



Red Hat JBoss Data Grid 6.3

Developer Guide

For use with Red Hat JBoss Data Grid 6.3.2

Red Hat JBoss Data Grid 6.3 Developer Guide

For use with Red Hat JBoss Data Grid 6.3.2

Misha Husnain Ali

Red Hat Engineering Content Services

mhusnain@redhat.com

Gemma Sheldon

Red Hat Engineering Content Services

gsheldon@redhat.com

Rakesh Ghatvisave

Red Hat Engineering Content Services

rghatvis@redhat.com

Legal Notice

Copyright © 2015 Red Hat, Inc.

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

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

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

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

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

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

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

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

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

All other trademarks are the property of their respective owners.

Abstract

An advanced guide intended for developers using Red Hat JBoss Data Grid 6.3.2

Table of Contents

PART I. PROGRAMMABLE APIS	4
CHAPTER 1. THE CACHE API	5
1.1. USING THE CONFIGURATIONBUILDER API TO CONFIGURE THE CACHE API	5
1.2. PER-INVOCATION FLAGS	6
1.3. THE ADVANCEDCACHE INTERFACE	7
CHAPTER 2. THE BATCHING API	13
2.1. ABOUT JAVA TRANSACTION API	13
2.2. BATCHING AND THE JAVA TRANSACTION API (JTA)	13
2.3. USING THE BATCHING API	14
CHAPTER 3. THE GROUPING API	17
3.1. GROUPING API OPERATIONS	17
3.2. GROUPING API USE CASE	17
3.3. CONFIGURE THE GROUPING API	18
CHAPTER 4. THE PERSISTENCE SPI	21
4.1. PERSISTENCE SPI BENEFITS	21
4.2. PROGRAMMATICALLY CONFIGURE THE PERSISTENCE SPI	21
CHAPTER 5. THE CONFIGURATIONBUILDER API	23
5.1. USING THE CONFIGURATIONBUILDER API	23
CHAPTER 6. THE EXTERNALIZABLE API	30
6.1. CUSTOMIZE EXTERNALIZERS	30
6.2. ANNOTATING OBJECTS FOR MARSHALLING USING @SERIALIZEWITH	30
6.3. USING AN ADVANCED EXTERNALIZER	31
6.4. CUSTOM EXTERNALIZER ID VALUES	35
CHAPTER 7. THE NOTIFICATION/LISTENER API	38
7.1. LISTENER EXAMPLE	38
7.2. CACHE ENTRY MODIFIED LISTENER CONFIGURATION	38
7.3. LISTENER NOTIFICATIONS	38
7.4. NOTIFYINGFUTURES	39
PART II. SECURING DATA IN RED HAT JBOSS DATA GRID	41
CHAPTER 8. RED HAT JBOSS DATA GRID SECURITY: AUTHORIZATION AND AUTHENTICATION	42
8.1. RED HAT JBOSS DATA GRID SECURITY: AUTHORIZATION AND AUTHENTICATION	42
8.2. PERMISSIONS	42
8.3. ROLE MAPPING	44
8.4. CONFIGURING AUTHENTICATION AND ROLE MAPPING USING JBOSS EAP LOGIN MODULES	45
8.5. CONFIGURING RED HAT JBOSS DATA GRID FOR AUTHORIZATION	47
8.6. DATA SECURITY FOR LIBRARY MODE	48
8.7. DATA SECURITY FOR REMOTE CLIENT SERVER MODE	57
8.8. ACTIVE DIRECTORY AUTHENTICATION (NON-KERBEROS)	76
8.9. ACTIVE DIRECTORY AUTHENTICATION USING KERBEROS (GSSAPI)	76
8.10. THE SECURITY AUDIT LOGGER	77
CHAPTER 9. SECURITY FOR CLUSTER TRAFFIC	80
9.1. NODE AUTHENTICATION AND AUTHORIZATION (REMOTE CLIENT-SERVER MODE)	80
9.2. CONFIGURE NODE SECURITY IN LIBRARY MODE	83
9.3. JGROUPS ENCRYPT	87

PART III. ADVANCED FEATURES IN RED HAT JBOSS DATA GRID	91
CHAPTER 10. TRANSACTIONS	92
10.1. ABOUT JAVA TRANSACTION API	92
10.2. TRANSACTIONS SPANNING MULTIPLE CACHE INSTANCES	92
10.3. THE TRANSACTION MANAGER	92
10.4. ABOUT JTA TRANSACTION MANAGER LOOKUP CLASSES	93
CHAPTER 11. MARSHALLING	94
11.1. ABOUT MARSHALLING FRAMEWORK	94
11.2. SUPPORT FOR NON-SERIALIZABLE OBJECTS	94
11.3. HOT ROD AND MARSHALLING	94
11.4. CONFIGURING THE MARSHALLER USING THE REMOTECACHEMANAGER	95
11.5. TROUBLESHOOTING	96
CHAPTER 12. LISTENERS AND NOTIFICATIONS	102
12.1. THE NOTIFICATION/LISTENER API	102
12.2. LISTENER NOTIFICATIONS	102
12.3. MODIFYING CACHE ENTRIES	103
12.4. NOTIFYINGFUTURES	104
CHAPTER 13. THE INFINISPAN CDI MODULE	105
13.1. USING INFINISPAN CDI	105
13.2. USING THE INFINISPAN CDI MODULE	105
CHAPTER 14. MAPREDUCE	117
14.1. THE MAPREDUCE API	118
14.2. MAPREDUCETASK DISTRIBUTED EXECUTION	121
14.3. MAP REDUCE EXAMPLE	124
CHAPTER 15. DISTRIBUTED EXECUTION	127
15.1. DISTRIBUTEDCALLABLE API	127
15.2. CALLABLE AND CDI	128
15.3. DISTRIBUTED TASK FAILOVER	128
15.4. DISTRIBUTED TASK EXECUTION POLICY	129
15.5. DISTRIBUTED EXECUTION EXAMPLE	130
CHAPTER 16. DATA INTEROPERABILITY	133
16.1. INTEROPERABILITY BETWEEN LIBRARY AND REMOTE CLIENT-SERVER ENDPOINTS	133
16.2. USING COMPATIBILITY MODE	133
16.3. PROTOCOL INTEROPERABILITY	133
APPENDIX A. REVISION HISTORY	137

PART I. PROGRAMMABLE APIS

Red Hat JBoss Data Grid provides the following programmable APIs:

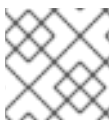
- Cache
- Batching
- Grouping
- Persistence (formerly CacheStore)
- ConfigurationBuilder
- Externalizable
- Notification (also known as the Listener API because it deals with Notifications and Listeners)

[Report a bug](#)

CHAPTER 1. THE CACHE API

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

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



NOTE

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

[Report a bug](#)

1.1. USING THE CONFIGURATIONBUILDER API TO CONFIGURE THE CACHE API

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

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

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

Example 1.1. Programmatic Cache Configuration

```
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).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`).

2. `String newCacheName = "repl";`

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

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

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

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

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

NOTE

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

To avoid this, configure global configuration as follows:

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

[Report a bug](#)

1.2. PER-INVOCATION FLAGS

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

[Report a bug](#)

1.2.1. Per-Invocation Flag Functions

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

- The `ZERO_LOCK_ACQUISITION_TIMEOUT` flag: JBoss Data Grid uses an almost zero lock acquisition time when loading data from an external source into a cache.
- The `FAIL_SILENTLY` flag: If the locks cannot be acquired, JBoss Data Grid fails silently without throwing any lock acquisition exceptions.

- The **FORCE_ASYNC** flag: If clustered, the cache replicates asynchronously, irrespective of the cache mode set. As a result, a response from other nodes is not required.

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

[Report a bug](#)

1.2.2. Configure Per-Invocation Flags

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

Example 1.2. Configuring Per-Invocation Flags

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



NOTE

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

[Report a bug](#)

1.2.3. Per-Invocation Flags Example

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

Example 1.3. Using the *IGNORE_RETURN_VALUES* Flag

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

[Report a bug](#)

1.3. THE ADVANCEDCACHE INTERFACE

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

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

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

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

[Report a bug](#)

1.3.1. Flag Usage with the AdvancedCache Interface

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

Example 1.4. Applying Flags to a Cache Invocation

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

[Report a bug](#)

1.3.2. Custom Interceptors and the AdvancedCache Interface

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

[Report a bug](#)

1.3.3. Custom Interceptors

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

[Report a bug](#)

1.3.3.1. Custom Interceptor Design

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

- A custom interceptor must extend the **CommandInterceptor**.
- A custom interceptor must declare a public, empty constructor to allow for instantiation.

- A custom interceptor must have JavaBean style setters defined for any property that is defined through the **property** element.

[Report a bug](#)

1.3.3.2. Adding Custom Interceptors Declaratively

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

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

Procedure 1.1. Adding Custom Interceptors

1. Define Custom Interceptors

All custom interceptors must extend `org.jboss.cache.interceptors.base.CommandInterceptor`. Use the `customInterceptors` method to add custom interceptors to the cache:

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

2. Define the Position of the New Custom Interceptor

Interceptors must have a defined position. Valid options are:

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

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

◦ Define Interceptor Properties

Define specific interceptor properties.

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

3. Apply Other Custom Interceptors

In this example, the next custom interceptor is called `CustomInterceptor2`.

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

```

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

```

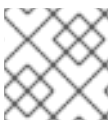
4. Define the `index`, `before`, and `after` Attributes.

- The `index` identifies the position of this interceptor in the chain, with 0 being the first position. This attribute is mutually exclusive with `position`, `before`, and `after`.
- The `after` method places the new interceptor directly after the instance of the named interceptor specified via its fully qualified class name. This attribute is mutually exclusive with `position`, `before`, and `index`.
- The `before` method places the new interceptor directly before the instance of the named interceptor specified via its fully qualified class name. This attribute is mutually exclusive with `position`, `after`, and `index`.

```

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

```



NOTE

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

[Report a bug](#)

1.3.3.3. Adding Custom Interceptors Programmatically

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

Example 1.5. Obtain a Reference to the AdvancedCache

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

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

Example 1.6. Add the Interceptor

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

[Report a bug](#)

1.3.4. Other Management Tools and Operations

Managing Red Hat JBoss Data Grid instances requires exposing significant amounts of relevant statistical information. This information allows administrators to get a clear view of each JBoss Data Grid node's state. A single installation can comprise of tens or hundreds of JBoss Data Grid nodes and it is important to provide this information in a clear and concise manner.

JBoss Operations Network is one example of a tool that provides runtime visibility. Other tools, such as JConsole can be used where JMX is enabled.

[Report a bug](#)

1.3.4.1. Accessing Data via URLs

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

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

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

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



NOTE

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

[Report a bug](#)

1.3.4.2. Limitations of Map Methods

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

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

Performance Concerns

From Red Hat JBoss Data Grid 6.3 onwards, the map methods `size()`, `values()`, `keySet()`, and `entrySet()` include entries in the cache loader by default whereas previously these methods only included the local data container. The underlying cache loader directly affects the performance of these commands. As an example, when using a database, these methods run a complete scan of the table where data is stored which can result in slower processing. Use `Cache.getAdvancedCache().withFlags(Flag.SKIP_CACHE_LOAD).values()` to maintain the old behavior and not loading from the cache loader which would avoid the slower performance.

[Report a bug](#)

CHAPTER 2. THE BATCHING API

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



NOTE

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

[Report a bug](#)

2.1. ABOUT JAVA TRANSACTION API

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

JBoss Data Grid does the following for each cache operation:

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

[Report a bug](#)

2.2. BATCHING AND THE JAVA TRANSACTION API (JTA)

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

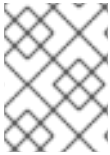
1. Locks acquired during an invocation are retained until the transaction commits or rolls back.
2. All changes are replicated in a batch on all nodes in the cluster as part of the transaction commit process. Ensuring that multiple changes occur within the single transaction, the replication traffic remains lower and improves performance.
3. When using synchronous replication or invalidation, a replication or invalidation failure causes the transaction to roll back.
4. When a cache is transactional and a cache loader is present, the cache loader is not enlisted in the cache's transaction. This results in potential inconsistencies at the cache loader level when the transaction applies the in-memory state but (partially) fails to apply the changes to the store.
5. All configurations related to a transaction apply for batching as well.

Example 2.1. Configuring a Transaction that Applies for Batching

```
<transaction syncRollbackPhase="false"  
    syncCommitPhase="false"  
    useEagerLocking="true"
```

