Interfaces and the Application.

Cache Architecture Elements

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

1. The Persistent Store permanently stores cache instances and entries.
2. JBoss Data Grid offers two Persistent Store Interfaces to access the persistent store. Persistent store interfaces can be either:
    - A cache loader is a read only interface that provides a connection to a persistent data store. A cache loader can locate and retrieve data from cache instances and from the persistent store.
    - A cache store extends the cache loader functionality to include write capabilities by exposing methods that allow the cache loader to load and store states.
3. The Level 1 Cache (or L1 Cache) stores remote cache entries after they are initially accessed, preventing unnecessary remote fetch operations for each subsequent use of the same entries.
4. The Cache Manager is the primary mechanism used to retrieve a Cache instance in JBoss Data Grid, and can be used as a starting point for using the Cache.
5. The Cache stores cache instances retrieved by a Cache Manager.
6. Cache Interfaces use protocols such as Memcached and Hot Rod, or REST to interface with the cache. For details about the remote interfaces, refer to the *Developer Guide*.
    - Memcached is a distributed memory object caching system used to store key-values in-memory. The Memcached caching system defines a text based, client-server caching protocol called the Memcached protocol.
    - Hot Rod is a binary TCP client-server protocol used in JBoss Data Grid. It was created to overcome deficiencies in other client/server protocols, such as Memcached. Hot Rod enables clients to do smart routing of requests in partitioned or distributed JBoss Data Grid server clusters.
    - The REST protocol eliminates the need for tightly coupled client libraries and bindings. The REST API introduces an overhead, and requires a REST client or custom code to understand and create REST calls.
7. An application allows the user to interact with the cache via a cache interface. Browsers are a common example of such end-user applications.

Report a bug

1.8. JBOSS DATA GRID APIS

JBoss Data Grid provides the following programmable APIs:

- Cache
- Batching
- Grouping

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

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

PART I. PROGRAMMABLE APIS

CHAPTER 2. THE CACHE API

2.1. ABOUT THE CACHE API

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

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

NOTE

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

Report a bug

2.2. USING THE CONFIGURATIONBUILDER API TO CONFIGURE THE CACHE API

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.

Report a bug

2.3. PER-INVOCATION FLAGS

2.3.1. About Per-Invocation Flags

Per-invocation flags can be used with data grids in JBoss Data Grid 6 to specify behavior for each cache call. Per-invocation flags facilitate the implementation of potentially time saving optimizations.

Report a bug

2.3.2. Per-Invocation Flag Functions

The **putForExternalRead()** method in 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 should retry the operation.

Report a bug

2.3.3. Configure Per-Invocation Flags

To use per-invocation flags in 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

2.3.4. Per-Invocation Flags Example

In a use case for JBoss Data Grid, where a write operation, such as **`put()`**, should not return the previous value, two flags are used. The two flags prevent 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, the two flags prevent the previous value from being loaded from its cache store.

An example of using the two flags is:

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

Report a bug

2.4. THE ADVANCEDCACHE INTERFACE

2.4.1. About the AdvancedCache Interface

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

2.4.2. Flag Usage with the AdvancedCache Interface

Flags, when applied to certain cache methods in 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

2.4.3. 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

2.4.4. Custom Interceptors

2.4.4.1. About Custom Interceptors

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

Report a bug

2.4.4.2. Custom Interceptor Design

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

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

Report a bug

2.4.4.3. Add Custom Interceptors

2.4.4.3.1. Adding Custom Interceptors Declaratively

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

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

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

NOTE

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

Report a bug

2.4.4.3.2. Adding Custom Interceptors Programmatically

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

For example:

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

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

For example:

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

Report a bug

CHAPTER 3. THE BATCHING API

3.1. ABOUT THE BATCHING API

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

NOTE

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

Report a bug

3.2. ABOUT JAVA TRANSACTION API TRANSACTIONS

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

JBoss Data Grid does the following for each cache operation:

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

IMPORTANT

With JBoss Data Grid 6.0.x, it is recommended to disable transactions in Remote Client-Server Mode. However, if an error displays warning of an **`ExceptionTimeout`** where JBoss Data Grid is **`Unable to acquire lock after {time} on key {key} for requester {thread}`**, enable transactions. This occurs because non-transactional caches acquire locks on each node they write on. Using transactions prevents deadlocks because caches acquire locks on a single node. This problem is resolved in JBoss Data Grid 6.1.

Report a bug

3.3. BATCHING AND THE JAVA TRANSACTION API (JTA)

In 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 supported in Library mode only.

Report a bug

3.4. USING THE BATCHING API

3.4.1. Enable the Batching API

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">
...
</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

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

Report a bug

3.4.3. 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 3.1. Without Using Batch

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

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

Example 3.2. 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

3.4.4. 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 3.3. Batching API Usage Example

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 4. THE GROUPING API

4.1. ABOUT THE GROUPING API

When using the Grouping API, JBoss Data Grid creates and uses the hash of the group instead of the hash of the key to determine which node will house the entry.

With the Grouping API, it is still important that each node can still use an algorithm to determine the owner of each key. For this purpose, the group cannot be manually specified and must be either intrinsic to the entry (generated by the key class) or extrinsic to the entry (generated by an external function).

Report a bug

4.2. GROUPING API OPERATIONS

JBoss Data Grid normally uses the hash of a specific key to determine a destination node to store an entry. When using the Grouping API, JBoss Data Grid uses a hash of the group instead of the hash of the key to determine the destination node. However, the hash of the key is still used when actually storing the entry on a node.

It is important that each node is able to use an algorithm to determine the owner of each key, rather than pass metadata about the location of entries between nodes. As a result of this requirement, 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

4.3. GROUPING API CONFIGURATION

In order to use the Grouping API in JBoss Data Grid, first enable groups.

Programmatic configuration

To configure JBoss Data Grid using the programmatic API, call the following:

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

XML configuration

To configure JBoss Data Grid using XML, use the following:

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

If the class definition of the key is alterable, and the group's determination is not an orthogonal concern to the key class, use an intrinsic group. Use the **@Group** annotation within the method to specify the intrinsic group. For example:

```
class User {

   ...
   String office;
   ...

   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;
   }

}
```

NOTE

The group must be a **String**.

Without key class control, or in a case where the determination of the group is an orthogonal concern to the key class, use an extrinsic group. Specify an extrinsic group by implementing the **Grouper** interface. The **computeGroup** method within the **Grouper** interface returns the group.

Grouper operates as an interceptor and passes previously computed values to the **computeGroup()** method. If defined, **@Group** determines which group is passed to the first **Grouper**, providing improved group control when using intrinsic groups.

To use a **grouper** to determine a key's group, its ***keyType*** must be assignable from the target key.

The following is an example of a **Grouper**:

```
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)(<a>\\d</a>)$");

    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;
    }

}
```

In this example, a simple **grouper** uses the key class to extract the group from a key using a pattern. Information specified on the key class is ignored. Each **grouper** must be registered to be used.

Programmatic configuration

When configuring JBoss Data Grid programmatically:

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

XML Configuration

Or when configuring JBoss Data Grid using XML:

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

Report a bug

CHAPTER 5. THE CACHESTORE AND CONFIGURATIONBUILDER APIS

5.1. ABOUT THE CACHESTORE API

An implementation of the CacheStore interface defines the cache's read-through and write-through behavior. A cache store method is called to perform operations in the secondary storage such as the addition or removal of entries.

Report a bug

5.2. THE CONFIGURATIONBUILDER API

5.2.1. About the ConfigurationBuilder API

The ConfigurationBuilder API is a programmatic configuration API in 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

5.2.2. Using the ConfigurationBuilder API

5.2.2.1. Programmatically Create a CacheManager and Replicated Cache

Programmatic configuration in JBoss Data Grid almost exclusively involves the ConfigurationBuilder API and the CacheManager.

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.

Report a bug

5.2.2.2. 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.

To customize the default cache to fit the above example requirements, use the following steps:

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)
.build();
```

3. Register/define your cache configuration with a cache manager:

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

Report a bug

5.2.2.3. 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.

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)
.build();
```

3. Register/define your cache configuration with a cache manager:

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

Report a bug

5.2.2.4. 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

5.2.2.5. Global Configuration Examples

5.2.2.5.1. Globally Configure the Transport Layer

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

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

Report a bug

5.2.2.5.2. 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")
    .mBeanServerLookupClass(JBossMBeanServerLookup.class)
  .build();
```

Report a bug

5.2.2.5.3. Globally Customize Thread Pool Executors

Some 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(DefaultScheduledExecutorFactory.class)
  .addProperty("threadNamePrefix", "RQThread")
.build();
```

Report a bug

5.2.2.6. Cache Level Configuration Examples

5.2.2.6.1. Cache Level Configuration for the Cluster Mode

The following configuration allows you to use 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)
    .hash().numOwners(3)
  .build();
```

Report a bug

5.2.2.6.2. Cache Level Eviction and Expiration Configuration

The following configuration allows you 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

5.2.2.6.3. Cache Level Configuration for JTA Transactions

To interact with a cache for JTA transaction configuration, you must 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 you enable JMX management and statistics gathering as well.

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

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

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

Report a bug

5.2.2.6.4. 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()
      .loaders()
```

```
        .shared(false).passivation(false).preload(false)

.addFileCacheStore().location("/tmp").streamBufferSize(1800).async().enabl
e().threadPoolSize(20).build();
```

Report a bug

5.2.2.6.5. 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(PersonExternalizer.class)
    .addAdvancedExternalizer(999, AddressExternalizer.class)
  .build();
```

Report a bug

5.2.2.6.6. Cache Level Configuration for Custom Interceptors

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