```
eagerLockSingleNode="true" />
```

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



NOTE

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

[Report a bug](#)

2.3. USING THE BATCHING API

2.3.1. Enable the Batching API

Red Hat JBoss Data Grid's Batching API uses the JBoss Enterprise Application Platform syntax to enable invocation batching in your cache configuration.

Example 2.2. Enable Invocation Batching

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

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

[Report a bug](#)

2.3.2. Configure the Batching API

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

XML Configuration

To configure the Batching API in the XML file:

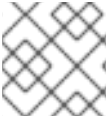
```
<invocationBatching enabled="true" />
```

Programmatic Configuration

To configure the Batching API programmatically use:

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

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

**NOTE**

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

[Report a bug](#)

2.3.3. Use the Batching API

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

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

Example 2.3. Without Using Batch

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

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

Example 2.4. Using Batch

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

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

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

[Report a bug](#)

2.3.4. Batching API Usage Example

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

Example 2.5. Batching API Usage Example

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

within JBoss Data Grid and the other in a database, distributed transactions are required for the transaction.

[Report a bug](#)

CHAPTER 3. THE GROUPING API

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

[Report a bug](#)

3.1. GROUPING API OPERATIONS

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

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

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

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

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

[Report a bug](#)

3.2. GROUPING API USE CASE

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

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

Example 3.1. Grouping API Example

Acme, Inc. is a home appliance company with over one hundred offices worldwide. Some offices house employees from various departments, while certain locations are occupied exclusively by the employees of one or two departments. The Human Resources (HR) department has employees in Bangkok, London, Chicago, Nice and Venice.

Acme, Inc. uses Red Hat JBoss Data Grid's Grouping API to ensure that all the employee records for the HR department are moved to a single node (Node AB) in the cache. As a result, when attempting to retrieve a record for a HR employee, the `DistributedExecutor` only checks node AB and quickly and easily retrieves the required employee records.

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

[Report a bug](#)

3.3. CONFIGURE THE GROUPING API

Use the following steps to configure the Grouping API:

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

[Report a bug](#)

3.3.1. Enable Groups

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

Example 3.2. Declaratively Enable Groups

Use the following configuration to enable groups using XML:

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

Example 3.3. Programmatically Enable Groups

Use the following to enable groups programmatically:

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

[Report a bug](#)

3.3.2. Specify an Intrinsic Group

Use an intrinsic group with the Grouping API if:

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

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

Example 3.4. Specifying an Intrinsic Group Example

```
class User {
    ...
    String office;
    ...

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

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

[Report a bug](#)

3.3.3. Specify an Extrinsic Group

Specify an extrinsic group for the Grouping API if:

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

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

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

Example 3.5. Specifying an Extrinsic Group Example

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

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

    // A pattern that can extract from a "kX" (e.g. k1, k2) style key
    // The pattern requires a String key, of length 2, where the first
```

```

character is
    // "k" and the second character is a digit. We take that digit, and
    // perform
    // modular arithmetic on it to assign it to group "1" or group "2".

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

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

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

```

[Report a bug](#)

3.3.4. Register Groupers

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

Example 3.6. Declaratively Register a Grouper

```

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

```

Example 3.7. Programmatically Register a Grouper

```

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

```

[Report a bug](#)

CHAPTER 4. THE PERSISTENCE SPI

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

Persistent external storage provides several benefits:

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

[Report a bug](#)

4.1. PERSISTENCE SPI BENEFITS

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

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


[Report a bug](#)

4.2. PROGRAMMATICALLY CONFIGURE THE PERSISTENCE SPI

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

Example 4.1. Configure the Single File Store via the Persistence SPI

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
    .passivation(false)
    .addSingleFileStore()
        .preload(true)
        .shared(false)
        .fetchPersistentState(true)
        .ignoreModifications(false)
```



```
.purgeOnStartup(false)
.location(System.getProperty("java.io.tmpdir"))
.async()
  .enabled(true)
  .threadPoolSize(5)
.singleton()
  .enabled(true)
  .pushStateWhenCoordinator(true)
  .pushStateTimeout(20000);
```

**NOTE**

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

[Report a bug](#)

CHAPTER 5. THE CONFIGURATIONBUILDER API

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

The ConfigurationBuilder API is designed to assist with:

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

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

[Report a bug](#)

5.1. USING THE CONFIGURATIONBUILDER API

5.1.1. Programmatically Create a CacheManager and Replicated Cache

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

Example 5.1. Configure the CacheManager Programmatically

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

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

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

Procedure 5.1. Steps for Programmatic Configuration in JBoss Data Grid

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

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

2. Create a new synchronously replicated cache programmatically.
 - a. Create a new configuration object instance using the ConfigurationBuilder helper object:

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

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

- b. Set the cache mode to synchronous replication:

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

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

- c. Define or register the configuration with a manager:

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

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

- d.

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

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



NOTE

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

[Report a bug](#)

5.1.2. Create a Customized Cache Using the Default Named Cache

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

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

Example 5.2. Configuring Customized Default Cache

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

```

Configuration dcc = cacheManager.getDefaultCacheConfiguration();
Configuration c = new ConfigurationBuilder().read(dcc).clustering()
    .cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000L).enable()
    .build();
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);

```

An explanation of the provided configuration is as follows:

Procedure 5.2. Customize the Default Cache

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

```

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

```

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

```

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

```

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

```

manager.defineConfiguration(newCacheName, c);

```

4. Get default cache with custom configuration changes:

```

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

```

[Report a bug](#)

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

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

The difference in approach is due to taking a named cache called `replicatedCache` as the base instead of the default cache.

Example 5.3. Creating a Customized Cache Using a Non-Default Named Cache

```

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

```

```

.cacheMode(CacheMode.DIST_SYNC).l1().lifespan(60000L).enable()
.build();
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);

```

An explanation of the provided example is as follows:

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

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

```

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

```

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

```

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

```

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

```

manager.defineConfiguration(newCacheName, c);

```

4. Get a default cache with custom configuration changes:

```

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

```

[Report a bug](#)

5.1.4. Using the Configuration Builder to Create Caches Programmatically

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

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

[Report a bug](#)

5.1.5. Global Configuration Examples

5.1.5.1. Globally Configure the Transport Layer

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

Example 5.4. Configuring the Transport Layer

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

[Report a bug](#)

5.1.5.2. Globally Configure the Cache Manager Name

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

Example 5.5. Configuring the Cache Manager Name

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

[Report a bug](#)

5.1.5.3. Globally Customize Thread Pool Executors

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

Example 5.6. Customize Thread Pool Executors

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

[Report a bug](#)

5.1.6. Cache Level Configuration Examples

5.1.6.1. Cache Level Configuration for the Cluster Mode

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

Example 5.7. Configure Cluster Mode at Cache Level

```

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

```

[Report a bug](#)**5.1.6.2. Cache Level Eviction and Expiration Configuration**

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

Example 5.8. Configuring Expiration and Eviction at the Cache Level

```

Configuration config = new ConfigurationBuilder()
    .eviction()

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

```

[Report a bug](#)**5.1.6.3. Cache Level Configuration for JTA Transactions**

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

Example 5.9. Configuring JTA Transactions at Cache Level

```

Configuration config = new ConfigurationBuilder()
    .locking()

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

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

```


[Report a bug](#)

5.1.6.4. Cache Level Configuration Using Chained Persistent Stores

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

Example 5.10. Configuring Chained Persistent Stores at Cache Level

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

[Report a bug](#)

5.1.6.5. Cache Level Configuration for Advanced Externalizers

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

Example 5.11. Configuring Advanced Externalizers at Cache Level

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

[Report a bug](#)

CHAPTER 6. THE EXTERNALIZABLE API

An `Externalizer` is a class that can:

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

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

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

[Report a bug](#)

6.1. CUSTOMIZE EXTERNALIZERS

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

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

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

[Report a bug](#)

6.2. ANNOTATING OBJECTS FOR MARSHALLING USING @SERIALIZEWITH

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

Example 6.1. Using the @SerializeWith Annotation

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

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

    final String name;
    final int age;

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

```

    }

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

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

```

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

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

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

Advanced Externalizers are available for users affected by these disadvantages.



NOTE

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

[Report a bug](#)

6.3. USING AN ADVANCED EXTERNALIZER

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

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

[Report a bug](#)

6.3.1. Implement the Methods

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

Example 6.2. Define and Implement the Methods

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

public class Person {

    final String name;
    final int age;

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

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

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

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

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



NOTE

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

[Report a bug](#)

6.3.2. Link Externalizers with Marshaller Classes

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

The following is a sample implementation:

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

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

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

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

[Report a bug](#)

6.3.3. Register the Advanced Externalizer (Declaratively)

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

Procedure 6.1. Register the Advanced Externalizer

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

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

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

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

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

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

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

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

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

[Report a bug](#)

6.3.4. Register the Advanced Externalizer (Programmatically)

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

Example 6.3. Registering the Advanced Externalizer Programmatically

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

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

[Report a bug](#)

6.3.5. Register Multiple Externalizers

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

Example 6.4. Registering Multiple Externalizers

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

[Report a bug](#)

6.4. CUSTOM EXTERNALIZER ID VALUES

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

Table 6.1. Reserved Externalizer ID Ranges

ID Range	Reserved For
1000-1099	The Infinispan Tree Module
1100-1199	Red Hat JBoss Data Grid Server modules
1200-1299	Hibernate Infinispan Second Level Cache
1300-1399	JBoss Data Grid Lucene Directory
1400-1499	Hibernate OGM
1500-1599	Hibernate Search
1600-1699	Infinispan Query Module
1700-1799	Infinispan Remote Query Module

[Report a bug](#)

6.4.1. Customize the Externalizer ID (Declaratively)

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

Procedure 6.2. Customizing the Externalizer ID (Declaratively)

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

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

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

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

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

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

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

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

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

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

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

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

[Report a bug](#)

6.4.2. Customize the Externalizer ID (Programmatically)

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

Example 6.5. Assign an ID to the Externalizer

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

Replace the *\$ID* with the desired ID.

[Report a bug](#)

CHAPTER 7. THE NOTIFICATION/LISTENER API

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

[Report a bug](#)

7.1. LISTENER EXAMPLE

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

Example 7.1. Configuring a Listener

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

[Report a bug](#)

7.2. CACHE ENTRY MODIFIED LISTENER CONFIGURATION

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

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

[Report a bug](#)

7.3. LISTENER NOTIFICATIONS

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

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

[Report a bug](#)

7.3.1. About Cache-level Notifications

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

[Report a bug](#)

7.3.2. Cache Manager-level Notifications

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

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

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

[Report a bug](#)

7.3.3. About Synchronous and Asynchronous Notifications

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

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

Annotate listeners using the following:

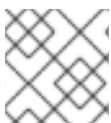
```
@Listener (sync = false)
public class MyAsyncListener { .... }
```

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

[Report a bug](#)

7.4. NOTIFYINGFUTURES

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



NOTE

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

[Report a bug](#)

7.4.1. NotifyingFutures Example

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

Example 7.2. Configuring `NotifyingFutures`

```
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. SECURING DATA IN RED HAT JBOSS DATA GRID

In Red Hat JBoss Data Grid, data security can be implemented in the following ways:

Role-based Access Control

JBoss Data Grid features role-based access control for operations on designated secured caches. Roles can be assigned to users who access your application, with roles mapped to permissions for cache and cache-manager operations. Only authenticated users are able to perform the operations that are authorized for their role.

In Library mode, data is secured via role-based access control for CacheManagers and Caches, with authentication delegated to the container or application. In Remote Client-Server mode, JBoss Data Grid is secured by passing identity tokens from the Hot Rod client to the server, and role-based access control of Caches and CacheManagers.

Node Authentication and Authorization

Node-level security requires new nodes or merging partitions to authenticate before joining a cluster. Only authenticated nodes that are authorized to join the cluster are permitted to do so. This provides data protection by preventing unauthorized servers from storing your data.

Encrypted Communications Within the Cluster

JBoss Data Grid increases data security by supporting encrypted communications between the nodes in a cluster by using a user-specified cryptography algorithm, as supported by Java Cryptography Architecture (JCA).

JBoss Data Grid also provides audit logging for operations, and the ability to encrypt communication between the Hot Rod Client and Server using Transport Layer Security (TLS/SSL).

[Report a bug](#)

CHAPTER 8. RED HAT JBOSS DATA GRID SECURITY: AUTHORIZATION AND AUTHENTICATION

8.1. RED HAT JBOSS DATA GRID SECURITY: AUTHORIZATION AND AUTHENTICATION

Red Hat JBoss Data Grid is able to perform authorization on CacheManagers and Caches. JBoss Data Grid authorization is built on standard security features available in a JDK, such as JAAS and the SecurityManager.

If an application attempts to interact with a secured CacheManager and Cache, it must provide an identity which JBoss Data Grid's security layer can validate against a set of required roles and permissions. Once validated, the client is issued a token for subsequent operations. Where access is denied, an exception indicating a security violation is thrown.

When a cache has been configured for with authorization, retrieving it returns an instance of **SecureCache**. **SecureCache** is a simple wrapper around a cache, which checks whether the "current user" has the permissions required to perform an operation. The "current user" is a Subject associated with the **AccessControlContext**.

JBoss Data Grid maps Principals names to roles, which in turn, represent one or more permissions. The following diagram represents these relationships:

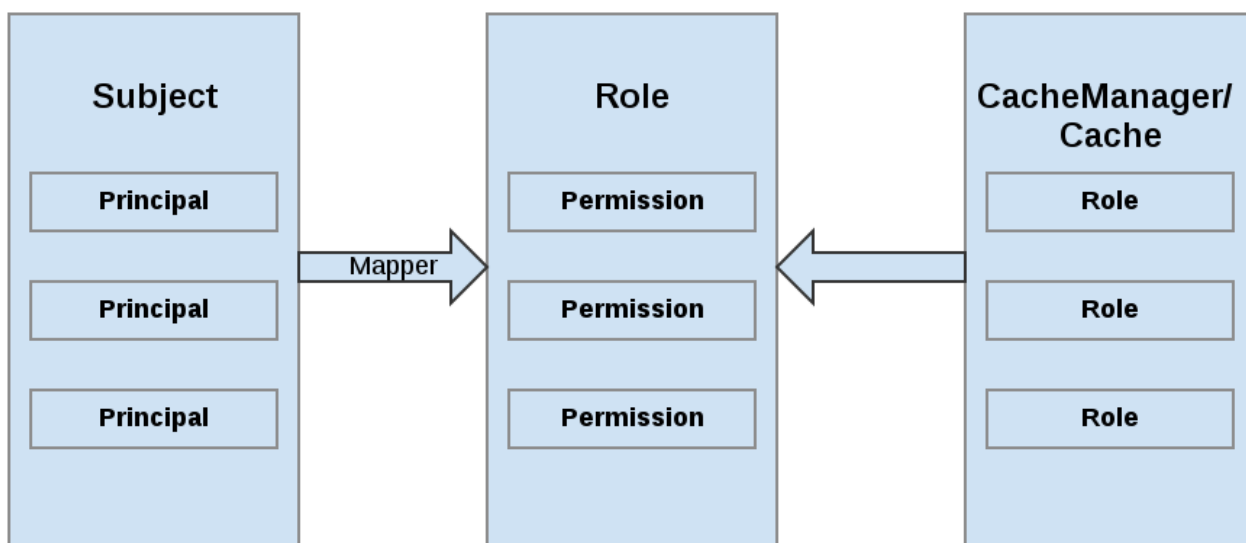


Figure 8.1. Roles and Permissions Mapping

[Report a bug](#)

8.2. PERMISSIONS

Access to a CacheManager or a Cache is controlled using a set of required permissions. Permissions control the type of action that is performed on the CacheManager or Cache, rather than the type of data being manipulated. Some of these permissions can apply to specifically name entities, such as a named cache. Different types of permissions are available depending on the entity.

Table 8.1. CacheManager Permissions

Permission	Function	Description
CONFIGURATION	defineConfiguration	Whether a new cache configuration can be defined.
LISTEN	addListener	Whether listeners can be registered against a cache manager.
LIFECYCLE	stop, start	Whether the cache manager can be stopped or started respectively.
ALL		A convenience permission which includes all of the above.

Table 8.2. Cache Permissions

Permission	Function	Description
READ	get, contains	Whether entries can be retrieved from the cache.
WRITE	put, putIfAbsent, replace, remove, evict	Whether data can be written/replaced/removed/evicted from the cache.
EXEC	distexec, mapreduce	Whether code execution can be run against the cache.
LISTEN	addListener	Whether listeners can be registered against a cache.
BULK_READ	keySet, values, entrySet, query	Whether bulk retrieve operations can be executed.
BULK_WRITE	clear, putAll	Whether bulk write operations can be executed.
LIFECYCLE	start, stop	Whether a cache can be started / stopped.

Permission	Function	Description
ADMIN	getVersion, addInterceptor*, removeInterceptor, getInterceptorChain, getEvictionManager, getComponentRegistry, getDistributionManager, getAuthorizationManager, evict, getRpcManager, getCacheConfiguration, getCacheManager, getInvocationContextContainer, setAvailability, getDataContainer, getStats, getXAResource	Whether access to the underlying components/internal structures is allowed.
ALL		A convenience permission which includes all of the above.
ALL_READ		Combines READ and BULK_READ.
ALL_WRITE		Combines WRITE and BULK_WRITE.

**NOTE**

Some permissions may need to be combined with others in order to be useful. For example, EXEC with READ or with WRITE.

[Report a bug](#)

8.3. ROLE MAPPING

In order to convert the Principals in a Subject into a set of roles used for authorization, a **PrincipalRoleMapper** must be specified in the global configuration. Red Hat JBoss Data Grid ships with three mappers, and also allows you to provide a custom mapper.

Table 8.3. Mappers

Mapper Name	Java	XML	Description
IdentityRoleMapper	org.infinispan.security.impl.IdentityRoleMapper	<identity-role-mapper />	Uses the Principal name as the role name.

Mapper Name	Java	XML	Description
CommonNameRoleMapper	org.infinispan.security.impl.CommonRoleMapper	<common-name-role-mapper />	If the Principal name is a Distinguished Name (DN), this mapper extracts the Common Name (CN) and uses it as a role name. For example the DN cn=managers, ou=people, dc=example, dc=com will be mapped to the role managers .
ClusterRoleMapper	org.infinispan.security.impl.ClusterRoleMapper	<cluster-role-mapper />	Uses the ClusterRegistry to store principal to role mappings. This allows the use of the CLI's GRANT and DENY commands to add/remove roles to a Principal.
Custom Role Mapper		<custom-role-mapper class="a.b.c" />	Supply the fully-qualified class name of an implementation of org.infinispan.security.impl.PrincipalRoleMapper

[Report a bug](#)

8.4. CONFIGURING AUTHENTICATION AND ROLE MAPPING USING JBOSS EAP LOGIN MODULES

When using Red Hat JBoss EAP log in module for querying roles from LDAP, you must implement your own mapping of Principals to Roles, as JBoss EAP uses its own custom classes. The following example demonstrates how to map a principal obtained from JBoss EAP login module to a role. It maps user principal name to a role, performing a similar action to the **IdentityRoleMapper**:

Example 8.1. Mapping a Principal from JBoss EAP's Login Module

```
public class SimplePrincipalGroupRoleMapper implements
PrincipalRoleMapper {
    @Override
    public Set<String> principalToRoles(Principal principal) {
        if (principal instanceof SimpleGroup) {
            Enumeration<Principal> members = ((SimpleGroup)
principal).members();
            if (members.hasMoreElements()) {
```

```

        Set<String> roles = new HashSet<String>();
        while (members.hasMoreElements()) {
            Principal innerPrincipal = members.nextElement();
            if (innerPrincipal instanceof SimplePrincipal) {
                SimplePrincipal sp = (SimplePrincipal)
innerPrincipal;
                roles.add(sp.getName());
            }
        }
        return roles;
    }
}
return null;
}
}
}

```

Example 8.2. Example of JBoss EAP LDAP login module configuration

```

<security-domain name="ispn-secure" cache-type="default">
    <authentication>
        <login-module
code="org.jboss.security.auth.spi.LdapLoginModule" flag="required">
            <module-option
name="java.naming.factory.initial"
value="com.sun.jndi.ldap.LdapCtxFactory"/>
            <module-option name="java.naming.provider.url"
value="ldap://localhost:389"/>
            <module-option
name="java.naming.security.authentication" value="simple"/>
            <module-option name="principalDNPrefix"
value="uid"/>
            <module-option name="principalDNSuffix"
value=",ou=People,dc=infinispan,dc=org"/>
            <module-option name="rolesCtxDN"
value="ou=Roles,dc=infinispan,dc=org"/>
            <module-option name="uidAttributeID"
value="member"/>
            <module-option name="matchOnUserDN"
value="true"/>
            <module-option name="roleAttributeID"
value="cn"/>
            <module-option name="roleAttributeIsDN"
value="false"/>
            <module-option name="searchScope"
value="ONELEVEL_SCOPE"/>
        </login-module>
    </authentication>
</security-domain>

```

Example 8.3. Example of JBoss EAP Login Module Configuration

```

<security-domain name="krb-admin" cache-type="default">

```

```

        <authentication>
            <login-module code="Kerberos" flag="required">
                <module-option name="useKeyTab"
value="true"/>
                <module-option name="principal"
value="admin@INFINISPAN.ORG"/>
                <module-option name="keyTab"
value="\${basedir}/keytab/admin.keytab"/>
            </login-module>
        </authentication>
    </security-domain>

```

When using GSSAPI authentication, this would typically involve using LDAP for role mapping, with JBoss EAP server authenticating itself to the LDAP server via GSSAPI. For more information on how to configure this, see the *JBoss EAP Administration and Configuration Guide*



IMPORTANT

For information about how to configure JBoss EAP login modules, see the *JBoss EAP Administration and Configuration Guide* and see the *Red Hat Directory Server Administration Guide* how to configure LDAP server, and specify users and their role mapping.

[Report a bug](#)

8.5. CONFIGURING RED HAT JBOSS DATA GRID FOR AUTHORIZATION

Authorization is configured at two levels: the cache container (CacheManager), and at the single cache.

CacheManager

The following is an example configuration for authorization at the CacheManager level:

Example 8.4. CacheManager Authorization (Declarative Configuration)

```

<cache-container name="local" default-cache="default">
    <security>
        <authorization>
            <identity-role-mapper />
            <role name="admin" permissions="ALL"/>
            <role name="reader" permissions="READ"/>
            <role name="writer" permissions="WRITE"/>
            <role name="supervisor" permissions="ALL_READ ALL_WRITE"/>
        </authorization>
    </security>
</cache-container>

```

Each cache container determines:

- whether to use authorization.

- a class which will map principals to a set of roles.
- a set of named roles and the permissions they represent.

You can choose to use only a subset of the roles defined at the container level.

Roles

Roles for each cache can be defined as follows:

Example 8.5. Defining Roles

```
<local-cache name="secured">
  <security>
    <authorization roles="admin reader writer supervisor"/>
  </security>
</local-cache>
```

Programmatic CacheManager Authorization (Library Mode)

The following example shows how to set up the same authorization parameters for Library mode using programmatic configuration:

Example 8.6. CacheManager Authorization Programmatic Configuration

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
global
    .security()
    .authorization()
    .principalRoleMapper(new IdentityRoleMapper())
    .role("admin")
    .permission(CachePermission.ALL)
    .role("supervisor")
    .permission(CachePermission.EXEC)
    .permission(CachePermission.READ)
    .permission(CachePermission.WRITE)
    .role("reader")
    .permission(CachePermission.READ);
ConfigurationBuilder config = new ConfigurationBuilder();
config
    .security()
    .enable()
    .authorization()
    .role("admin")
    .role("supervisor")
    .role("reader");
```

[Report a bug](#)

8.6. DATA SECURITY FOR LIBRARY MODE

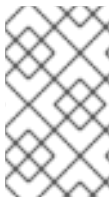
8.6.1. Subject and Principal Classes

To authorize access to resources, applications must first authenticate the request's source. The JAAS framework defines the term `subject` to represent a request's source. The `Subject` class is the central class in JAAS. A `Subject` represents information for a single entity, such as a person or service. It encompasses the entity's principals, public credentials, and private credentials. The JAAS APIs use the existing Java 2 `java.security.Principal` interface to represent a principal, which is a typed name.

During the authentication process, a subject is populated with associated identities, or principals. A subject may have many principals. For example, a person may have a name principal (John Doe), a social security number principal (123-45-6789), and a user name principal (johnd), all of which help distinguish the subject from other subjects. To retrieve the principals associated with a subject, two methods are available:

```
public Set getPrincipals() {...}
public Set getPrincipals(Class c) {...}
```

`getPrincipals()` returns all principals contained in the subject. `getPrincipals(Class c)` returns only those principals that are instances of class `c` or one of its subclasses. An empty set is returned if the subject has no matching principals.



NOTE

The `java.security.acl.Group` interface is a sub-interface of `java.security.Principal`, so an instance in the principals set may represent a logical grouping of other principals or groups of principals.

[Report a bug](#)

8.6.2. Obtaining a Subject

In order to use a secured cache in Library mode, you must obtain a `javax.security.auth.Subject`. The `Subject` represents information for a single cache entity, such as a person or a service.

Red Hat JBoss Data Grid allows a JAAS `Subject` to be obtained either by using your container's features, or by using a third-party library.

In JBoss containers, this can be done using the following:

```
Subject subject = SecurityContextAssociation.getSubject();
```

The `Subject` must be populated with a set of `Principals`, which represent the user and groups it belongs to in your security domain, for example, an LDAP or Active Directory.

The Java EE API allows retrieval of a container-set `Principal` through the following methods:

- Servlets: `ServletRequest.getUserPrincipal()`
- EJBs: `EJBContext.getCallerPrincipal()`
- MessageDrivenBeans: `MessageDrivenContext.getCallerPrincipal()`

The `mapper` is then used to identify the principals associated with the `Subject` and convert them into roles that correspond to those you have defined at the container level.