```
  Configuration config = new ConfigurationBuilder()
  .customInterceptors().interceptors()
    .add(new FirstInterceptor()).first()
    .add(new LastInterceptor()).last()
    .add(new FixPositionInterceptor()).atIndex(8)
    .add(new AfterInterceptor()).after(LockingInterceptor.class)
    .add(new BeforeInterceptor()).before(CallInterceptor.class)
  .build();
```

Report a bug

CHAPTER 6. THE EXTERNALIZABLE API

6.1. ABOUT EXTERNALIZER

An `Externalizer` is a class that can:

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

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

Report a bug

6.2. ABOUT THE EXTERNALIZABLE API

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

Report a bug

6.3. USING THE EXTERNALIZABLE API

6.3.1. The Externalizable API Usage

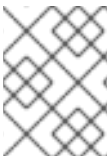
JBoss Data Grid uses JBoss Marshalling to serialize objects.

Serialized classes require a corresponding Externalizer interface implementation that is configured to:

- Transform an object class into a serialized class
- Read an object class from an output.

Use a separate class to implement the Externalizer interface. Externalizer implementations control how objects of a class are created when reading an object from a stream.

The `readObject()` implementations create object instances of the target class. This provides flexibility in the creation of instances and allows target classes to persist immutably.



NOTE

It is recommended that Externalizer implementations are stored within classes that they externalize as inner static public classes.

Report a bug

6.3.2. The Externalizable API Configuration Example

To configure JBoss Data Grid's Externalizable API:

- Provide an ***externalizer*** implementation for the type of object to be marshalled/unmarshalled.
- Annotate the marshalled type class using {@link SerializeWith} to indicate the ***externalizer*** class.

For example:

```
import org.infinispan.marshall.Externalizer;
import org.infinispan.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());
      }
   }
}
```

There are several disadvantages to configuring Externalizers in this manner:

- The payload size generated using this method can be inefficient due to constraints within the model.
- An Externalizer can be required for a class for which the source code is not available, or the source code cannot be modified.
- The use of annotations can limit framework developers or service providers attempting to abstract lower level details, such as marshalling layer.

Report a bug

6.3.3. Linking Externalizers with Marshaller Classes

The Externalizer's **`readObject()`** and **`writeObject()`** methods link with the type classes they are configured to externalize by providing a **`getTypeClasses()`** implementation.

For example:

```
import org.infinispan.util.Util;
...
@Override
public Set<Class<? extends ReplicableCommand>> getTypeClasses() {
  return Util.asSet(LockControlCommand.class, RehashControlCommand.class,
      StateTransferControlCommand.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 example, ReplicableCommandExternalizer indicates that it can externalize multiple commands.

There can be instances where the class instance to be externalized cannot be referenced because the source code is not available or cannot be modified. In this case, users can attempt to look up the class using the fully qualified class name. For example:

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

Report a bug

6.4. THE ADVANCEDEXTERNALIZER

6.4.1. About the AdvancedExternalizer

JBoss Data Grid's AdvancedExternalizer provides externalizers for marshalling/unmarshalling user-defined classes.

The AdvancedExternalizer overcomes the deficiencies identified when using the basic Externalizable API configuration as the AdvancedExternalizer does not require user classes to be annotated.

Report a bug

6.4.2. AdvancedExternalizer Example Configuration

Use the AdvancedExternalizer using the following configuration:

```
  import org.infinispan.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;
      }
   }
}
```

Report a bug

6.4.3. Externalizer Identifiers

AdvancedExternalizers in JBoss Data Grid require Externalizer implementations to provide an identifier using one of the following:

- **`getId()`** implementations.
- Declarative or Programmatic configuration that identifies the externalizer when unmarshalling a payload.

When registering Externalizers using declarative or programmatic configuration, registration must occur when cache managers are created.

`GetId()` will either return a positive integer or a null value:

- A positive integer allows the externalizer to be identified when read and assigned to the correct Externalizer capable of reading the contents.
- A null value indicates that the identifier of the AdvancedExternalizer will be defined via declarative or programmatic configuration.

Any positive integer can be used, provided it is not used by any other identifier in the system.

JBoss Data Grid will check for identifier duplicates on start up, and halts the start up process if duplicates are found.

Report a bug

6.4.4. Registering Advanced Externalizers

In JBoss Data Grid, AdvancedExternalizer implementation can be registered using XML or Programmatic configuration, or via annotation.

An Advanced Externalizer implementation for Person object stored as a static inner class can be configured programmatically or declaratively.

Declarative Configuration:

The following is an example of a declarative configuration for an advanced Externalizer implementation:

```
<infinispan>
  <global>
    <serialization>
      <advancedExternalizers>
        <advancedExternalizer
externalizerClass="Person$PersonExternalizer"/>
      </advancedExternalizers>
    </serialization>
  </global>
  ...
</infinispan>
```

Programmatic Configuration:

The following is an example of a programmatic configuration for an advanced Externalizer implementation:

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

An AdvancedExternalizer implementation must be associated with an identifier, regardless of the configuration method used.

If an identifier in an AdvancedExternalizer implementation is defined using both XML/Programmatic configuration as well as annotation, XML/Programmatic defined value will be used.

The following example shows an Externalizer where the identifier is defined at the time of registration:

Declarative Configuration:

The following is a declarative configuration for the location of the identifier definition during registration:

```
<infinispan>
  <global>
    <serialization>
      <advancedExternalizers>
        <advancedExternalizer id="123"

externalizerClass="Person$PersonExternalizer"/>
      </advancedExternalizers>
    </serialization>
  </global>
  ...
</infinispan>
```

Programmatic Configuration:

The following is a programmatic configuration for the location of the identifier definition during registration:

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
   .addAdvancedExternalizer(123, new Person.PersonExternalizer());
```

Report a bug

6.4.5. Register Multiple Externalizers Programmatically

Multiple externalizers can be registered at the same time using JBoss Data Grid's programmatic configuration. This feature requires that the relevant identifiers have already been defined using the `@Marshalls` annotation.

The following is an example of registering multiple externalizers:

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

Report a bug

6.5. INTERNAL EXTERNALIZER IMPLEMENTATION ACCESS

6.5.1. Internal Externalizer Implementation Access

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

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

```
      ma.writeObject(output, object.getMap());
   }

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

   ...
```

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

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

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

   ...
}
```

Report a bug

CHAPTER 7. THE NOTIFICATION/LISTENER API

7.1. ABOUT THE LISTENER API

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

Report a bug

7.2. LISTENER EXAMPLE

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

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

Report a bug

7.3. CACHE ENTRY MODIFIED LISTENER CONFIGURATION

In a cache entry modified listener, the modified value cannot be retrieved via **`Cache.get()`** when ***`isPre`*** (an Event method) is **`false`**. For more information about ***`isPre()`***, refer to the JBoss Data Grid *API Documentation*'s listing for the **`org.infinispan.notifications.cachelistener.event`** package.

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

Report a bug

7.4. NOTIFICATIONS

7.4.1. About Listener Notifications

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

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

Report a bug

7.4.2. About Cache-level Notifications

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

Report a bug

7.4.3. Cache Manager-level Notifications

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

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

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

Report a bug

7.4.4. About Synchronous and Asynchronous Notifications

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

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

Annotate listeners using the following:

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

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

Report a bug

7.5. NOTIFYING FUTURES

7.5.1. About NotifyingFutures

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

NOTE

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

Report a bug

7.5.2. NotifyingFutures Example

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

```
FutureListener futureListener = new FutureListener() {

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

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

Report a bug

PART II. REMOTE CLIENT-SERVER MODE INTERFACES

CHAPTER 8. THE ASYNCHRONOUS API

8.1. ABOUT THE ASYNCHRONOUS API

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

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

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

Report a bug

8.2. 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

8.3. ABOUT ASYNCHRONOUS PROCESSES

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

- Network calls
- Marshalling

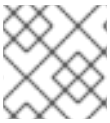
- Writing to a cache store (optional)
- Locking

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

Report a bug

8.4. RETURN VALUES AND THE ASYNCHRONOUS API

When the asynchronous API is used in 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 available in JBoss Data Grid's library mode only.

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

9.1. ABOUT THE REST INTERFACE IN JBOSS DATA GRID

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

9.2. RUBY CLIENT CODE

The following code is an example of interacting with 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

9.3. USING JSON WITH RUBY EXAMPLE

Prerequisites

To use JavaScript Object Notation (JSON) with ruby to interact with JBoss Data Grid's REST Interface, install the JSON Ruby library (refer to 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

9.4. PYTHON CLIENT CODE

The following code is an example of interacting with the 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

9.5. JAVA CLIENT CODE

The following code is an example of interacting with 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&amp;nbsp; 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

9.6. CONFIGURE THE REST INTERFACE

9.6.1. About JBoss Data Grid Connectors

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

9.6.2. Configure REST Connectors

The following are the configuration elements for the **`rest-connector`** element in JBoss Data Grid's Remote Client-Server mode.

```
<subsystem xmlns="urn:jboss:domain:datagrid:1.0">
   <rest-connector virtual-server="default-host"
                   cache-container="default"
                   context-path="$CONTEXT_PATH"
                   security-domain="other"
                   auth-method="BASIC"
                   security-mode="WRITE" />
</subsystem>
```

Report a bug

9.6.3. REST Connector Attributes

The following is a list of attributes used to configure the REST connector in JBoss Data Grid's Remote Client-Server Mode.

- The **`rest-connector`** element specifies the configuration information for the REST connector.
  - 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.
  - The ***`cache-container`*** parameter names the cache container used by the REST connector. This is a mandatory 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.

- 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.
- 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.
- 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.

Report a bug

9.7. USING THE REST INTERFACE

9.7.1. REST Interface Operations

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

- Adding data.
- Retrieving data.
- Removing data.

Report a bug

9.7.2. Adding Data

9.7.2.1. Adding Data Using the REST Interface

In 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

9.7.2.2. 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

9.7.2.3. 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

9.7.3. Retrieving Data

9.7.3.1. Retrieving Data Using the REST Interface

In 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

9.7.3.2. 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 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

9.7.3.3. 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

9.7.4. Removing Data

9.7.4.1. Removing Data Using the REST Interface

To remove data from 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 a cache. (**DELETE /{cacheName}**)

Report a bug

9.7.4.2. 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

9.7.4.3. 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

9.7.4.4. 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

9.7.5. REST Interface Operation Headers

9.7.5.1. Headers

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

Table 9.1. Header Types

Headers	Mandatory/Optional	Values	Default Value	Details

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.
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.
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.

Header	Algorithm	Example	Details
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

9.8. REST INTERFACE SECURITY

NOTE

Note that the JBoss Data Grid 6.1 REST endpoint is set to public as a default.

Report a bug

9.8.1. Enable Security for the REST Endpoint

Prerequisite

JBoss Data Grid includes an example **`standalone-rest-auth.xml`** file located within the JBoss Data Grid directory at the location **`/docs/examples/configs`**).

Copy the file to the **$JDG_HOME/standalone/configuration** directory to use the configuration. From the **$JDG_HOME** location, enter the following command to create a copy of the **standalone-rest-auth.xml** in the appropriate location:

```
$ cp docs/examples/configs/standalone-rest-auth.xml
standalone/configuration/standalone.xml
```

If required, create a new copy of the example **standalone-rest-auth.xml** to start with a new configuration template.

Procedure 9.1. Enable Security for the REST Endpoint

To enable security for the 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:jboss:domain:datagrid:1.0">
              <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, refer to the JBoss Application Server 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:

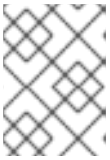
```
user1=2dc3eacfed8cf95a4a31159167b936fc
```

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

10.1. ABOUT THE MEMCACHED PROTOCOL

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.

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

10.2. ABOUT MEMCACHED SERVERS IN JBOSS DATA GRID

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

10.3. USING THE MEMCACHED INTERFACE

10.3.1. Memcached Statistics

The following table contains a list of valid statistics available using the memcached protocol in 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.

Statistic	Data Type	Details
total_items	32-bit unsigned integer.	Contains the total number of items stored by the instance during its lifetime.
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.

Statistic	Data Type	Details
cas_hits	64-bit unsigned integer	Contains the number of keys to be compared and swapped that were found and successfully compared and swapped.
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_badvalue	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

10.4. CONFIGURE THE MEMCACHED INTERFACE

10.4.1. About JBoss Data Grid Connectors

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

10.4.2. Configure Memcached Connectors

The following are the configuration elements for the `memcached-connector` element in JBoss Data Grid's Remote Client-Server Mode.

```
<subsystem xmlns="urn:jboss:domain:datagrid:1.0">
<memcached-connector socket-binding="memcached"
                     cache-container="default"
                     worker-threads="4"
                     idle-timeout="-1"
                     tcp-nodelay="true"
                     send-buffer-size="0"
                     receive-buffer-size="0" />
</subsystem>
```

Report a bug

10.4.3. Memcached Connector Attributes

- The following is a list of attributes used to configure the memcached connector within the **connectors** element in JBoss Data Grid's Remote Client-Server Mode.
  - The **memcached-connector** element defines the configuration elements for use with memcached.
    - The ***socket-binding*** parameter specifies the socket binding port used by the memcached connector. This is a mandatory parameter.
    - The ***cache-container*** parameter names the cache container used by the memcached connector. This is a mandatory parameter.
    - The ***worker-threads*** parameter specifies the number of worker threads available for the memcached connector. The default value for this parameter is the number of cores available multiplied by two. This is an optional 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.
    - 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.
    - 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.
    - 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.

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 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.

Report a bug

11.2. THE BENEFITS OF USING HOT ROD OVER MEMCACHED

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. ABOUT HOT ROD SERVERS IN JBOSS DATA GRID

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

11.4. HOT ROD HASH FUNCTIONS

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.

Report a bug

11.5. HOT ROD SERVER NODES

11.5.1. 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

11.5.2. The hotrod.properties File

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

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

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

Properties that can be included in hotrod.properties are:

`infinispan.client.hotrod.request_balancing_strategy`

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

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

`infinispan.client.hotrod.server_list`

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

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

`infinispan.client.hotrod.force_return_values`

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

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

`infinispan.client.hotrod.tcp_no_delay`

Affects TCP NODELAY on the TCP stack.

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

`infinispan.client.hotrod.ping_on_startup`

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

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

`infinispan.client.hotrod.transport_factory`

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

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

`infinispan.client.hotrod.marshaller`

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

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

`infinispan.client.hotrod.async_executor_factory`

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

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

`infinispan.client.hotrod.default_executor_factory.pool_size`

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

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

`infinispan.client.hotrod.default_executor_factory.queue_size`

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

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

infinispan.client.hotrod.hash_function_impl.1

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

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

infinispan.client.hotrod.key_size_estimate

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

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

infinispan.client.hotrod.value_size_estimate

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

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

infinispan.client.hotrod.socket_timeout

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

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

infinispan.client.hotrod.protocol_version

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

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

infinispan.client.hotrod.connect_timeout

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

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

See Also:

- Section 12.1, “About the RemoteCache Interface”

Report a bug

11.6. HOT ROD HEADERS

11.6.1. Hot Rod Header Data Types

All keys and values used for Hot Rod in 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 11.1. Header Data Types

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

Report a bug

11.6.2. Request Header

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

Table 11.2. Request Header Fields

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

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

Report a bug

11.6.3. Response Header

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

Table 11.3. Response Header Fields

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

Report a bug

11.6.4. Topology Change Headers

11.6.4.1. About Topology Change Headers

When using Hot Rod to access 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

11.6.4.2. Topology Change Marker Values

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

Table 11.4. Topology Change Marker Field Values

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

Report a bug

11.6.4.3. 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 11.5. Topology Change Header Fields

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

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

Report a bug

11.6.4.4. 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 11.6. Topology Change Header Fields

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

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

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

Report a bug

11.7. HOT ROD OPERATIONS

11.7.1. Hot Rod Operations

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

- Get
- BulkGet
- GetWithVersion
- Put
- PutIfAbsent
- Remove
- RemoveIfUnmodified
- Replace
- ReplaceIfUnmodified
- Clear

- ContainsKey
- Ping
- Stats

Report a bug

11.7.2. Hot Rod Get Operation

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

Table 11.7. Get 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 **6** 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 11.8. 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 11.9. Get Operation Response Format

Field	Data Type	Details
Header	-	-

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

Report a bug

11.7.3. 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 JBoss Data Grid entries to be returned by the server. The entry count value equals the sum of the key and the associated value.

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	-	-
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

11.7.4. Hot Rod GetWithVersion Operation

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

Table 11.12. GetWithVersion 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 **6** 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 11.13. GetWithVersion Operation Response Format

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

The response for this operation contains the following:

Table 11.14.

Field	Data Type/Size	Details
Entry Version	8 bytes	Contains the unique value of an existing entry's modification.

Field	Data Type/Size	Details
Value Length	vInt	Contains the length of the value.
Value	Byte array	Contains the requested value.

Report a bug

11.7.5. Hot Rod Put Operation

The `put` operation request format includes the following:

Table 11.15.

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 11.16.

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

11.7.6. Hot Rod PutIfAbsent Operation

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

Table 11.17. 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 11.18.

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

11.7.7. Hot Rod Remove Operation

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

Table 11.19. 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 **`9`** 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 11.20. 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

11.7.8. Hot Rod RemoveIfUnmodified Operation

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

Table 11.21. RemoveIfUnmodified Operation Request Fields

Field	Data Type	Details
Header	-	-
Key Length	vInt	Contains the length of the key.
Key	Byte array	Contains the key value.
Entry Version	8 bytes	Uses the value returned by the **`GetWithVersion`** operation.

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

Table 11.22. 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

11.7.9. Hot Rod Replace Operation

The `replace` operation request format includes the following:

Table 11.23. Replace Operation Request Fields

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

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

Table 11.24. 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

11.7.10. Hot Rod ReplaceIfUnmodified Operation

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

Table 11.25. 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.
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 11.26. ReplaceIfUnmodified Operation Response

Response Status	Details
0x00	Returned status if the entry was replaced or removed.

Response Status	Details
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

11.7.11. Hot Rod Clear Operation

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

Valid response statuses for this operation are as follows:

Table 11.27. Clear Operation Response

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

Report a bug

11.7.12. Hot Rod ContainsKey Operation

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

Table 11.28. ContainsKey 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 **6** 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`**.

Field	Data Type	Details
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 11.29. 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

11.7.13. 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 11.30. Ping Operation Response

Response Status	Details
0x00	Successful ping without any errors.

Report a bug

11.7.14. 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 11.31. Stats Operation Request Fields

Name	Details
timeSinceStart	Contains the number of seconds since Hot Rod started.

Name	Details
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 11.32. 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

11.7.15. Hot Rod Operation Values

11.7.15.1. Opcode Request and Response Values

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

Table 11.33. 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

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

Report a bug

11.7.15.2. Magic Values

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

Table 11.34. Magic Field Values

Value	Details
0xA0	Cache request marker.

Value	Details
0xA1	Cache response marker.

Report a bug

11.7.15.3. Status Values

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

Table 11.35. 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

11.7.15.4. Transaction Type Values

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

Table 11.36. Transaction Type Field Values

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

Report a bug

11.7.15.5. Client Intelligence Values

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

Table 11.37. 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

11.7.15.6. Flag Values

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

Table 11.38. Flag Field Values

Value	Details
0x0001	ForceReturnPreviousValue

Report a bug

11.7.15.7. Hot Rod Error Handling

Table 11.39. 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.

Field	Data Type	Details
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. EXAMPLES

11.8.1. Put Request Example

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

Table 11.40. 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 11.41. Example Request Field Names and Values

Field Name	Byte	Value
Magic	0	0xA0
Version	2	0x41
Cache Name Length	4	0x07
Flag	12	0x00

Field Name	Byte	Value
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 11.42. 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 11.43. Example Response Field Names and Values

Field Name	Byte	Value
Magic	0	0xA1
Opcode	2	0x01
Topology Change Marker	4	0x00

Field Name	Byte	Value
Message ID	1	0x09
Status	3	0x00

Report a bug

11.9. CONFIGURE THE HOT ROD INTERFACE

11.9.1. About JBoss Data Grid Connectors

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

11.9.2. Configure Hot Rod Connectors

The following are the configuration elements for the **`hotrod-connector`** element in JBoss Data Grid's Remote Client-Server Mode.