A **Principal** is only one of the components of a **Subject**, which is retrieved from the `java.security.AccessControlContext`. Either the container sets the **Subject** on the `AccessControlContext`, or the user must map the **Principal** to an appropriate **Subject** before wrapping the call to the JBoss Data Grid API using a `Security.doAs()` method.

Once a **Subject** has been obtained, the cache can be interacted with in the context of a `PrivilegedAction`.

Example 8.7. Obtaining a Subject

```
import org.infinispan.security.Security;

Security.doAs(subject, new PrivilegedExceptionAction<Void>() {
    public Void run() throws Exception {
        cache.put("key", "value");
    }
});
```

The `Security.doAs()` method is in place of the typical `Subject.doAs()` method. Unless the `AccessControlContext` must be modified for reasons specific to your application's security model, using `Security.doAs()` provides a performance advantage.

To obtain the current **Subject**, use `Security.getSubject()`, which will retrieve the **Subject** from either the JBoss Data Grid context, or from the `AccessControlContext`.

[Report a bug](#)

8.6.3. Subject Authentication

Subject Authentication requires a JAAS login. The login process consists of the following points:

1. An application instantiates a `LoginContext` and passes in the name of the login configuration and a `CallbackHandler` to populate the `Callback` objects, as required by the configuration `LoginModules`.
2. The `LoginContext` consults a `Configuration` to load all the `LoginModules` included in the named login configuration. If no such named configuration exists the `other` configuration is used as a default.
3. The application invokes the `LoginContext.login` method.
4. The login method invokes all the loaded `LoginModules`. As each `LoginModule` attempts to authenticate the subject, it invokes the `handle` method on the associated `CallbackHandler` to obtain the information required for the authentication process. The required information is passed to the `handle` method in the form of an array of `Callback` objects. Upon success, the `LoginModules` associate relevant principals and credentials with the subject.
5. The `LoginContext` returns the authentication status to the application. Success is represented by a return from the login method. Failure is represented through a `LoginException` being thrown by the login method.
6. If authentication succeeds, the application retrieves the authenticated subject using the `LoginContext.getSubject` method.

7. After the scope of the subject authentication is complete, all principals and related information associated with the subject by the `login` method can be removed by invoking the `LoginContext.logout` method.

The `LoginContext` class provides the basic methods for authenticating subjects and offers a way to develop an application that is independent of the underlying authentication technology. The `LoginContext` consults a `Configuration` to determine the authentication services configured for a particular application. `LoginModule` classes represent the authentication services. Therefore, you can plug different login modules into an application without changing the application itself. The following code shows the steps required by an application to authenticate a subject.

```

CallbackHandler handler = new MyHandler();
LoginContext lc = new LoginContext("some-config", handler);

try {
    lc.login();
    Subject subject = lc.getSubject();
} catch(LoginException e) {
    System.out.println("authentication failed");
    e.printStackTrace();
}

// Perform work as authenticated Subject
// ...

// Scope of work complete, logout to remove authentication info
try {
    lc.logout();
} catch(LoginException e) {
    System.out.println("logout failed");
    e.printStackTrace();
}

// A sample MyHandler class
class MyHandler
    implements CallbackHandler
{
    public void handle(Callback[] callbacks) throws
        IOException, UnsupportedCallbackException
    {
        for (int i = 0; i < callbacks.length; i++) {
            if (callbacks[i] instanceof NameCallback) {
                NameCallback nc = (NameCallback)callbacks[i];
                nc.setName(username);
            } else if (callbacks[i] instanceof PasswordCallback) {
                PasswordCallback pc = (PasswordCallback)callbacks[i];
                pc.setPassword(password);
            } else {
                throw new UnsupportedCallbackException(callbacks[i],
                    "Unrecognized
Callback");
            }
        }
    }
}

```

Developers integrate with an authentication technology by creating an implementation of the **LoginModule** interface. This allows an administrator to plug different authentication technologies into an application. You can chain together multiple **LoginModules** to allow for more than one authentication technology to participate in the authentication process. For example, one **LoginModule** may perform user name/password-based authentication, while another may interface to hardware devices such as smart card readers or biometric authenticators.

The life cycle of a **LoginModule** is driven by the **LoginContext** object against which the client creates and issues the login method. The process consists of two phases. The steps of the process are as follows:

- The **LoginContext** creates each configured **LoginModule** using its public no-arg constructor.
- Each **LoginModule** is initialized with a call to its initialize method. The **Subject** argument is guaranteed to be non-null. The signature of the initialize method is: **public void initialize(Subject subject, CallbackHandler callbackHandler, Map sharedState, Map options)**
- The **login** method is called to start the authentication process. For example, a method implementation might prompt the user for a user name and password and then verify the information against data stored in a naming service such as NIS or LDAP. Alternative implementations might interface to smart cards and biometric devices, or simply extract user information from the underlying operating system. The validation of user identity by each **LoginModule** is considered phase 1 of JAAS authentication. The signature of the **login** method is **boolean login() throws LoginException**. A **LoginException** indicates failure. A return value of true indicates that the method succeeded, whereas a return value of false indicates that the login module should be ignored.
- If the **LoginContext**'s overall authentication succeeds, **commit** is invoked on each **LoginModule**. If phase 1 succeeds for a **LoginModule**, then the commit method continues with phase 2 and associates the relevant principals, public credentials, and/or private credentials with the subject. If phase 1 fails for a **LoginModule**, then **commit** removes any previously stored authentication state, such as user names or passwords. The signature of the **commit** method is: **boolean commit() throws LoginException**. Failure to complete the commit phase is indicated by throwing a **LoginException**. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.
- If the **LoginContext**'s overall authentication fails, then the **abort** method is invoked on each **LoginModule**. The **abort** method removes or destroys any authentication state created by the login or initialize methods. The signature of the **abort** method is **boolean abort() throws LoginException**. Failure to complete the **abort** phase is indicated by throwing a **LoginException**. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.
- To remove the authentication state after a successful login, the application invokes **logout** on the **LoginContext**. This in turn results in a **logout** method invocation on each **LoginModule**. The **logout** method removes the principals and credentials originally associated with the subject during the **commit** operation. Credentials should be destroyed upon removal. The signature of the **logout** method is: **boolean logout() throws LoginException**. Failure to complete the logout process is indicated by throwing a **LoginException**. A return of true indicates that the method succeeded, whereas a return of false indicates that the login module should be ignored.

When a `LoginModule` must communicate with the user to obtain authentication information, it uses a `CallbackHandler` object. Applications implement the `CallbackHandler` interface and pass it to the `LoginContext`, which send the authentication information directly to the underlying login modules.

Login modules use the `CallbackHandler` both to gather input from users, such as a password or smart card PIN, and to supply information to users, such as status information. By allowing the application to specify the `CallbackHandler`, underlying `LoginModules` remain independent from the different ways applications interact with users. For example, a `CallbackHandler`'s implementation for a GUI application might display a window to solicit user input. On the other hand, a `CallbackHandler` implementation for a non-GUI environment, such as an application server, might simply obtain credential information by using an application server API. The `CallbackHandler` interface has one method to implement:

```
void handle(Callback[] callbacks)
    throws java.io.IOException,
           UnsupportedOperationException;
```

The `Callback` interface is the last authentication class we will look at. This is a tagging interface for which several default implementations are provided, including the `NameCallback` and `PasswordCallback` used in an earlier example. A `LoginModule` uses a `Callback` to request information required by the authentication mechanism. `LoginModules` pass an array of `Callbacks` directly to the `CallbackHandler` . `handle` method during the authentication's login phase. If a `callbackhandler` does not understand how to use a `Callback` object passed into the `handle` method, it throws an `UnsupportedCallbackException` to abort the login call.

[Report a bug](#)

8.6.4. Authorization Using a SecurityManager

In Red Hat JBoss Data Grid's Remote Client-Server mode, authorization is able to work without a `SecurityManager` for basic cache operations. In Library mode, a `SecurityManager` may also be used to perform some of the more complex tasks, such as `distexec`, `map/reduce`, and `query`.

In order to enforce access restrictions, enable the `SecurityManager` in your JVM using one of the following methods:

Command Line

```
java -Djava.security.manager ...
```

Programmatically

```
System.setSecurityManager(new SecurityManager());
```

Using the JDK's default implementation is not required, however an appropriate policy file must be supplied. The JBoss Data Grid distribution includes an example policy file, which demonstrates the permissions required by some of JBoss Data Grid's JAR files. These permissions must be integrated with those required by your application.

[Report a bug](#)

8.6.5. Security Manager in Java

8.6.5.1. About the Java Security Manager

Java Security Manager

The Java Security Manager is a class that manages the external boundary of the Java Virtual Machine (JVM) sandbox, controlling how code executing within the JVM can interact with resources outside the JVM. When the Java Security Manager is activated, the Java API checks with the security manager for approval before executing a wide range of potentially unsafe operations.

The Java Security Manager uses a security policy to determine whether a given action will be permitted or denied.

[Report a bug](#)

8.6.5.2. About Java Security Manager Policies

Security Policy

A set of defined permissions for different classes of code. The Java Security Manager compares actions requested by applications against the security policy. If an action is allowed by the policy, the Security Manager will permit that action to take place. If the action is not allowed by the policy, the Security Manager will deny that action. The security policy can define permissions based on the location of code, on the code's signature, or based on the subject's principals.

The Java Security Manager and the security policy used are configured using the Java Virtual Machine options `java.security.manager` and `java.security.policy`.

Basic Information

A security policy's entry consists of the following configuration elements, which are connected to the `policytool`:

CodeBase

The URL location (excluding the host and domain information) where the code originates from. This parameter is optional.

SignedBy

The alias used in the keystore to reference the signer whose private key was used to sign the code. This can be a single value or a comma-separated list of values. This parameter is optional. If omitted, presence or lack of a signature has no impact on the Java Security Manager.

Principals

A list of *principal_type/principal_name* pairs, which must be present within the executing thread's principal set. The Principals entry is optional. If it is omitted, it signifies that the principals of the executing thread will have no impact on the Java Security Manager.

Permissions

A permission is the access which is granted to the code. Many permissions are provided as part of the Java Enterprise Edition 6 (Java EE 6) specification. This document only covers additional permissions which are provided by JBoss EAP 6.



IMPORTANT

Refer to your container documentation on how to configure the security policy, as it may differ depending on the implementation.

[Report a bug](#)

8.6.5.3. Write a Java Security Manager Policy

Introduction

An application called `policytool` is included with most JDK and JRE distributions, for the purpose of creating and editing Java Security Manager security policies. Detailed information about `policytool` is linked from <http://docs.oracle.com/javase/6/docs/technotes/tools/>.

Procedure 8.1. Setup a new Java Security Manager Policy

1. **Start `policytool`.**

Start the `policytool` tool in one of the following ways.

- o **Red Hat Enterprise Linux**

From your GUI or a command prompt, run `/usr/bin/policytool`.

- o **Microsoft Windows Server**

Run `policytool.exe` from your Start menu or from the `bin\` of your Java installation. The location can vary.

2. **Create a policy.**

To create a policy, select **Add Policy Entry**. Add the parameters you need, then click **Done**.

3. **Edit an existing policy**

Select the policy from the list of existing policies, and select the **Edit Policy Entry** button. Edit the parameters as needed.

4. **Delete an existing policy.**

Select the policy from the list of existing policies, and select the **Remove Policy Entry** button.

[Report a bug](#)

8.6.5.4. Run Red Hat JBoss Data Grid Server Within the Java Security Manager

To specify a Java Security Manager policy, you need to edit the Java options passed to the domain or server instance during the bootstrap process. For this reason, you cannot pass the parameters as options to the `domain.sh` or `standalone.sh` scripts. The following procedure guides you through the steps of configuring your instance to run within a Java Security Manager policy.

Prerequisites

- Before you following this procedure, you need to write a security policy, using the `policytool` command which is included with your Java Development Kit (JDK). This procedure assumes that your policy is located at `JDG_HOME/bin/server.policy`. As an alternative, write the security policy using any text editor and manually save it as `JDG_HOME/bin/server.policy`

- The domain or standalone server must be completely stopped before you edit any configuration files.

Perform the following procedure for each physical host or instance in your domain, if you have domain members spread across multiple systems.

Procedure 8.2. Configure the Security Manager for JBoss Data Grid Server

1. Open the configuration file.

Open the configuration file for editing. This file is located in one of two places, depending on whether you use a managed domain or standalone server. This is not the executable file used to start the server or domain.

o Managed Domain

- For Linux: `JDG_HOME/bin/domain.conf`
- For Windows: `JDG_HOME\bin\domain.conf.bat`

o Standalone Server

- For Linux: `JDG_HOME/bin/standalone.conf`
- For Windows: `JDG_HOME\bin\standalone.conf.bat`

2. Add the Java options to the file.

To ensure the Java options are used, add them to the code block that begins with:

```
if [ "$JAVA_OPTS" = "x" ]; then
```

You can modify the `-Djava.security.policy` value to specify the exact location of your security policy. It should go onto one line only, with no line break. Using `==` when setting the `-Djava.security.policy` property specifies that the security manager will use *only* the specified policy file. Using `=` specifies that the security manager will use the specified policy *combined with* the policy set in the `policy.url` section of `JAVA_HOME/lib/security/java.security`.