```
<subsystem xmlns="urn:jboss:domain:datagrid:1.0">
      <hotrod-connector socket-binding="hotrod"
                        cache-container="default"
                        worker-threads="4"
                        idle-timeout="-1"
                        tcp-nodelay="true"
                        send-buffer-size="0"
                        receive-buffer-size="0"   />
      <topology-state-transfer lock-timeout="10000"
                               replication-timeout="10000"
                               update-timeout="30000"
                               external-host="192.168.0.1"
                               external-port="11222"
                               lazy-retrieval="true" />
</subsystem>
```

Report a bug

11.9.3. Hot Rod Connector Attributes

The following is a list of attributes used to configure the Hot Rod connector in JBoss Data Grid's Remote Client-Server Mode.

The Hotrod-Connector Element

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

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

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.

- 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.
- 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.
- 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.
- 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.
- 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.

- 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.

Report a bug

CHAPTER 12. THE REMOTECACHE INTERFACE

12.1. ABOUT THE REMOTECACHE INTERFACE

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

Report a bug

12.2. CREATE A NEW REMOTECACHEMANAGER

Use the following configuration to declaratively configure a new **RemoteCacheManager**:

```
Properties props = new Properties();
props.put("infinispan.client.hotrod.server_list", "127.0.0.1:11222");
RemoteCacheManager manager = new RemoteCacheManager(props);
RemoteCache defaultCache = manager.getCache();
```

NOTE

To learn more about using **Hot Rod** with JBoss Data Grid, refer to the *Developer Guide*'s Hot Rod Chapter.

Report a bug

PART III. ROLLING UPGRADES

CHAPTER 13. ROLLING UPGRADES IN JBOSS DATA GRID

13.1. ABOUT ROLLING UPGRADES

In 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 the application or risk losing data.

In JBoss Data Grid 6.1, rolling upgrades can only be performed in Remote Client-Server mode using Hot Rod.

Report a bug

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

The following process is used to perform rolling upgrades on 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.

Prerequisite

This procedure assumes you have a cluster 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", where as 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"`**).

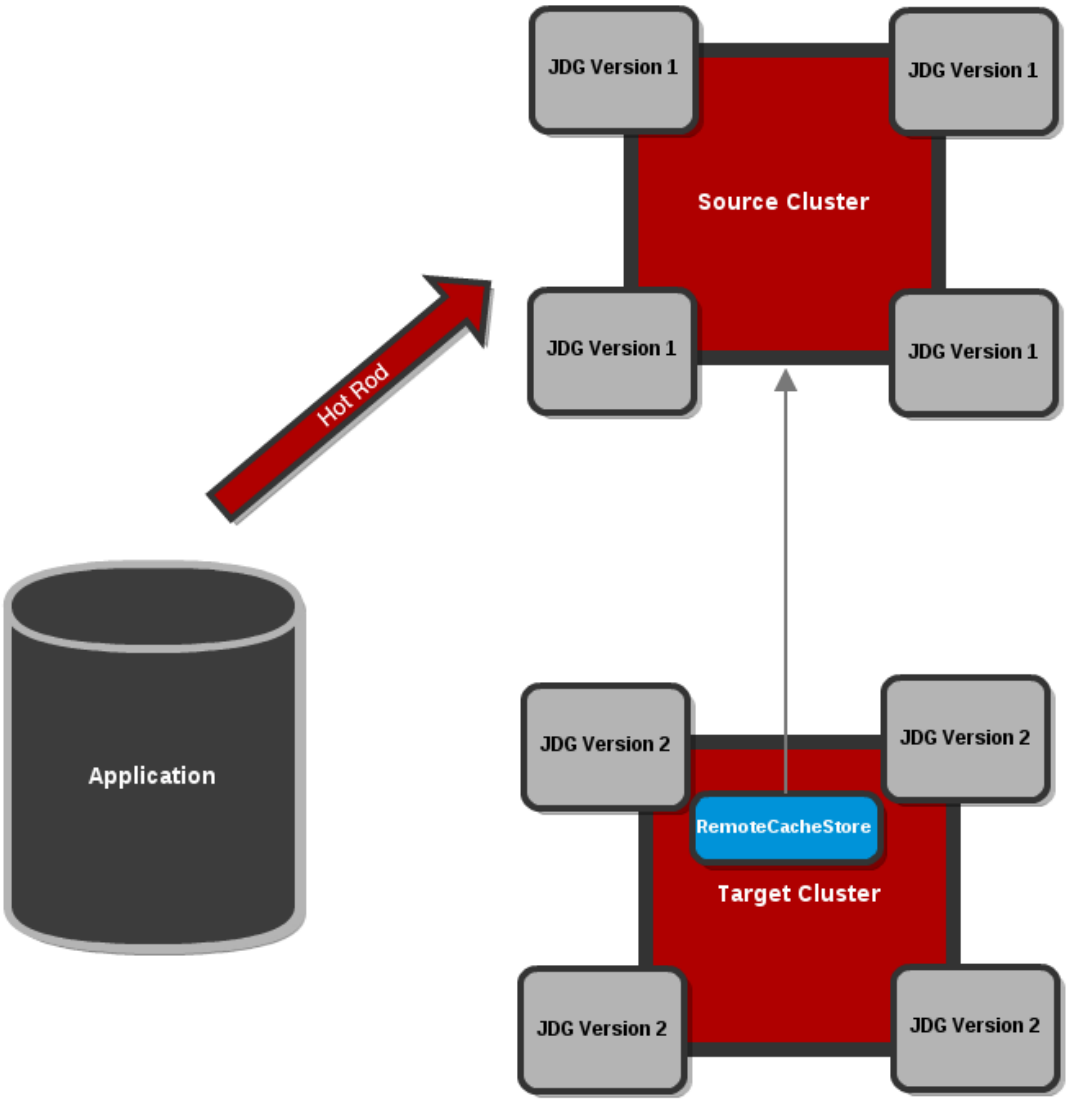


Figure 13.1. Configure the Target Cluster with a RemoteCacheStore

NOTE

Refer to 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:jboss:datagrid:infinispan:6.1"
           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"
                          hotrod-wrapping="true">
                <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.

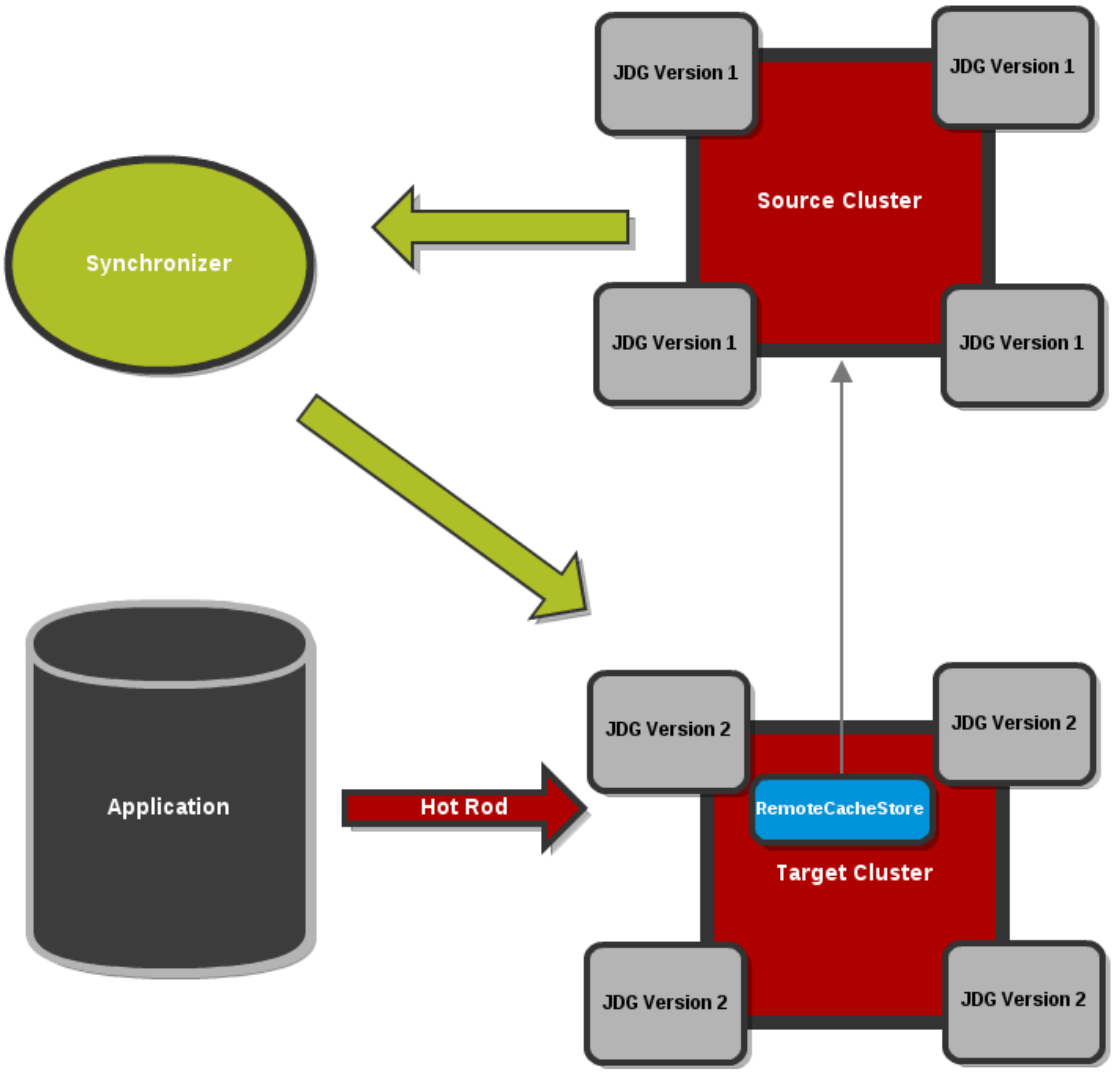


Figure 13.2. Clients point to the Target Cluster with the Source Cluster as `RemoteCacheStore` for the Target Cluster.

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 needs to be migrated.

   - **CLI**
     Invoke the **`upgrade --dumpkeys`** command on the Source Cluster for every cache that needs to 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 now needs to fetch 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 needs to be migrated.

- **CLI**
  Invoke the **`upgrade --synchronize=hotrod`** command on the Target Cluster for every cache that needs to 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

13.3. ROLLINGUPGRADEMANAGER OPERATIONS

The **`RollingUpgradeManager`** Mbean handles the operations that allow data to be migrated from one version of 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

PART IV. THE INFINISPAN CDI MODULE

CHAPTER 14. THE INFINISPAN CDI MODULE

14.1. ABOUT INFINSPAN CDI

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. (future feature?)
- Partial support for the JCACHE caching annotations.

Report a bug

14.2. USING INFINISPAN CDI

14.2.1. Infinispan CDI Prerequisites

The following is a list of prerequisites to use the Infinispan CDI module with 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

14.2.2. 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

14.3. USING THE INFINISPAN CDI MODULE

14.3.1. 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

14.3.2. Configure and Inject Infinispan Caches

14.3.2.1. 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

14.3.2.2. 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

14.3.2.3. Set the Injection's Target Cache

14.3.2.3.1. Setting 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

14.3.2.3.2. 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

14.3.2.3.3. 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

14.3.2.3.4. 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

14.3.3. Configure Cache Managers

14.3.3.1. Configuring Cache Managers with CDI

A 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

14.3.3.2. Specify the Default Configuration

Specify a method annotated as a producer for the 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

14.3.3.3. Override the Creation of the Embedded Cache Manager

Prerequisites

Section 14.3.3.2, “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 ocne.

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 two 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

14.3.3.4. 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() {
  return new RemoteCacheManager(ADDRESS, PORT);
 }
}
```

Report a bug

14.3.3.5. 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 EmbeddedCacheManager
  defaultEmbeddedCacheManager() {
   Configuration cfg = new ConfigurationBuilder()
     .eviction()
     .strategy(EvictionStrategy.LRU)
     .maxEntries(150)
     .build();
   return new DefaultCacheManager(cfg);
  }

 @Produces
 @ApplicationScoped
 @DefaultClustered
 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);
 }

 @Produces
 @ApplicationScoped
 @DefaultRemote
 public RemoteCacheManager
  defaultRemoteCacheManager() {
   return new RemoteCacheManager(ADDRESS, PORT);
  }

 @Produces
```

```
 @ApplicationScoped
 @RemoteCacheInDifferentDataCentre
 public RemoteCacheManager newRemoteCacheManager() {
  return new RemoteCacheManager(ADDRESS_FAR_AWAY, PORT);
 }
}
```

Report a bug

14.3.4. Storage and Retrieval Using CDI Annotations

14.3.4.1. 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

14.3.4.2. 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.cdi.interceptor.CacheResultInterceptor
  </class>
  <class>
   org.infinispan.cdi.interceptor.CachePutInterceptor
  </class>
  <class>
   org.infinispan.cdi.interceptor.CacheRemoveEntryInterceptor
  </class>
  <class>
   org.infinispan.cdi.interceptor.CacheRemoveAllInterceptor
  </class>
 </interceptors>
</beans>
```

Report a bug

14.3.4.3. Catch the Result of a Method Invocation

14.3.4.3.1. Catching 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";
}
```

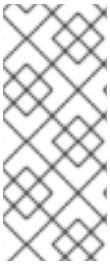
A common approach is to cache the results of this method call and to check the cache when the result is next required. The following is an example of a code snippet that looks up the result of such an operation in a cache. If the results are not found, the code snippet runs the **`toCelsiusFormatted`**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

```
@javax.cache.interceptor.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

14.3.4.3.2. 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

14.3.4.3.3. 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 **@CacheKeyParam** annotation as follows:

```
@CacheResult
public void doSomething
 (@CacheKeyParam String p1,
 @CacheKeyParam 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

14.3.4.3.4. Generate a Custom Key

Generate a custom key as follows:

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

public class MyCacheKeyGenerator implements CacheKeyGenerator {

   @Override
   public CacheKey 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

14.3.4.3.5. 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.CacheKey;

public class MyCacheKey implements CacheKey {
   private Object p;

   public CustomCacheKey(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 **CacheKey** to work as expected.

Report a bug

14.3.5. Cache Operations

14.3.5.1. 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:

```
@CachePut (cacheName = "personCache")
public void updatePerson
 (@CacheKeyParam 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 **@CacheKeyParam** to control key generation.

See Also:

- Section 14.3.4.4.3.3, "Cache Keys for Cached Results"

Report a bug

14.3.5.2. 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:

```
@CacheRemoveEntry (cacheName = "cacheOfPeople")
public void changePersonName
 (@CacheKeyParam long personId,
 string newName {
...
 }
```

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

Report a bug

14.3.5.3. 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

```
@CacheRemoveAll (cacheName = "statisticsCache")
public void resetStatistics() {
...
}
```

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

Report a bug

PART V. QUERYING IN JBOSS DATA GRID

CHAPTER 15. THE QUERY MODULE

15.1. ABOUT THE QUERY MODULE

Users have the ability to query the entire stored data set for specific items in JBoss Data Grid. Applications may not always be aware of specific keys, however different parts of a value can be queried using the Query Module.

The JBoss Data Grid Query Module utilizes the capabilities of **Infinispan Query** and **Apache Lucene** to index search objects in the cache. The Query Module uses the **Infinispan Query** internally and independent of the data source. This allows objects to be located within the cache based on their properties, rather than requiring the keys for each object.

Objects can be searched for based on some of their properties. For example:

- Retrieve all red cars (an exact metadata match).
- Search for all books about a specific topic (full text search and relevance scoring).

An exact data match can also be implemented with the MapReduce function, however full text and relevance based scoring can only be performed via the Query Module.

WARNING

The Query Module is in Technical Preview for JBoss Data Grid 6.1. Despite the Query Module itself not being supported, undocumented APIs and packages must not be used in JBoss Data Grid 6.1.

Report a bug

15.2. APACHE LUCENE AND INFINISPAN QUERY

In order to perform querying on the entire data set stored in the distributed grid, JBoss Data Grid utilizes the capabilities of the **Apace Lucene** indexing tool, as well as **Infinispan Query**.

- **Apache Lucene** is a document indexing tool and search engine. JBoss Data Grid uses **Apache Lucene 3.6**.
- **Infinispan Query** is a toolkit based on **Hibernate Search** that reduces Java objects into a format similar to a document, which is able to be indexed and queried by **Apache Lucene**.

In JBoss Data Grid, the Query Module indexes keys and values annotated with **Infinispan Query** indexing annotations, then updates the index based in **Apache Lucene** accordingly.

IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

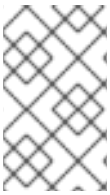
15.3. ABOUT LUCENE DIRECTORY

The **Lucene Directory** is the Input Output API for **Apache Lucene** to store the query indexes.

The most common **Lucene Directory** implementations used with JBoss Data Grid's Query Module are:

- Ram - stores the index in a local map to the node. This index cannot be shared.
- File system - stores the index in a locally mounted file system. This could be a network shared file system, however sharing in this manner is not recommended.
- JBoss Data Grid - stores the indexes in a different set of dedicated JBoss Data Grid caches. These caches can be configured as replicated or distributed in order to share the index between nodes.

The Query Module is not aware of where indexes are stored.



NOTE

The Lucene Directory provided by JBoss Data Grid is not limited to the Query Module. It can seamlessly replace any other requirement to store Lucene indexes where your application uses Lucene directly.

The JBoss Data Grid Query Module ships with several **Lucene Directory** implementations, and accepts third party implementations.

IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

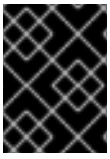
Report a bug

15.4. INFINISPAN QUERY

15.4.1. Infinispan Query in JBoss Data Grid

In JBoss Data Grid, the identifier for all ***@Indexed*** objects is the key used to store the value. How the key is indexed can still be customized by using a combination of ***@Transformable***, ***@ProvidedID***, custom types and custom ***FieldBridge*** implementations.

The ***@DocumentID*** identifier does not apply to JBoss Data Grid values.



IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

15.4.2. Infinispan Query Dependencies for JBoss Data Grid

To run Infinispan Query in JBoss Data Grid, you must have installed:

- JBoss Data Grid
- A JVM
- Maven

To use the JBoss Data Grid directory for Infinispan Query via Maven, the following dependency must be added:

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

To store indexes in JBoss Data Grid, apply the following dependency:

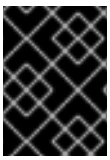
```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-search-infinispan</artifactId>
   <version>4.2.0.Final-redhat-1</version>
</dependency>
```

Optionally, analysis support can be included by applying the following dependency:

```
<groupId>org.hibernate</groupId>
<artifactId>hibernate-search-analyzers</artifactId>
<version>4.2.0.Final-redhat-1</version>
```

Non-Maven users must install all `.jar` files from the JBoss Data Grid distribution.

For more detailed information about configuring Hibernate Search, refer to the JBoss Web Framework Kit *Hibernate Search* guide.



IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

15.4.3. Configuring Infinispan Query for JBoss Data Grid

The following procedure shows how to set up a simple configuration for using Infinispan Query with JBoss Data Grid.

1. Enable the Infinispan Query default configuration and cluster-replicated index for JBoss Data Grid's Query Module, using either **"default"** or specifying the name of an index. The default configuration does not enable any form of persistence for the index.

   ```
   hibernate.search.[default|<indexname>].directory_provider infinispan
   ```

   It is possible to use multiple named indexes, so configuration can be applied to all or overridden in a per-index base by using the index name.

   In order to enable a persistent index, either a JBoss Data Grid configuration file must be provided, or point to JNDI as described in Step 2.

2. Infinispan Query requires a **CacheManager**, which can be obtained in two ways.

   - Look up and reuse an existing **CacheManager** programmatically using **Java Naming and Directory Interface** (JNDI). To use this method, the following property must be set in addition to the property specified in Step 1:

     ```
     hibernate.search.[default|
     <indexname>].infinispan.cachemanager_jndiname = [jndiname]
     ```

   - Start and manage a new **CacheManager**. To use a new **CacheManager** apply the following:

     ```
     hibernate.search.[default|
     <indexname>].infinispan.configuration_resourcename = [infinispan
     configuration filename]
     ```

   If both parameters are defined, the JNDI will be prioritized.

   If neither of these parameters are defined, Infinispan Query will use the default JBoss Data Grid configuration included in **hibernate-search-infinispan.jar**, which does not store the index in a persistent cache store.

Specific methods, classes, and packages for this feature are unsupported in JBoss Data Grid, and should not be used. For a full list of prohibited methods, classes, and packages for this feature, see the "Prohibited Classes, Methods, and Packages" section of the *JBoss Data Grid 6.1 Release Notes*

IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

15.5. CONFIGURING THE QUERY MODULE

15.5.1. Configure the Query Module

Indexing must be enabled in either the XML configuration or the programmatic configuration. Once this has been enabled, you can use the **Search** capabilities using the **SearchManager**, which exposes methods to perform queries.

IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

15.5.2. Configure Indexing using XML

Indexing can be configured in XML by adding the ***<indexing ... />*** element to the cache configuration in the Infinispan core configuration file, and optionally pass additional properties in the embedded **Infinispan Query** engine. For Example:

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:5.2
http://www.infinispan.org/schemas/infinispan-config-5.2.xsd"
      xmlns="urn:infinispan:config:5.2">
   <default>
      <indexing enabled="true" indexLocalOnly="true">
         <properties>
            <property name="default.directory_provider" value="ram" />
         </properties>
      </indexing>
   </default>
</infinispan>
```

In this example, the index is stored in memory. As a result, when the relevant nodes shut down the index is lost. This arrangement is ideal for brief demonstration purposes, but in real world applications, use the default (store on file system) or store the index in JBoss Data Grid to persist the index.

hibernate.search is also a valid optional prefix for configuration ***property*** keys.

IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

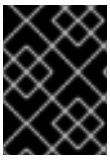
15.5.3. Configure Indexing Programmatically

Indexing can be configured programmatically, avoiding XML configuration files.

In this example, JBoss Data Grid is started programmatically and also maps an object ***Author***, which is stored in the grid and made searchable via two properties, without annotating the class.