IMPORTANT

JBoss Enterprise Application Platform releases from 6.2.2 onwards require that the system property `jboss.modules.policy-permissions` is set to `true`.

Example 8.8. domain.conf

```
JAVA_OPTS="$JAVA_OPTS -Djava.security.manager -
Djava.security.policy==$PWD/server.policy -
Djboss.home.dir=/path/to/JDG_HOME -Djboss.modules.policy-
permissions=true"
```

Example 8.9. domain.conf.bat

```
set "JAVA_OPTS=%JAVA_OPTS% -Djava.security.manager -
```

```
Djava.security.policy==\path\to\server.policy -
Djboss.home.dir=\path\to\JDG_HOME -Djboss.modules.policy-
permissions=true"
```

Example 8.10. standalone.conf

```
JAVA_OPTS="$JAVA_OPTS -Djava.security.manager -
Djava.security.policy==\$PWD/server.policy -
Djboss.home.dir=$JBOSS_HOME -Djboss.modules.policy-
permissions=true"
```

Example 8.11. standalone.conf.bat

```
set "JAVA_OPTS=%JAVA_OPTS% -Djava.security.manager -
Djava.security.policy==\path\to\server.policy -
Djboss.home.dir=%JBOSS_HOME% -Djboss.modules.policy-
permissions=true"
```

3. Start the domain or server.

Start the domain or server as normal.

[Report a bug](#)

8.7. DATA SECURITY FOR REMOTE CLIENT SERVER MODE

8.7.1. About Security Realms

A *security realm* is a series of mappings between users and passwords, and users and roles. Security realms are a mechanism for adding authentication and authorization to your EJB and Web applications. Red Hat JBoss Data Grid Server provides two security realms by default:

- **ManagementRealm** stores authentication information for the Management API, which provides the functionality for the Management CLI and web-based Management Console. It provides an authentication system for managing JBoss Data Grid Server itself. You could also use the **ManagementRealm** if your application needed to authenticate with the same business rules you use for the Management API.
- **ApplicationRealm** stores user, password, and role information for Web Applications and EJBs.

Each realm is stored in two files on the filesystem:

- **REALM-users.properties** stores usernames and hashed passwords.
- **REALM-roles.properties** stores user-to-role mappings.
- **mgmt-groups.properties** stores user-to-role mapping file for **ManagementRealm**.

The properties files are stored in the **domain/configuration/** and

`standalone/configuration/` directories. The files are written simultaneously by the `add-user . sh` or `add-user . bat` command. When you run the command, the first decision you make is which realm to add your new user to.

[Report a bug](#)

8.7.2. Add a New Security Realm

1. Run the Management CLI.

Start the `cli . sh` or `cli . bat` command and connect to the server.

2. Create the new security realm itself.

Run the following command to create a new security realm named `MyDomainRealm` on a domain controller or a standalone server.

```
/host=master/core-service=management/security-
realm=MyDomainRealm:add()
```

3. Create the references to the properties file which will store information about the new role.

Run the following command to create a pointer a file named `myfile . properties`, which will contain the properties pertaining to the new role.



NOTE

The newly-created properties file is not managed by the included `add-user . sh` and `add-user . bat` scripts. It must be managed externally.

```
/host=master/core-service=management/security-
realm=MyDomainRealm/authentication=properties:add(path=myfile.proper
ties)
```

Result

Your new security realm is created. When you add users and roles to this new realm, the information will be stored in a separate file from the default security realms. You can manage this new file using your own applications or procedures.

[Report a bug](#)

8.7.3. Add a User to a Security Realm

1. Run the `add-user . sh` or `add-user . bat` command.

Open a terminal and change directories to the `JDG_HOME/bin/` directory. If you run Red Hat Enterprise Linux or another UNIX-like operating system, run `add-user . sh`. If you run Microsoft Windows Server, run `add-user . bat`.

2. Choose whether to add a Management User or Application User.

For this procedure, type `b` to add an Application User.

3. Choose the realm the user will be added to.

By default, the only available realm is `ApplicationRealm`. If you have added a custom realm, you can type its name instead.

4. Type the username, password, and roles, when prompted.

Type the desired username, password, and optional roles when prompted. Verify your choice by typing **yes**, or type **no** to cancel the changes. The changes are written to each of the properties files for the security realm.

[Report a bug](#)

8.7.4. Configuring Security Realms Declaratively

In Remote Client-Server mode, a Hot Rod endpoint must specify a security realm.

The security realm declares an `<authentication>` and an `<authorization>` section.

Example 8.12. Configuring Security Realms Declaratively

```
<security-realms>
  <security-realm name="ManagementRealm">
    <authentication>
      <local default-user="$local" skip-group-
loading="true"/>
      <properties path="mgmt-users.properties" relative-
to="jboss.server.config.dir"/>
    </authentication>
    <authorization map-groups-to-roles="false">
      <properties path="mgmt-groups.properties"
relative-to="jboss.server.config.dir"/>
    </authorization>
  </security-realm>
  <security-realm name="ApplicationRealm">
    <authentication>
      <local default-user="$local" allowed-users="*"
skip-group-loading="true"/>
      <properties path="application-users.properties"
relative-to="jboss.server.config.dir"/>
    </authentication>
    <authorization>
      <properties path="application-roles.properties"
relative-to="jboss.server.config.dir"/>
    </authorization>
  </security-realm>
</security-realms>
```

The `<server-identities>` parameter can also be used to specify certificates.

[Report a bug](#)

8.7.5. Loading Roles from LDAP for Authorization (Remote Client-Server Mode)

An LDAP directory contains entries for user accounts and groups, cross referenced by attributes. Depending on the LDAP server configuration, a user entity may map the groups the user belongs to through `memberOf` attributes; a group entity may map which users belong to it through `uniqueMember` attributes; or both mappings may be maintained by the LDAP server.

Users generally authenticate against the server using a simple user name. When searching for group

membership information, depending on the directory server in use, searches could be performed using this simple name or using the distinguished name of the user's entry in the directory.

The authentication step of a user connecting to the server always happens first. Once the user is successfully authenticated the server loads the user's groups. The authentication step and the authorization step each require a connection to the LDAP server. The realm optimizes this process by reusing the authentication connection for the group loading step. As will be shown within the configuration steps below it is possible to define rules within the authorization section to convert a user's simple user name to their distinguished name. The result of a "user name to distinguished name mapping" search during authentication is cached and reused during the authorization query when the **force** attribute is set to "false". When **force** is true, the search is performed again during authorization (while loading groups). This is typically done when different servers perform authentication and authorization.

```
<authorization>
  <ldap connection="...">
    <!-- OPTIONAL -->
    <username-to-dn force="true">
      <!-- Only one of the following. -->
      <username-is-dn />
      <username-filter base-dn="..." recursive="..." user-dn-
attribute="..." attribute="..." />
      <advanced-filter base-dn="..." recursive="..." user-dn-
attribute="..." filter="..." />
    </username-to-dn>

    <group-search group-name="..." iterative="..." group-dn-
attribute="..." group-name-attribute="..." >
      <!-- One of the following -->
      <group-to-principal base-dn="..." recursive="..." search-
by="...">
        <membership-filter principal-attribute="..." />
      </group-to-principal>
      <principal-to-group group-attribute="..." />
    </group-search>
  </ldap>
</authorization>
```



IMPORTANT

These examples specify some attributes with their default values. This is done for demonstration. Attributes that specify their default values are removed from the configuration when it is persisted by the server. The exception is the **force** attribute. It is required, even when set to the default value of **false**.

username-to-dn

The **username-to-dn** element specifies how to map the user name to the distinguished name of their entry in the LDAP directory. This element is only required when *both* of the following are true:

- The authentication and authorization steps are against different LDAP servers.
- The group search uses the distinguished name.

1:1 username-to-dn

This specifies that the user name entered by the remote user is the user's distinguished name.

```
<username-to-dn force="false">
  <username-is-dn />
</username-to-dn>
```

This defines a 1:1 mapping and there is no additional configuration.

username-filter

The next option is very similar to the simple option described above for the authentication step. A specified attribute is searched for a match against the supplied user name.

```
<username-to-dn force="true">
  <username-filter base-dn="dc=people,dc=harold,dc=example,dc=com"
    recursive="false" attribute="sn" user-dn-attribute="dn" />
</username-to-dn>
```

The attributes that can be set here are:

- **base-dn:** The distinguished name of the context to begin the search.
- **recursive:** Whether the search will extend to sub contexts. Defaults to `false`.
- **attribute:** The attribute of the users entry to try and match against the supplied user name. Defaults to `uid`.
- **user-dn-attribute:** The attribute to read to obtain the users distinguished name. Defaults to `dn`.

advanced-filter

The final option is to specify an advanced filter, as in the authentication section this is an opportunity to use a custom filter to locate the users distinguished name.

```
<username-to-dn force="true">
  <advanced-filter base-dn="dc=people,dc=harold,dc=example,dc=com"
    recursive="false" filter="sAMAccountName={0}" user-dn-attribute="dn" />
</username-to-dn>
```

For the attributes that match those in the *username-filter* example, the meaning and default values are the same. There is one new attribute:

- **filter:** Custom filter used to search for a user's entry where the user name will be substituted in the `{0}` place holder.



IMPORTANT

The XML must remain valid after the filter is defined so if any special characters are used such as `&` ensure the proper form is used. For example `&` for the `&` character.

The Group Search

There are two different styles that can be used when searching for group membership information. The first style is where the user's entry contains an attribute that references the groups the user is a member of. The second style is where the group contains an attribute referencing the users entry.

When there is a choice of which style to use Red Hat recommends that the configuration for a user's entry referencing the group is used. This is because with this method group information can be loaded by reading attributes of known distinguished names without having to perform any searches. The other approach requires extensive searches to identify the groups that reference the user.

Before describing the configuration here are some LDIF examples to illustrate this.

Example 8.13. Principal to Group - LDIF example.

This example illustrates where we have a user **TestUserOne** who is a member of **GroupOne**, **GroupOne** is in turn a member of **GroupFive**. The group membership is shown by the use of a **memberOf** attribute which is set to the distinguished name of the group of which the user (or group) is a member.

It is not shown here but a user could potentially have multiple **memberOf** attributes set, one for each group of which the user is directly a member.

```
dn: uid=TestUserOne,ou=users,dc=principal-to-group,dc=example,dc=org
objectClass: extensibleObject
objectClass: top
objectClass: groupMember
objectClass: inetOrgPerson
objectClass: uidObject
objectClass: person
objectClass: organizationalPerson
cn: Test User One
sn: Test User One
uid: TestUserOne
distinguishedName: uid=TestUserOne,ou=users,dc=principal-to-
group,dc=example,dc=org
memberOf: uid=GroupOne,ou=groups,dc=principal-to-group,dc=example,dc=org
memberOf: uid=Slashy/Group,ou=groups,dc=principal-to-
group,dc=example,dc=org
userPassword: :
e1NTSEF9WFpURzhLVjc4WVZBQUJNbEI3Ym96UVAva0RTNlFNWUpLOTdTMUE9PQ==

dn: uid=GroupOne,ou=groups,dc=principal-to-group,dc=example,dc=org
objectClass: extensibleObject
objectClass: top
objectClass: groupMember
objectClass: group
objectClass: uidObject
uid: GroupOne
distinguishedName: uid=GroupOne,ou=groups,dc=principal-to-
group,dc=example,dc=org
memberOf: uid=GroupFive,ou=subgroups,ou=groups,dc=principal-to-
group,dc=example,dc=org

dn: uid=GroupFive,ou=subgroups,ou=groups,dc=principal-to-
group,dc=example,dc=org
```

```

objectClass: extensibleObject
objectClass: top
objectClass: groupMember
objectClass: group
objectClass: uidObject
uid: GroupFive
distinguishedName: uid=GroupFive,ou=subgroups,ou=groups,dc=principal-to-
group,dc=example,dc=org

```

Example 8.14. Group to Principal - LDIF Example

This example shows the same user `TestUserOne` who is a member of `GroupOne` which is in turn a member of `GroupFive` - however in this case it is an attribute `uniqueMember` from the group to the user being used for the cross reference.