```
SearchMapping mapping = new SearchMapping();
```

```
mapping.entity(Author.class).indexed().providedId()
      .property("name", ElementType.METHOD).field()
      .property("surname", ElementType.METHOD).field();

Properties properties = new Properties();
properties.put(org.hibernate.search.Environment.MODEL_MAPPING, mapping);
properties.put("hibernate.search.[other options]", "[...]");

Configuration infinispanConfiguration = new ConfigurationBuilder()
      .indexing()
         .enable()
         .indexLocalOnly(true)
         .withProperties(properties)
      .build();

DefaultCacheManager cacheManager = new
DefaultCacheManager(infinispanConfiguration);

Cache<Long, Author> cache = cacheManager.getCache();
SearchManager sm = Search.getSearchManager(cache);

Author author = new Author(1, "FirstName", "Surname");
cache.put(author.getId(), author);

QueryBuilder qb = sm.buildQueryBuilderForClass(Author.class).get();
Query q =
qb.keyword().onField("name").matching("FirstName").createQuery();
CacheQuery cq = sm.getQuery(q, Author.class);
Assert.assertEquals(cq.getResultSize(), 1);
```



IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

15.5.4. Rebuilding the Index

The Lucene index can be rebuilt, if required, by reconstructing it from the data store in the cache.

The index must be rebuilt if:

- The definition of what is indexed in the types has changed.
- A parameter affecting how the index is defined, such as the ***`Analyser`*** changes.
- The index is destroyed or corrupted, possibly due to a system administration error.

To rebuild the index, obtain a reference to the **`MassIndexer`** and start it as follows:

```
SearchManager searchManager = Search.getSearchManager(cache);
searchManager.getMassIndexer().start();
```

This operation reprocesses all data in the grid, and therefore may take some time.

Rebuilding the index is also available as a JMX operation.

NOTE

The `MassIndexer` is implemented using `MapReduce`, and is therefore only functional in distributed cache mode.

Report a bug

15.6. ANNOTATING OBJECTS AND STORING INDEXES

15.6.1. Annotating Objects for Infinispan Query

Once indexing has been enabled, custom objects being stored in JBoss Data Grid need to be assigned appropriate **Infinispan Query** annotations.

As a basic requirement, all objects required to be indexed must be annotated with

- ***@Entity***
- ***@Indexed***
- ***@ProvidedId***

In addition, all fields within the object that will be searched need to be annotated with ***@Field***.

For example:

```
@Entity @ProvidedId @Indexed
public class Person
 implements Serializable {
  @Field(store = Store.YES)
  private String name;
  @Field(store = Store.YES)
  private String description;
  @Field(store = Store.YES)
  private int age;
...
}
```

For more useful annotations and options, refer to the JBoss Web Framework Kit *Hibernate Search* guide.

IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

15.6.2. Registering a Transformer via Annotations

The key for each value must also be indexed, and the key instance must then be transformed in a String.

JBoss Data Grid includes some default transformation routines for encoding common primitives, however to use a custom key you must provide an implementation of ***`org.infinispan.query.Transformer`***.

The following example shows how to annotate your key type using ***`org.infinispan.query.Transformer`***:

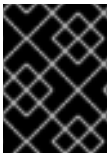
```
@Transformable(transformer = CustomTransformer.class)
public class CustomKey {
   ...
}

public class CustomTransformer implements Transformer {
   @Override
   public Object fromString(String s) {
      ...
      return new CustomKey(...);
   }

   @Override
   public String toString(Object customType) {
      CustomKey ck = (CustomKey) customType;
      return ...
   }
}
```

The two methods must implement a biunique correspondence.

For example, for any object A the following must be true:

```
A.equals( transformer.fromString( transformer.toString( A ) )
```

This assumes that the transformer is the appropriate Transformer implementation for objects of type A.



IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

15.7. CACHE MODES AND STORING INDEXES

15.7.1. Storing Lucene Indexes

In JBoss Data Grid's Query Module, **Lucene** is used to store and manage indexes. **Lucene** ships with several index storage subsystems, also known as directories.

These include directories for the purpose of:

- simple, in-memory storage.
- file system storage.

To configure the storage of indexes, set the appropriate properties when enabling indexing in the JBoss Data Grid configuration.

The following example demonstrates an in-memory, RAM-based index store:

```
<namedCache name="indexesInMemory">
 <indexing enabled="true">
  <properties>
   <property name=
    "default.directory_provider" value="ram"/>
  </properties>
 </indexing>
</namedCache>
```

This second example shows a disk-based index store:

```
<namedCache name="indexesOnDisk">
 <indexing enabled="true">
  <properties>
   <property name=
    "default.directory_provider" value="filesystem"/>
  </properties>
 </indexing>
</namedCache>
```

`hibernate.search` is also a valid optional prefix for configuration ***`property`*** keys.

IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

15.7.2. The Infinispan Directory

In addition to the **Lucene** directory implementations, JBoss Data Grid also ships with an **`infinispan-directory`** module.

The **infinispan-directory** allows **Lucene** to store indexes within the distributed data grid. This allows the indexes to be distributed , stored in-memory, and optionally written to disk using the cache store for durability.

This can be configured by having the named cache store indexes in JBoss Data Grid. For example:

```
<namedCache name="indexesInInfinispan">
 <indexing enabled="true">
  <properties>
   <property name=
    "default.directory_provider" value="infinispan"/>
   <property name=
    "default.exclusive_index_use" value="false"/>
  </properties>
 </indexing>
</namedCache>
```

hibernate.search is also a valid optional prefix for configuration ***property*** keys.

Sharing the same index instance using the Infinispan **Directory Provider**, introduces a write contention point, as only one instance can write on the same index at the same time. The property ***exclusive_index_use*** must be set to **"false"** and in most cases an alternative backend must be setup.

For more detailed information, refer to the JBoss Web Framework Kit *Hibernate Search* guide.

The default backend can be used if there is very low contention on writes, or if the application can guarantee all writes on the index are originated on the same node.

IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

15.7.3. Cache Modes and Managing Indexes

In JBoss Data Grid's Query Module there are two options for storing indexes:

1. Each node can maintain an individual copy of the global index.
2. The index can be shared across all nodes.

IMPORTANT

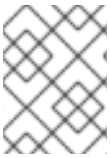
The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

15.7.4. Storing Global Indexes Locally

Storing the global index locally in JBoss Data Grid's Query Module allows each node to

- maintain its own index.
- use **Lucene**'s in-memory or filesystem-based index directory.



NOTE

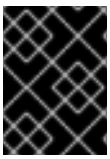
The JBoss Data Grid cluster must be operating in replicated mode in order to ensure each node's indexes are always up to date.

When enabling indexing with the global index stored locally, the ***indexLocalOnly*** attribute of the ***indexing*** element must be set to **"false"** in order for changes originating from elsewhere in the cluster are indexed.

The following example shows how to configure storing the global index as a local copy:

```
<namedCache name="localCopyOfGlobalIndexes">
 <clustering mode="replicated"/>
 <indexing enabled="true" indexLocalOnly="false">
  <property name=
   "default.directory_provider"
   value="ram"/>
 </indexing>
</namedCache>
```

hibernate.search is also a valid optional prefix for configuration ***property*** keys.



IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

15.7.5. Sharing the Global Index

The Query Module in JBoss Data Grid has the option to have a single set of indexes shared by all nodes. The only **Lucene** directories supported in this mode, and where indexes can be made available to the entire cluster are:

- The JBoss Data Grid directory provider. Either replicated or distributed cache modes can be used when sharing the indexes in this manner.
- A local filesystem-based index, which is periodically synchronized with other nodes using simple file copy. This requires a shared network drive configured externally.

When enabling shared indexes, the ***indexLocalOnly*** attribute of the ***indexing*** element must be set to **"true"**. For example:

```
<namedCache name="globalSharedIndexes">
 <clustering mode="distributed"/>
 <indexing enabled="true" indexLocalOnly="true">
  <property name=
```

```
    "default.directory_provider" value="infinispan"/>
   <property name=
    "default.exclusive_index_use" value="false"/>
  </indexing>
</namedCache>
```

hibernate.search is also a valid optional prefix for configuration ***property*** keys.

IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

15.8. QUERYING EXAMPLE

15.8.1. The Query Module Example

The following provides an example of how to set up and run a query in JBoss Data Grid.

In this example, the "Person" object has been annotated using the following:

```
@Entity @ProvidedId @Indexed
public class Person
 implements Serializable {
  @Field(store = Store.YES)
  private String name;
  @Field
  private String description;
  @Field(store = Store.YES)
  private int age;
...
}
```

Assuming several of these "Person" objects have been stored in JBoss Data Grid, they can be searched using querying. The following code creates a ***SearchManager*** and ***QueryBuilder*** instance:

```
SearchManager manager=
 Search.getSearchManager(cache);
QueryBuilder builder=
 sm.buildQueryBuilderForClass(Person.class) .get();
Query luceneQuery = builder.keyword()
   .onField("name")
   .matching("FirstName")
  .createQuery();
```

The ***SearchManager*** and ***QueryBuilder*** are used to construct a **Lucene** query. The **Lucene** query is then passed to the ***SearchManager*** to obtain a ***CacheQuery*** instance:

```
CacheQuery query = manager.getQuery(luceneQuery);
for (Object result: query) {
        System.out.println("Found " + result);
```