Again the attribute used for the group membership cross reference can be repeated, if you look at `GroupFive` there is also a reference to another user `TestUserFive` which is not shown here.

```

dn: uid=TestUserOne,ou=users,dc=group-to-principal,dc=example,dc=org
objectClass: top
objectClass: inetOrgPerson
objectClass: uidObject
objectClass: person
objectClass: organizationalPerson
cn: Test User One
sn: Test User One
uid: TestUserOne
userPassword::
e1NTSEF9Sjr00TRDR1ltaHc1VVZQOEJvbXhUYjl1dkFVd1lQTmRLSEdzaWc9PQ==

dn: uid=GroupOne,ou=groups,dc=group-to-principal,dc=example,dc=org
objectClass: top
objectClass: groupOfUniqueNames
objectClass: uidObject
cn: Group One
uid: GroupOne
uniqueMember: uid=TestUserOne,ou=users,dc=group-to-
principal,dc=example,dc=org

dn: uid=GroupFive,ou=subgroups,ou=groups,dc=group-to-
principal,dc=example,dc=org
objectClass: top
objectClass: groupOfUniqueNames
objectClass: uidObject
cn: Group Five
uid: GroupFive
uniqueMember: uid=TestUserFive,ou=users,dc=group-to-
principal,dc=example,dc=org
uniqueMember: uid=GroupOne,ou=groups,dc=group-to-
principal,dc=example,dc=org

```

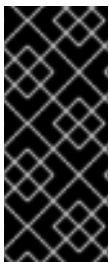
General Group Searching

Before looking at the examples for the two approaches shown above we first need to define the attributes common to both of these.

```
<group-search group-name="..." iterative="..." group-dn-attribute="..."
group-name-attribute="..." >
    ...
</group-search>
```

- **group-name:** This attribute is used to specify the form that should be used for the group name returned as the list of groups of which the user is a member. This can either be the simple form of the group name or the group's distinguished name. If the distinguished name is required this attribute can be set to **DISTINGUISHED_NAME**. Defaults to **SIMPLE**.
- **iterative:** This attribute is used to indicate if, after identifying the groups a user is a member of, we should also iteratively search based on the groups to identify which groups the groups are a member of. If iterative searching is enabled we keep going until either we reach a group that is not a member if any other groups or a cycle is detected. Defaults to **false**.

Cyclic group membership is not a problem. A record of each search is kept to prevent groups that have already been searched from being searched again.



IMPORTANT

For iterative searching to work the group entries need to look the same as user entries. The same approach used to identify the groups a user is a member of is then used to identify the groups of which the group is a member. This would not be possible if for group to group membership the name of the attribute used for the cross reference changes or if the direction of the reference changes.

- **group-dn-attribute:** On an entry for a group which attribute is its distinguished name. Defaults to **dn**.
- **group-name-attribute:** On an entry for a group which attribute is its simple name. Defaults to **uid**.

Example 8.15. Principal to Group Example Configuration

Based on the example LDIF from above here is an example configuration iteratively loading a user's groups where the attribute used to cross reference is the **memberOf** attribute on the user.

```
<authorization>
  <ldap connection="LocalLdap">
    <username-to-dn>
      <username-filter base-dn="ou=users,dc=principal-to-
group,dc=example,dc=org" recursive="false" attribute="uid" user-dn-
attribute="dn" />
    </username-to-dn>
    <group-search group-name="SIMPLE" iterative="true" group-dn-
attribute="dn" group-name-attribute="uid">
      <principal-to-group group-attribute="memberOf" />
    </group-search>
  </ldap>
</authorization>
```

The most important aspect of this configuration is that the `principal-to-group` element has been added with a single attribute.

- **group-attribute:** The name of the attribute on the user entry that matches the distinguished name of the group the user is a member of. Defaults to `memberOf`.

Example 8.16. Group to Principal Example Configuration

This example shows an iterative search for the group to principal LDIF example shown above.

```
<authorization>
  <ldap connection="LocalLdap">
    <username-to-dn>
      <username-filter base-dn="ou=users,dc=group-to-
principal,dc=example,dc=org" recursive="false" attribute="uid" user-dn-
attribute="dn" />
    </username-to-dn>
    <group-search group-name="SIMPLE" iterative="true" group-dn-
attribute="dn" group-name-attribute="uid">
      <group-to-principal base-dn="ou=groups,dc=group-to-
principal,dc=example,dc=org" recursive="true" search-
by="DISTINGUISHED_NAME">
        <membership-filter principal-attribute="uniqueMember"
/>
      </group-to-principal>
    </group-search>
  </ldap>
</authorization>
```

Here an element `group-to-principal` is added. This element is used to define how searches for groups that reference the user entry will be performed. The following attributes are set:

- **base-dn:** The distinguished name of the context to use to begin the search.
- **recursive:** Whether sub-contexts also be searched. Defaults to `false`.
- **search-by:** The form of the role name used in searches. Valid values are `SIMPLE` and `DISTINGUISHED_NAME`. Defaults to `DISTINGUISHED_NAME`.

Within the `group-to-principal` element there is a `membership-filter` element to define the cross reference.

- **principal-attribute:** The name of the attribute on the group entry that references the user entry. Defaults to `member`.

[Report a bug](#)

8.7.6. Hot Rod Interface Security

8.7.6.1. Publish Hot Rod Endpoints as a Public Interface

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

```
<socket-binding name="hotrod" interface="public" port="11222" />
```

[Report a bug](#)

8.7.6.2. Encryption of communication between Hot Rod Server and Hot Rod client

Hot Rod can be encrypted using TLS/SSL, and has the option to require certificate-based client authentication.

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

Procedure 8.3. Secure Hot Rod Using SSL/TLS

1. **Generate a Keystore**

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

2. **Place the Keystore in the Configuration Directory**

Put the keystore in the `~/JDG_HOME/standalone/configuration` directory with the `standalone-hotrod-ssl.xml` file from the `~/JDG_HOME/docs/examples/configs` directory.

3. **Declare an SSL Server Identity**

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

```
<server-identities>
  <ssl protocol="...">
    <keystore path="..." relative-to="..." keystore-
password="${VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::ENCRYPTED_VALUE}" />
  </ssl>
  <secret value="..." />
</server-identities>
```

See [Section 8.7.7.4, “Configure Hot Rod Authentication \(X.509\)”](#) for details about these parameters.

4. **Add the Security Element**

Add the security element to the Hot Rod connector as follows:

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

- a. **Server Authentication of Certificate**

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

5. Start the Server

Start the server using the following:

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

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

Securing Hot Rod using SSL can also be configured programmatically. This configuration should be used in order to prevent passwords from being stored as plain text in configuration files

Example 8.17. Secure Hot Rod Using SSL/TLS

```
package org.infinispan.client.hotrod.configuration;

import java.util.Arrays;

import javax.net.ssl.KeyManager;
import javax.net.ssl.SSLContext;
import javax.net.ssl.TrustManager;

public class SslConfiguration {
    private final boolean enabled;
    private final String keyStoreFileName;
    private final char[]
VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::keyStorePassword;
    private final SSLContext sslContext;
    private final String trustStoreFileName;
    private final char[]
VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::trustStorePassword;

    SslConfiguration(boolean enabled, String keyStoreFileName, char[]
keyStorePassword, SSLContext sslContext, String trustStoreFileName,
char[] trustStorePassword) {
        this.enabled = enabled;
        this.keyStoreFileName = keyStoreFileName;
        this.keyStorePassword =
VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::keyStorePassword;
        this.sslContext = sslContext;
        this.trustStoreFileName = trustStoreFileName;
        this.trustStorePassword =
VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::trustStorePassword;
    }

    public boolean enabled() {
        return enabled;
    }

    public String keyStoreFileName() {
        return keyStoreFileName;
    }
}
```

```

public char[] keyStorePassword() {
    return keyStorePassword;
}

public SSLContext sslContext() {
    return sslContext;
}

public String trustStoreFileName() {
    return trustStoreFileName;
}

public char[] trustStorePassword() {
    return trustStorePassword;
}

@Override
public String toString() {
    return "SslConfiguration [enabled=" + enabled + ", keyStoreFileName="
        + keyStoreFileName + ", sslContext=" + sslContext + ",
trustStoreFileName=" + trustStoreFileName + "];"
}
}

```



IMPORTANT

To prevent plain text passwords from appearing in configurations or source codes, plain text passwords should be changed to Vault passwords. For more information about how to set up Vault passwords, see the *Red Hat Enterprise Application Platform Security Guide*.

[Report a bug](#)

8.7.7. User Authentication over Hot Rod Using SASL

User authentication over Hot Rod can be implemented using the following Simple Authentication and Security Layer (SASL) mechanisms:

- **PLAIN** is the least secure mechanism because credentials are transported in plain text format. However, it is also the simplest mechanism to implement. This mechanism can be used in conjunction with encryption (SSL) for additional security.
- **DIGEST-MD5** is a mechanism that hashes the credentials before transporting them. As a result, it is more secure than the **PLAIN** mechanism.
- **GSSAPI** is a mechanism that uses Kerberos tickets. As a result, it requires a correctly configured Kerberos Domain Controller (for example, Microsoft Active Directory).
- **EXTERNAL** is a mechanism that obtains the required credentials from the underlying transport (for example, from a X.509 client certificate) and therefore requires client certificate encryption to work correctly.

[Report a bug](#)

8.7.7.1. Configure Hot Rod Authentication (GSSAPI/Kerberos)

Use the following steps to set up Hot Rod Authentication using the SASL GSSAPI/Kerberos mechanism:

Procedure 8.4. Configure SASL GSSAPI/Kerberos Authentication

1. Server-side Configuration

The following steps must be configured on the server-side:

- a. Define a Kerberos security login module using the security domain subsystem:

```
<system-properties>
  <property name="java.security.krb5.conf"
value="/tmp/infinispan/krb5.conf"/>
  <property name="java.security.krb5.debug" value="true"/>
  <property name="jboss.security.disable.secdomain.option"
value="true"/>
</system-properties>

<security-domain name="infinispan-server" cache-type="default">
  <authentication>
    <login-module code="Kerberos" flag="required">
      <module-option name="debug" value="true"/>
      <module-option name="storeKey" value="true"/>
      <module-option name="refreshKrb5Config"
value="true"/>
      <module-option name="useKeyTab" value="true"/>
      <module-option name="doNotPrompt" value="true"/>
      <module-option name="keyTab"
value="/tmp/infinispan/infinispan.keytab"/>
      <module-option name="principal"
value="HOTROD/localhost@INFINISPAN.ORG"/>
    </login-module>
  </authentication>
</security-domain>
```

- b. Configure a Hot Rod connector as follows:

```
<hotrod-connector socket-binding="hotrod"
cache-container="default">
  <authentication security-realm="ApplicationRealm">
    <sasl server-name="node0"
mechanisms="{mechanism_name}"
qop="{qop_name}"
strength="{value}">
    <policy>
      <no-anonymous value="true" />
    </policy>
    <property
name="com.sun.security.sasl.digest.utf8">true</property>
  </sasl>
  </authentication>
</hotrod-connector>
```

- The *server-name* attribute specifies the name that the server declares to incoming clients. The client configuration must also contain the same server name value.
- The *server-context-name* attribute specifies the name of the login context used to retrieve a server subject for certain SASL mechanisms (for example, GSSAPI).
- The *mechanisms* attribute specifies the authentication mechanism in use. See [Section 8.7.7, “User Authentication over Hot Rod Using SASL”](#) for a list of supported mechanisms.
- The *qop* attribute specifies the SASL quality of protection value for the configuration. Supported values for this attribute are **auth** (authentication), **auth-int** (authentication and integrity, meaning that messages are verified against checksums to detect tampering), and **auth-conf** (authentication, integrity, and confidentiality, meaning that messages are also encrypted). Multiple values can be specified, for example, **auth-int auth-conf**. The ordering implies preference, so the first value which matches both the client and server's preference is chosen.
- The *strength* attribute specifies the SASL cipher strength. Valid values are **low**, **medium**, and **high**.
- The **no-anonymous** element within the **policy** element specifies whether mechanisms that accept anonymous login are permitted. Set this value to **false** to permit and **true** to deny.

2. Client-side Configuration

The following steps must be configured on the client-side:

- a. Define a login module in a login configuration file (**gss.conf**) on the client side:

```
GssExample {
    com.sun.security.auth.module.Krb5LoginModule required
    client=TRUE;
};
```

- b. Set up the following system properties:

```
java.security.auth.login.config=gss.conf
java.security.krb5.conf=/etc/krb5.conf
```



NOTE

The **krb5.conf** file is dependent on the environment and must point to the Kerberos Key Distribution Center.