```
}
```

This ***CacheQuery*** instance contains the results of the query, and can be used to produce a list or it can be used for repeat queries.

IMPORTANT

The Query Module is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

PART VI. MAPREDUCE AND DISTRIBUTED TASKS

CHAPTER 16. MAPREDUCE

16.1. ABOUT MAPREDUCE

The 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 very large amounts of data across the data grid by performing most computations as locally possible to where the data is stored.

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(Reducer)`***, the `Combiner` is called on the `Mapper` results and the combiner's results are retured 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

16.2. THE MAPREDUCE API

16.2.1. The MapReduce API

In 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 the 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 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

16.2.2. MapReduceTask

In 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

16.2.3. Mapper and CDI

The **Mapper** is invoked with appropriate input key/value pairs on an executing node, however 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 WordCountCacheInjectedMapper 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

16.3. MAPREDUCETASK DISTRIBUTED EXECUTION

16.3.1. 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 16.1. 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 16.2. 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 16.3. 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

16.4. MAP REDUCE EXAMPLE

The following example uses a word count application to demonstrate MapReduce and it's 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","JBoss Application Server");
      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>
c) {
         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 17. DISTRIBUTED EXECUTION

17.1. ABOUT DISTRIBUTED EXECUTION

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

17.2. 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 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

17.3. 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 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

17.4. DISTRIBUTED TASK FAILOVER

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

17.5. DISTRIBUTED TASK EXECUTION POLICY

The **`DistributedTaskExecutionPolicy`** allows tasks to specify a custom execution policy across the 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 you want to use only a dedicated subset of a certain JBoss Data Grid rack nodes 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

17.6. 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:
   $C = S\pi/4$

   $4C = S\pi$

$$4C/S = \pi$$

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

PART VII. MARSHALLING IN JBOSS DATA GRID

CHAPTER 18. MARSHALLING

18.1. ABOUT MARSHALLING

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

Report a bug

18.2. MARSHALLING BENEFITS

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

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

Report a bug

18.3. ABOUT MARSHALLING FRAMEWORK

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

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

Report a bug

18.4. SUPPORT FOR NON-SERIALIZABLE OBJECTS

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

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

NOTE

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

Report a bug

18.5. HOT ROD AND MARSHALLING

In Remote Client-Server mode, marshalling occurs both on the 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 needs to 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

18.6. CONFIGURING THE MARSHALLER USING THE REMOTECACHEMANAGER

The ***`infinispan.client.hotrod.marshaller`*** allows you to specify a custom Marshaller implementation to serialize and deserialize user objects.

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

The following example shows the default value the JBoss Marshaller.

```
Properties props = new Properties();
props.put("infinispan.client.hotrod.marshaller",
 "org.infinispan.marshall.jboss.GenericJBossMarshaller"
RemoteCacheManager remoteCacheManager = new RemoteCacheManager(props);
```

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 Library mode.

Report a bug

18.7. TROUBLESHOOTING

18.7.1. Marshalling Troubleshooting

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

The following is an example of such a stack trace:

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

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

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

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

```
java.io.NotSerializableException: java.lang.Object
...
Caused by: an exception which occurred:
in object java.lang.Object@b40ec4
-> toString = java.lang.Object@b40ec4
```

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

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

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

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

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

```
java.io.IOException: Injected failue!
...
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.VersionAwareMarshallerTest$1
-> classloader hierarchy:
-> type classloader = sun.misc.Launcher$AppClassLoader@198dfaf
-
>...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/ecl
ipse-testng.jar
```

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

Report a bug

18.7.2. State Receiver EOFExceptions

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

```
2010-12-09 10:26:21,533 20267 ERROR
[org.infinispan.remoting.transport.jgroups.JGroupsTransport]
(STREAMING_STATE_TRANSFER-sender-1,Infinispan-Cluster,NodeJ-2368:) Caught
while responding to state transfer request
org.infinispan.statetransfer.StateTransferException:
java.util.concurrent.TimeoutException: Could not obtain exclusive
processing lock
     at
org.infinispan.statetransfer.StateTransferManagerImpl.generateState(StateT
ransferManagerImpl.java:175)
     at
org.infinispan.remoting.InboundInvocationHandlerImpl.generateState(Inbound
InvocationHandlerImpl.java:119)
     at
org.infinispan.remoting.transport.jgroups.JGroupsTransport.getState(JGroup
sTransport.java:586)
     at
org.jgroups.blocks.MessageDispatcher$ProtocolAdapter.handleUpEvent(Message
Dispatcher.java:691)
     at
org.jgroups.blocks.MessageDispatcher$ProtocolAdapter.up(MessageDispatcher.
java:772)
     at org.jgroups.JChannel.up(JChannel.java:1465)
     at org.jgroups.stack.ProtocolStack.up(ProtocolStack.java:954)
     at org.jgroups.protocols.pbcast.FLUSH.up(FLUSH.java:478)
     at
org.jgroups.protocols.pbcast.STREAMING_STATE_TRANSFER$StateProviderHandler
.process(STREAMING_STATE_TRANSFER.java:653)
     at
org.jgroups.protocols.pbcast.STREAMING_STATE_TRANSFER$StateProviderThreadS
pawner$1.run(STREAMING_STATE_TRANSFER.java:582)
     at
java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.
java:886)
     at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java
:908)
     at java.lang.Thread.run(Thread.java:680)
Caused by: java.util.concurrent.TimeoutException: Could not obtain
exclusive processing lock
     at
org.infinispan.remoting.transport.jgroups.JGroupsDistSync.acquireProcessin
gLock(JGroupsDistSync.java:71)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.generateTransactionL
og(StateTransferManagerImpl.java:202)
     at
org.infinispan.statetransfer.StateTransferManagerImpl.generateState(StateT
ransferManagerImpl.java:165)
     ... 12 more
```

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

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

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

Report a bug

PART VIII. TRANSACTIONS

CHAPTER 19. TRANSACTIONS

19.1. ABOUT JAVA TRANSACTION API TRANSACTIONS

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

JBoss Data Grid does the following for each cache operation:

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

IMPORTANT

With JBoss Data Grid 6.0.x, it is recommended to disable transactions in Remote Client-Server Mode. However, if an error displays warning of an **`ExceptionTimeout`** where JBoss Data Grid is **`Unable to acquire lock after {time} on key {key} for requester {thread}`**, enable transactions. This occurs because non-transactional caches acquire locks on each node they write on. Using transactions prevents deadlocks because caches acquire locks on a single node. This problem is resolved in JBoss Data Grid 6.1.

Report a bug

19.2. TRANSACTIONS SPANNING MULTIPLE CACHE INSTANCES

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

Report a bug

19.3. TRANSACTION/BATCHING AND INVALIDATION MESSAGES

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

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

Report a bug

CHAPTER 20. THE TRANSACTION MANAGER

20.1. ABOUT JTA TRANSACTION MANAGER LOOKUP CLASSES

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

Table 20.1. Transaction Manager Lookup Classes

Class Name	Details
org.infinispan.transaction.lookup.DummyTransactionManagerLookup	Used primarily for testing environments. This testing transaction manager is not for use in a production environment and is severely limited in terms of functionality, specifically for concurrent transactions and recovery.
org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup	The default transaction manager when JBoss Data Grid runs in a standalone environment. It is a fully functional JBoss Transactions based transaction manager that overcomes the functionality limits of the **`DummyTransactionManager`**.
org.infinispan.transaction.lookup.GenericTransactionManagerLookup	Used to locate transaction managers in most Java EE application servers. If no transaction manager is located, it defaults to **`DummyTransactionManager`**.
org.infinispan.transaction.lookup.JBossTransactionManagerLookup	Locates a transaction manager within a JBoss Application Server instance.

Report a bug

20.2. OBTAIN THE TRANSACTION MANAGER FROM THE CACHE

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

Procedure 20.1. Obtain the Transaction Manager from the Cache

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

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

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

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

Report a bug

20.3. TRANSACTION MANAGER AND XARESOURCES

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

Report a bug

20.4. OBTAIN A XARESOURCE REFERENCE

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

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

Report a bug

20.5. DEFAULT DISTRIBUTED TRANSACTION BEHAVIOR

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

Report a bug

20.6. TRANSACTION SYNCHRONIZATION

20.6.1. About Transaction Synchronizations

Synchronizations are a type of listener that receives notifications about events leading to the transaction life cycle. Synchronizations receive an event just before and just after completion of an operation.

Synchronizations are primarily useful when specific tasks must be triggered before or after an event. An common application for synchronizations are clearing out an application's caches after an operation (such as a transaction) concludes.

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

Report a bug

CHAPTER 21. DEADLOCK DETECTION

21.1. ABOUT DEADLOCK DETECTION

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

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

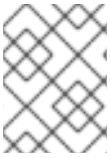
Report a bug

21.2. ENABLE DEADLOCK DETECTION

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

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

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



NOTE

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

Report a bug

PART IX. LISTENERS AND NOTIFICATIONS

CHAPTER 22. LISTENERS AND NOTIFICATIONS

22.1. ABOUT THE LISTENER API

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

Report a bug

22.2. LISTENER EXAMPLE

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

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

Report a bug

22.3. CACHE ENTRY MODIFIED LISTENER CONFIGURATION

In a cache entry modified listener, the modified value cannot be retrieved via **`Cache.get()`** when ***`isPre`*** (an Event method) is **`false`**. For more information about ***`isPre()`***, refer to the JBoss Data Grid *API Documentation*'s listing for the **`org.infinispan.notifications.cachelistener.event`** package.

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

Report a bug

22.4. NOTIFICATIONS

22.4.1. About Listener Notifications

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

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

Report a bug

22.4.2. About Cache-level Notifications

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

Report a bug

22.4.3. Cache Manager-level Notifications

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

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

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

Report a bug

22.4.4. About Synchronous and Asynchronous Notifications

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

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

Annotate listeners using the following:

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

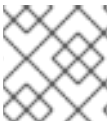
Use the ***<asyncListenerExecutor/>*** element in the configuration file to tune the thread pool that is used to dispatch asynchronous notifications.

Report a bug

22.5. NOTIFYING FUTURES

22.5.1. About NotifyingFutures

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



NOTE

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

Report a bug

22.5.2. NotifyingFutures Example

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

```
FutureListener futureListener = new FutureListener() {

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

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

Report a bug

PART X. THE INFINISPAN CLI

CHAPTER 23. THE INFINISPAN CLI

23.1. ABOUT THE INFINISPAN CLI

JBoss Data Grid includes the Infinispan Command Line Interface (CLI) that is used to inspect and modify data within caches and internal components (such as transactions, cross-datacentre replication sites, and rolling upgrades). The Infinispan CLI can also be used for more advanced operations such as transactions.

The CLI consists of a server-side module and a client command tool. The server-side module (**infinispan-cli-server-*$VERSION*.jar**) includes an interpreter for commands and must be included in the application.

IMPORTANT

The Infinispan CLI is currently only available as a Technical Preview for JBoss Data Grid 6.1.

Report a bug

23.2. START THE CLI (SERVER)

Start the Infinispan CLI's server-side module with the **standalone** and **cluster** files. For Linux, use the **standlaone.sh** or **clustered.sh** script and for Windows, use the **standalone.bat** or **clustered.bat** file.