- c. Configure the Hot Rod Client:

```
public class MyCallbackHandler implements CallbackHandler {
    final private String username;
    final private char[] password;
    final private String realm;

    public MyCallbackHandler() { }
```

```

    public MyCallbackHandler (String username, String realm,
char[] password) {
        this.username = username;
        this.password = password;
        this.realm = realm;
    }

    @Override
    public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            if (callback instanceof NameCallback) {
                NameCallback nameCallback = (NameCallback) callback;
                nameCallback.setName(username);
            } else if (callback instanceof PasswordCallback) {
                PasswordCallback passwordCallback =
(PasswordCallback) callback;
                passwordCallback.setPassword(password);
            } else if (callback instanceof AuthorizeCallback) {
                AuthorizeCallback authorizeCallback =
(AuthorizeCallback) callback;

                authorizeCallback.setAuthorized(authorizeCallback.getAuthenticati
onID().equals(
                    authorizeCallback.getAuthorizationID()));
            } else if (callback instanceof RealmCallback) {
                RealmCallback realmCallback = (RealmCallback)
callback;
                realmCallback.setText(realm);
            } else {
                throw new UnsupportedCallbackException(callback);
            }
        }
    }
}

LoginContext lc = new LoginContext("GssExample", new
MyCallbackHandler("krb_user",
"krb_password".toCharArray()));lc.login();Subject clientSubject =
lc.getSubject();
ConfigurationBuilder clientBuilder = new
ConfigurationBuilder();clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
        .socketTimeout(1200000)
        .security()
            .authentication()
                .enable()
                .serverName("infinispan-server")
                .saslmMechanism("GSSAPI")
                .clientSubject(clientSubject)
                .callbackHandler(new
MyCallbackHandler());remoteCacheManager = new
RemoteCacheManager(clientBuilder.build());RemoteCache<String,
String> cache = remoteCacheManager.getCache("secured");

```

[Report a bug](#)

8.7.7.2. Configure Hot Rod Authentication (MD5)

Use the following steps to set up Hot Rod Authentication using the SASL using the MD5 mechanism:

Procedure 8.5. Configure Hot Rod Authentication (MD5)

1. Set up the Hot Rod Connector configuration by adding the `sasl` element to the `authentication` element (for details on the `authentication` element, see [Section 8.7.4, “Configuring Security Realms Declaratively”](#)) as follows:

```
<hotrod-connector socket-binding="hotrod"
                 cache-container="default">
  <authentication security-realm="ApplicationRealm">
    <sasl server-name="myhotrodserver"
         mechanisms="DIGEST-MD5"
         qop="auth" />
  </authentication>
</hotrod-connector>
```

- o The *server-name* attribute specifies the name that the server declares to incoming clients. The client configuration must also contain the same server name value.
 - o The *mechanisms* attribute specifies the authentication mechanism in use. See [Section 8.7.7, “User Authentication over Hot Rod Using SASL”](#) for a list of supported mechanisms.
 - o The *qop* attribute specifies the SASL quality of production value for the configuration. Supported values for this attribute are `auth`, `auth-int`, and `auth-conf`.
2. Connect the client to the configured Hot Rod connector as follows:

```
public class MyCallbackHandler implements CallbackHandler {
    final private String username;
    final private char[] password;
    final private String realm;

    public MyCallbackHandler (String username, String realm, char[]
password) {
        this.username = username;
        this.password = password;
        this.realm = realm;
    }

    @Override
    public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            if (callback instanceof NameCallback) {
                NameCallback nameCallback = (NameCallback) callback;
                nameCallback.setName(username);
            } else if (callback instanceof PasswordCallback) {
                PasswordCallback passwordCallback = (PasswordCallback)
callback;

```

```

        passwordCallback.setPassword(password);
    } else if (callback instanceof AuthorizeCallback) {
        AuthorizeCallback authorizeCallback =
(AuthorizeCallback) callback;

authorizeCallback.setAuthorized(authorizeCallback.getAuthenticationI
D().equals(
        authorizeCallback.getAuthorizationID()));
    } else if (callback instanceof RealmCallback) {
        RealmCallback realmCallback = (RealmCallback) callback;
        realmCallback.setText(realm);
    } else {
        throw new UnsupportedOperationException(callback);
    }
}
}}
ConfigurationBuilder clientBuilder = new
ConfigurationBuilder();clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .socketTimeout(1200000)
    .security()
        .authentication()
            .enable()
            .serverName("myhotrodserver")
            .saslMechanism("DIGEST-MD5")
            .callbackHandler(new MyCallbackHandler("myuser",
"ApplicationRealm", "qwer1234!".toCharArray()));remoteCacheManager =
new RemoteCacheManager(clientBuilder.build());RemoteCache<String,
String> cache = remoteCacheManager.getCache("secured");

```

[Report a bug](#)

8.7.7.3. Configure Hot Rod Using LDAP/Active Directory

Use the following to configure authentication over Hot Rod using LDAP or Microsoft Active Directory:

```

<security-realms>
  <security-realm name="ApplicationRealm">
    <authentication>
      <ldap connection="ldap_connection"
        recursive="true"
        base-dn="cn=users,dc=infinispan,dc=org">
        <username-filter attribute="cn" />
      </ldap>
    </authentication>
  </security-realm>
</security-realms>
<outbound-connections>
  <ldap name="ldap_connection"
    url="ldap://my_ldap_server"
    search-dn="CN=test,CN=Users,DC=infinispan,DC=org"
    search-credential="Test_password"/>
</outbound-connections>

```

The following are some details about the elements and parameters used in this configuration:

- The `security-realm` element's `name` parameter specifies the security realm to reference to use when establishing the connection.
- The `authentication` element contains the authentication details.
- The `ldap` element specifies how LDAP searches are used to authenticate a user. First, a connection to LDAP is established and a search is conducted using the supplied user name to identify the distinguished name of the user. A subsequent connection to the server is established using the password supplied by the user. If the second connection succeeds, the authentication is a success.
 - The `connection` parameter specifies the name of the connection to use to connect to LDAP.
 - The (optional) `recursive` parameter specifies whether the filter is executed recursively. The default value for this parameter is `false`.
 - The `base-dn` parameter specifies the distinguished name of the context to use to begin the search from.
 - The (optional) `user-dn` parameter specifies which attribute to read for the user's distinguished name after the user is located. The default value for this parameter is `dn`.
- The `outbound-connections` element specifies the name of the connection used to connect to the LDAP directory.
- The `ldap` element specifies the properties of the outgoing LDAP connection.
 - The `name` parameter specifies the unique name used to reference this connection.
 - The `url` parameter specifies the URL used to establish the LDAP connection.
 - The `search-dn` parameter specifies the distinguished name of the user to authenticate and to perform the searches.
 - The `search-credential` parameter specifies the password required to connect to LDAP as the `search-dn`.
 - The (optional) `initial-context-factory` parameter allows the overriding of the initial context factory. the default value of this parameter is `com.sun.jndi.ldap.LdapCtxFactory`.

[Report a bug](#)

8.7.7.4. Configure Hot Rod Authentication (X.509)

The X.509 certificate can be installed at the node, and be made available to other nodes for authentication purposes for inbound and outbound SSL connections. This is enabled using the `<server-identities/>` element of a security realm definition, which defines how a server appears to external applications. This element can be used to configure a password to be used when establishing a remote connection, as well as the loading of an X.509 key.

The following example shows how to install an X.509 certificate on the node.

-

```

<security-realm name="ApplicationRealm">
  <server-identities>
    <ssl protocol="...">
      <keystore path="..." relative-to="..." keystore-password="..."
alias="..." key-password="..." />
    </ssl>
  </server-identities>

  [... authentication/authorization ...]

</security-realms>

```

In the provided example, the SSL element contains the `<keystore/>` element, which is used to define how to load the key from the file-based keystore. The following parameters are available for this element.

Table 8.4. `<server-identities/>` Options

Parameter	Mandatory/Optional	Description
<i>path</i>	Mandatory	This is the path to the keystore, this can be an absolute path or relative to the next attribute.
<i>relative-to</i>	Optional	The name of a service representing a path the keystore is relative to.
<i>keystore-password</i>	Mandatory	The password required to open the keystore.
<i>alias</i>	Optional	The alias of the entry to use from the keystore - for a keystore with multiple entries in practice the first usable entry is used but this should not be relied on and the alias should be set to guarantee which entry is used.
<i>key-password</i>	Optional	The password to load the key entry, if omitted the keystore-password will be used instead.



NOTE

If the following error occurs, specify a *key-password* as well as an *alias* to ensure only one key is loaded.

```
UnrecoverableKeyException: Cannot recover key
```

[Report a bug](#)

8.8. ACTIVE DIRECTORY AUTHENTICATION (NON-KERBEROS)

See [Example 8.2, “Example of JBoss EAP LDAP login module configuration”](#) for a non-Kerberos Active Directory Authentication configuration example.

[Report a bug](#)

8.9. ACTIVE DIRECTORY AUTHENTICATION USING KERBEROS (GSSAPI)

When using Red Hat JBoss Data Grid with Microsoft Active Directory, data security can be enabled via Kerberos authentication. To configure Kerberos authentication for Microsoft Active Directory, use the following procedure.

Procedure 8.6. Configure Kerberos Authentication for Active Directory (Library Mode)

1. Configure JBoss EAP server to authenticate itself to Kerberos. This can be done by configuring a dedicated security domain, for example:

```
<security-domain name="ldap-service" cache-type="default">
  <authentication>
    <login-module code="Kerberos" flag="required">
      <module-option name="storeKey" value="true"/>
      <module-option name="useKeyTab" value="true"/>
      <module-option name="refreshKrb5Config" value="true"/>
      <module-option name="principal"
value="ldap/localhost@INFINISPAN.ORG"/>
      <module-option name="keyTab"
value="${basedir}/keytab/ldap.keytab"/>
      <module-option name="doNotPrompt" value="true"/>
    </login-module>
  </authentication>
</security-domain>
```

2. The security domain for authentication must be configured correctly for JBoss EAP, an application must have a valid Kerberos ticket. To initiate the Kerberos ticket, you must reference another security domain using

```
<module-option name="usernamePasswordDomain" value="krb-admin"/>
```

. This points to the standard Kerberos login module described in Step 3.

```
<security-domain name="ispn-admin" cache-type="default">
  <authentication>
    <login-module code="SPNEGO" flag="requisite">
      <module-option name="password-stacking"
value="useFirstPass"/>
      <module-option name="serverSecurityDomain" value="ldap-
service"/>
      <module-option name="usernamePasswordDomain"
value="krb-admin"/>
    </login-module>
    <login-module code="AdvancedAdLdap" flag="required">
      <module-option name="password-stacking"
```



```

value="useFirstPass"/>
    <module-option name="bindAuthentication"
value="GSSAPI"/>
    <module-option name="jaasSecurityDomain" value="ldap-
service"/>
    <module-option name="java.naming.provider.url"
value="ldap://localhost:389"/>
    <module-option name="baseCtxDN"
value="ou=People,dc=infinispan,dc=org"/>
    <module-option name="baseFilter" value="
(krb5PrincipalName={0})"/>
    <module-option name="rolesCtxDN"
value="ou=Roles,dc=infinispan,dc=org"/>
    <module-option name="roleFilter" value="(member={1})"/>
    <module-option name="roleAttributeID" value="cn"/>
    </login-module>
</authentication>
</security-domain>

```

3. The security domain authentication configuration described in the previous step points to the following standard Kerberos login module:

```

<security-domain name="krb-admin" cache-type="default">
  <authentication>
    <login-module code="Kerberos" flag="required">
      <module-option name="useKeyTab" value="true"/>
      <module-option name="principal"
value="admin@INFINISPAN.ORG"/>
      <module-option name="keyTab"
value="\${basedir}/keytab/admin.keytab"/>
    </login-module>
  </authentication>
</security-domain>

```

[Report a bug](#)

8.10. THE SECURITY AUDIT LOGGER

Red Hat JBoss Data Grid includes a logger to audit security logs for the cache, specifically whether a cache or a cache manager operation was allowed or denied for various operations.

The default audit logger is `org.infinispan.security.impl.DefaultAuditLogger`. This logger outputs audit logs using the available logging framework (for example, JBoss Logging) and provides results at the **TRACE** level and the **AUDIT** category.

To send the **AUDIT** category to either a log file, a JMS queue, or a database, use the appropriate log appender.

[Report a bug](#)

8.10.1. Configure the Security Audit Logger (Library Mode)

Use the following to declaratively configure the audit logger in Red Hat JBoss Data Grid:

```

<infinispan>
  ...
  <global-security>
    <authorization audit-logger =
"org.infinispan.security.impl.DefaultAuditLogger">
      ...
    </authorization>
  </global-security>
  ...
</infinispan>

```

Use the following to programatically configure the audit logger in JBoss Data Grid:

```

GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
global.security()
    .authorization()
        .auditLogger(new DefaultAuditLogger());

```

[Report a bug](#)

8.10.2. Configure the Security Audit Logger (Remote Client-Server Mode)

Use the following code to configure the audit logger in Red Hat JBoss Data Grid Remote Client-Server Mode.