Report a bug

23.3. START THE CLI (CLIENT)

Start the Infinispan CLI client using the **ispn-cli** file at **bin/**. For Linux, run **bin/ispn-cli.sh** and for Windows, run **bin/ispn-cli.bat**.

When starting up the CLI client, specific command line switches can be used to customize the start up.

Report a bug

23.4. CLI CLIENT SWITCHES FOR THE COMMAND LINE

The listed command line switches are appended to the command line when starting the Infinispan CLI command:

Table 23.1. CLI Client Command Line Switches

Short Option	Long Option	Description

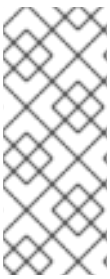
Short Option	Long Option	Description
-c	--connect=${URL}	Connects to a running JBoss Data Grid instance. For example, for JMX over RMI use **`jmx://[username[:password]]@host:port[/container[/cache]]`** and for JMX over JBoss Remoting use **`remoting://[username[:password]]@host:port[/container[/cache]]`**
-f	--file=${FILE}	Read the input from the specified file rather than using interactive mode. If the value is set to **-** then the ***`stdin`*** is used as the input.
-h	--help	Displays the help information.
-v	--version	Displays the CLI version information.

Report a bug

23.5. CONNECT TO THE APPLICATION

Use the following command to connect to the application using the CLI:

```
[disconnected//]> connect jmx://localhost:12000
[jmx://localhost:12000/MyCacheManager/>
```



NOTE

The port value **12000** depends on the value the JVM is started with. For example, starting the JVM with the **`-Dcom.sun.management.jmxremote.port=12000`** command line parameter uses this port, but otherwise a random port is chosen. When the remoting protocol (**`remoting://localhost:9999`**) is used, the JBoss Data Grid server administration port is used (the default is port **9999**).

The command line prompt displays the active connection information, including the currently selected **CacheManager**.

Use the **cache** command to select a cache before performing cache operations. The CLI supports tab completion, therefore using the **cache** and pressing the tab button displays a list of active caches:

```
[[jmx://localhost:12000/MyCacheManager/> cache
___defaultcache  namedCache
[jmx://localhost:12000/MyCacheManager/]> cache ___defaultcache
[jmx://localhost:12000/MyCacheManager/___defaultcache]>
```

Additionally, pressing tab displays a list of valid commands for the CLI.

Report a bug

23.6. CLI COMMANDS

23.6.1. The abort Command

The **abort** command aborts a running batch initiated using the **start** command. Batching must be enabled for the specified cache. The following is a usage example:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> start
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> abort
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
null
```

Report a bug

23.6.2. The begin Command

The **begin** command starts a transaction. This command requires transactions enabled for the cache it targets. An example of this command's usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> begin
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> commit
```

Report a bug

23.6.3. The cache Command

The **cache** command specifies the default cache used for all subsequent operations. If invoked without any parameters, it shows the currently selected cache. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> cache ___defaultcache
[jmx://localhost:12000/MyCacheManager/___defaultcache]> cache
___defaultcache
[jmx://localhost:12000/MyCacheManager/___defaultcache]>
```

Report a bug

23.6.4. The clear Command

The **clear** command clears all content from the cache. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> clear
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
null
```

Report a bug

23.6.5. The commit Command

The **commit** command commits changes to an ongoing transaction. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> begin
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> commit
```

Report a bug

23.6.6. The container Command

The **container** command selects the default cache container (cache manager). When invoked without any parameters, it lists all available containers. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> container
MyCacheManager OtherCacheManager
[jmx://localhost:12000/MyCacheManager/namedCache]> container
OtherCacheManager
[jmx://localhost:12000/OtherCacheManager/]>
```

Report a bug

23.6.7. The create Command

The **create** command creates a new cache based on the configuration of an existing cache definition. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> create newCache like
namedCache
[jmx://localhost:12000/MyCacheManager/namedCache]> cache newCache
[jmx://localhost:12000/MyCacheManager/newCache]>
```

Report a bug

23.6.8. The disconnect Command

The **disconnect** command disconnects the currently active connection, which allows the CLI to connect to another instance. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> disconnect
[disconnected//]
```

Report a bug

23.6.9. The encoding Command

The **encoding** command sets a default codec to use when reading and writing entries to and from a cache. If invoked with no arguments, the currently selected codec is displayed. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> encoding
none
[jmx://localhost:12000/MyCacheManager/namedCache]> encoding --list
memcached
hotrod
none
rest
[jmx://localhost:12000/MyCacheManager/namedCache]> encoding hotrod
```

Report a bug

23.6.10. The end Command

The **end** command ends a running batch initiated using the `start` command. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> start
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> end
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
a
```

Report a bug

23.6.11. The evict Command

The `evict` command evicts an an entry associated with a specific key from the cache. An example of it usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> evict a
```

Report a bug

23.6.12. The get Command

The `get` command shows the value associated with a specified key. For primitive types and Strings, the `get` command prints the default representation. For other objects, a JSON representation of the object is printed. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
a
```

Report a bug

23.6.13. The info Command

The **info** command displaysthe configuration of a selected cache or container. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> info
GlobalConfiguration{asyncListenerExecutor=ExecutorFactoryConfiguration{fac
tory=org.infinispan.executors.DefaultExecutorFactory@98add58},
asyncTransportExecutor=ExecutorFactoryConfiguration{factory=org.infinispan
.executors.DefaultExecutorFactory@7bc9c14c},
evictionScheduledExecutor=ScheduledExecutorFactoryConfiguration{factory=or
g.infinispan.executors.DefaultScheduledExecutorFactory@7ab1a411},
replicationQueueScheduledExecutor=ScheduledExecutorFactoryConfiguration{fa
ctory=org.infinispan.executors.DefaultScheduledExecutorFactory@248a9705},
globalJmxStatistics=GlobalJmxStatisticsConfiguration{allowDuplicateDomains
=true, enabled=true, jmxDomain='jboss.infinispan',
mBeanServerLookup=org.jboss.as.clustering.infinispan.MBeanServerProvider@6
c0dc01, cacheManagerName='local', properties={}},
transport=TransportConfiguration{clusterName='ISPN', machineId='null',
rackId='null', siteId='null', strictPeerToPeer=false,
distributedSyncTimeout=240000, transport=null, nodeName='null',
properties={}},
serialization=SerializationConfiguration{advancedExternalizers=
{1100=org.infinispan.server.core.CacheValue$Externalizer@5fabc91d,
1101=org.infinispan.server.memcached.MemcachedValue$Externalizer@720bffd,
1104=org.infinispan.server.hotrod.ServerAddress$Externalizer@771c7eb2},
marshaller=org.infinispan.marshall.VersionAwareMarshaller@6fc21535,
version=52,
classResolver=org.jboss.marshalling.ModularClassResolver@2efe83e5},
shutdown=ShutdownConfiguration{hookBehavior=DONT_REGISTER}, modules={},
site=SiteConfiguration{localSite='null'}}
```

Report a bug

23.6.14. The locate Command

The **locate** command displays the physical location of a specified entry in a distributed cluster. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> locate a
[host/node1,host/node2]
```

Report a bug

23.6.15. The put Command

The **put** command inserts an entry into the cache. If a mapping exists for a key, the **put** command overwrites the old value. The CLI allows control over the type of data used to store the key and value. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b 100
[jmx://localhost:12000/MyCacheManager/namedCache]> put c 4139l
[jmx://localhost:12000/MyCacheManager/namedCache]> put d true
[jmx://localhost:12000/MyCacheManager/namedCache]> put e {
"package.MyClass": {"i": 5, "x": null, "b": true } }
```

Optionally, the **put** can specify a life span and maximum idle time value as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a expires 10s
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a expires 10m
maxidle 1m
```

Report a bug

23.6.16. The replace Command

The **replace** command replaces an existing entry in the cache with a specified new value. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> replace a b
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
b
[jmx://localhost:12000/MyCacheManager/namedCache]> replace a b c
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
c
[jmx://localhost:12000/MyCacheManager/namedCache]> replace a b d
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
c
```

Report a bug

23.6.17. The rollback Command

The **rollback** command rolls back any changes made by an ongoing transaction. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> begin
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> rollback
```

Report a bug

23.6.18. The site Command

The **site** command performs administration tasks related to cross-datacentre replication. This command also retrieves information about the status of a site and toggles the status of a site. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> site --status NYC
online
[jmx://localhost:12000/MyCacheManager/namedCache]> site --offline NYC
ok
[jmx://localhost:12000/MyCacheManager/namedCache]> site --status NYC
offline
[jmx://localhost:12000/MyCacheManager/namedCache]> site --online NYC
```

Report a bug

23.6.19. The start Command

The **start** command initiates a batch of operations. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> start
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> end
```

Report a bug

23.6.20. The stats Command

The **stats** command displays statistics for the cache. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> stats
Statistics: {
  averageWriteTime: 143
  evictions: 10
  misses: 5
  hitRatio: 1.0
  readWriteRatio: 10.0
  removeMisses: 0
  timeSinceReset: 2123
  statisticsEnabled: true
  stores: 100
  elapsedTime: 93
  averageReadTime: 14
  removeHits: 0
  numberOfEntries: 100
  hits: 1000
}
LockManager: {
  concurrencyLevel: 1000
  numberOfLocksAvailable: 0
  numberOfLocksHeld: 0
}
```

Report a bug

23.6.21. The upgrade Command

The **upgrade** command implements the rolling upgrade procedure. For details about rolling upgrades, refer to Section 13.1, “About Rolling Upgrades”

An example of the **upgrade** command's use is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> upgrade --
synchronize=hotrod --all
[jmx://localhost:12000/MyCacheManager/namedCache]> upgrade --
disconnectsource=hotrod --all
```

Report a bug

23.6.22. The version Command

The **version** command displays version information for the CLI client and server. An example of its usage is as follows:

```
[jmx://localhost:12000/MyCacheManager/namedCache]> version
Client Version 5.2.1.Final
Server Version 5.2.1.Final
```

Report a bug

APPENDIX A. REVISION HISTORY

Revision 6.1.0-23.400 **2013-10-31** **Rüdiger Landmann**

Rebuild with publican 4.0.0

Revision 6.1.0-23 **Wed Oct 09 2013** **Misha Husnain Ali**

BZ-1015361: Updated information for REST, Memcached and Hot Rod endpoints.