To use a different audit logger, specify it in the `<authorization>` element. The `<authorization>` element must be within the `<cache-container>` element in the Infinispan subsystem (in the `standalone.xml` configuration file).

```

<cache-container name="local" default-cache="default">
  <security>
    <authorization audit-
logger="org.infinispan.security.impl.DefaultAuditLogger">
    <identity-role-mapper/>
    <role name="admin" permissions="ALL"/>
    <role name="reader" permissions="READ"/>
    <role name="writer" permissions="WRITE"/>
    <role name="supervisor" permissions="ALL_READ ALL_WRITE"/>
    </authorization>
  </security>
  <local-cache name="default" start="EAGER">
    <locking isolation="NONE" acquire-timeout="30000" concurrency-
level="1000" striping="false"/>
    <transaction mode="NONE"/>
    <security>
    <authorization roles="admin reader writer supervisor"/>
    </security>
  </local-cache>

```

**NOTE**

The default audit logger for server mode is **`org.jboss.as.clustering.infinispan.subsystem.ServerAuditLogger`** which sends the log messages to the server audit log. See the *Management Interface Audit Logging* chapter in the JBoss Enterprise Application Platform *Administration and Configuration Guide* for more information.

[Report a bug](#)

8.10.3. Custom Audit Loggers

Users can implement custom audit loggers in Red Hat JBoss Data Grid Library and Remote Client-Server Mode. The custom logger must implement the **`org.infinispan.security.AuditLogger`** interface. If no custom logger is provided, the default logger (**`DefaultAuditLogger`**) is used.

[Report a bug](#)

CHAPTER 9. SECURITY FOR CLUSTER TRAFFIC

9.1. NODE AUTHENTICATION AND AUTHORIZATION (REMOTE CLIENT-SERVER MODE)

Security can be enabled at node level via SASL protocol, which enables node authentication against a security realm. This requires nodes to authenticate each other when joining or merging with a cluster. For detailed information about security realms, see [Section 8.7.1, “About Security Realms”](#).

The following example depicts the `<sasl />` element, which leverages the SASL protocol. Both **DIGEST-MD5** or **GSSAPI** mechanisms are currently supported.

Example 9.1. Configure SASL Authentication

```
<management>
  <security-realms>
    ...
    <security-realm name="ClusterRealm">
      <authentication>
        <properties path="cluster-users.properties" relative-
to="jboss.server.config.dir"/>
      </authentication>
      <authorization>
        <properties path="cluster-roles.properties"
relative-to="jboss.server.config.dir"/>
      </authorization>
    </security-realm>
  </security-realms>
  ...
</management>

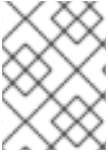
<stack name="udp">
  ...
  <sasl mech="DIGEST-MD5" security-realm="ClusterRealm" cluster-
role="cluster">
    <property name="client_name">node1</property>
    <property name="client_password">password</property>
  </sasl>
  ...
</stack>
```

In the provided example, the nodes use the **DIGEST-MD5** mechanism to authenticate against the **ClusterRealm**. In order to join, nodes must have the **cluster** role.

The **cluster-role** attribute determines the role all nodes must belong to in the security realm in order to **JOIN** or **MERGE** with the cluster. Unless it has been specified, the **cluster-role** attribute is the name of the clustered **<cache-container>** by default. Each node identifies itself using the **client-name** property. If none is specified, the hostname on which the server is running will be used.

This name can also be overridden by specifying the `jboss.node.name` system property that can be overridden on the command line. For example:

```
$ clustered.sh -Djboss.node.name=node001
```



NOTE

JGroups AUTH protocol is not integrated with security realms, and its use is not advocated for Red Hat JBoss Data Grid.

[Report a bug](#)

9.1.1. Configure Node Authentication for Cluster Security (DIGEST-MD5)

The following example demonstrates how to use **DIGEST-MD5** with a properties-based security realm, with a dedicated realm for cluster node.

Example 9.2. Using the DIGEST-MD5 Mechanism

```
<management>
  <security-realms>
    <security-realm name="ClusterRealm">
      <authentication>
        <properties path="cluster-users.properties"
relative-to="jboss.server.config.dir"/>
      </authentication>
      <authorization>
        <properties path="cluster-roles.properties"
relative-to="jboss.server.config.dir"/>
      </authorization>
    </security-realm>
  </security-realms>
</management>
<subsystem xmlns="urn:infinispa:server:jgroups:6.1" default-
stack="{jboss.default.jgroups.stack:udp}">
  <stack name="udp">
    <transport type="UDP" socket-binding="jgroups-udp"/>
    <protocol type="PING"/>
    <protocol type="MERGE2"/>
    <protocol type="FD SOCK" socket-binding="jgroups-udp-fd"/>
    <protocol type="FD_ALL"/>
    <protocol type="pbcast.NAKACK"/>
    <protocol type="UNICAST2"/>
    <protocol type="pbcast.STABLE"/>
    <protocol type="pbcast.GMS"/>
    <protocol type="UFC"/>
    <protocol type="MFC"/>
    <protocol type="FRAG2"/>
    <protocol type="RSVP"/>
    <sasl security-realm="ClusterRealm" mech="DIGEST-MD5">
      <property name="client_password">...</property>
    </sasl>
  </stack>
</subsystem>
<subsystem xmlns="urn:infinispan:server:core:6.1" default-cache-
container="clustered">
  <cache-container name="clustered" default-cache="default">
```

```

        <transport executor="infinispan-transport" lock-
        timeout="60000" stack="udp"/>
        <!-- various clustered cache definitions here -->
    </cache-container>
</subsystem>

```

In the provided example, supposing the hostnames of the various nodes are `node001`, `node002`, `node003`, the `cluster-users.properties` will contain:

- `node001=/<node001passwordhash>/`
- `node002=/<node002passwordhash>/`
- `node003=/<node003passwordhash>/`

The `cluster-roles.properties` will contain:

- `node001=clustered`
- `node002=clustered`
- `node003=clustered`

To generate these values, the following `add-users.sh` script can be used:

```

$ add-user.sh -up cluster-users.properties -gp cluster-roles.properties -r
ClusterRealm -u node001 -g clustered -p <password>

```

The MD5 password hash of the node must also be placed in the "`client_password`" property of the `<sas/>` element.

```

<property name="client_password">...</property>

```



NOTE

To increase security, it is recommended that this password be stored using a Vault. For more information about vault expressions, see the *Red Hat Enterprise Application Platform Security Guide*

Once node security has been set up as discussed here, the cluster coordinator will validate each **JOINing** and **MERGEing** node's credentials against the realm before letting the node become part of the cluster view.

[Report a bug](#)

9.1.2. Configure Node Authentication for Cluster Security (GSSAPI/Kerberos)

When using the GSSAPI mechanism, the `client_name` is used as the name of a Kerberos-enabled login module defined within the security domain subsystem. For a full procedure on how to do this, see [Section 8.7.7.1, "Configure Hot Rod Authentication \(GSSAPI/Kerberos\)"](#).

Example 9.3. Using the Kerberos Login Module

```

<security-domain name="krb-node0" cache-type="default">
  <authentication>
    <login-module code="Kerberos" flag="required">
      <module-option name="storeKey" value="true"/>
      <module-option name="useKeyTab" value="true"/>
      <module-option name="refreshKrb5Config" value="true"/>
      <module-option name="principal"
value="jgroups/node0/clustered@INFINISPAN.ORG"/>
      <module-option name="keyTab"
value="${jboss.server.config.dir}/keytabs/jgroups_node0_clustered.keytab
"/>
      <module-option name="doNotPrompt" value="true"/>
    </login-module>
  </authentication>
</security-domain>

```

The following property must be set in the `<sasl/>` element to reference it:

```

<sasl ...>
  <property name="login_module_name">...</property>
</sasl>

```

As a result, the `authentication` section of the security realm is ignored, as the nodes will be validated against the Kerberos Domain Controller. The `authorization` configuration is still required, as the node principal must belong to the required cluster-role.

In all cases, it is recommended that a shared authorization database, such as LDAP, be used to validate node membership in order to simplify administration.

By default, the principal of the joining node must be in the following format:

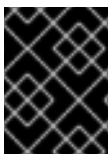
```
jgroups/$NODE_NAME/$CACHE_CONTAINER_NAME@REALM
```

[Report a bug](#)

9.2. CONFIGURE NODE SECURITY IN LIBRARY MODE

In Library mode, node authentication is configured directly in the JGroups configuration. JGroups can be configured so that nodes must authenticate each other when joining or merging with a cluster. The authentication uses SASL and is enabled by adding the SASL protocol to your JGroups XML configuration.

SASL relies on JAAS notions, such as `CallbackHandlers`, to obtain certain information necessary for the authentication handshake. Users must supply their own `CallbackHandlers` on both client and server sides.



IMPORTANT

The JAAS API is only available when configuring user authentication and authorization, and is not available for node security.

**NOTE**

In the provided example, **CallbackHandler** classes are examples only, and not contained in the Red Hat JBoss Data Grid release. Users must provide the appropriate **CallbackHandler** classes for their specific LDAP implementation.

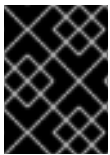
Example 9.4. Setting Up SASL Authentication in JGroups

```
<SASL mech="DIGEST-MD5"
  client_name="node_user"
  client_password="node_password"

  server_callback_handler_class="org.example.infinispan.security.JGroupsSaslServerCallbackHandler"

  client_callback_handler_class="org.example.infinispan.security.JGroupsSaslClientCallbackHandler"
  sasl_props="com.sun.security.sasl.digest.realm=test_realm" />
```

The above example uses the **DIGEST-MD5** mechanism. Each node must declare the user and password it will use when joining the cluster.

**IMPORTANT**

The SASL protocol must be placed before the GMS protocol in order for authentication to take effect.

The following example demonstrates how to implement a **CallbackHandler** class. In this example, login and password are checked against values provided via Java properties when JBoss Data Grid is started, and authorization is checked against *role* which is defined in the class ("test_user").

Example 9.5. Callback Handler Class

```
public class SaslPropAuthUserCallbackHandler implements CallbackHandler
{
    private static final String APPROVED_USER = "test_user";

    private final String name;
    private final char[] password;
    private final String realm;

    public SaslPropAuthUserCallbackHandler() {
        this.name = System.getProperty("sasl.username");
        this.password = System.getProperty("sasl.password").toCharArray();
        this.realm = System.getProperty("sasl.realm");
    }

    @Override
    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
        for (Callback callback : callbacks) {
```



```

        if (callback instanceof PasswordCallback) {
            ((PasswordCallback) callback).setPassword(password);
        } else if (callback instanceof NameCallback) {
            ((NameCallback) callback).setName(name);
        } else if (callback instanceof AuthorizeCallback) {
            AuthorizeCallback authorizeCallback = (AuthorizeCallback)
callback;
            if
(APPROVED_USER.equals(authorizeCallback.getAuthorizationID())) {
                authorizeCallback.setAuthorized(true);
            } else {
                authorizeCallback.setAuthorized(false);
            }
        } else if (callback instanceof RealmCallback) {
            RealmCallback realmCallback = (RealmCallback) callback;
            realmCallback.setText(realm);
        } else {
            throw new UnsupportedCallbackException(callback);
        }
    }
}
}
}

```

For authentication, specify the `javax.security.auth.callback.NameCallback` and `javax.security.auth.callback.PasswordCallback` callbacks

For authorization, specify the callbacks required for authentication, as well as specifying the `javax.security.sasl.AuthorizeCallback` callback.

[Report a bug](#)

9.2.1. Configure Node Authentication for Library Mode (DIGEST-MD5)

The behavior of a node differs depending on whether it is the coordinator node or any other node. The coordinator acts as the SASL server, with the joining or merging nodes behaving as SASL clients. When using the DIGEST-MD5 mechanism in Library mode, the server and client callback must be specified so that the server and client are aware of how to obtain the credentials. Therefore, two `CallbackHandlers` are required:

- The `server_callback_handler_class` is used by the coordinator.
- The `client_callback_handler_class` is used by other nodes.

The following example demonstrates these `CallbackHandlers`.

Example 9.6. Callback Handlers

```

<SASL mech="DIGEST-MD5"
  client_name="node_name"
  client_password="node_password"

  client_callback_handler_class="${CLIENT_CALLBACK_HANDLER_IN_CLASSPATH}"

```

```
server_callback_handler_class="${SERVER_CALLBACK_HANDLER_IN_CLASSPATH}"
    sasl_props="com.sun.security.sasl.digest.realm=test_realm"
/>
```

JGroups is designed so that all nodes are able to act as coordinator or client depending on cluster behavior, so if the current coordinator node goes down, the next node in the succession chain will become the coordinator. Given this behavior, both server and client callback handlers must be identified within SASL for Red Hat JBoss Data Grid implementations.

[Report a bug](#)

9.2.2. Configure Node Authentication for Library Mode (GSSAPI)

When performing node authentication in Library mode using the GSSAPI mechanism, the *login_module_name* parameter must be specified instead of *callback*.

This login module is used to obtain a valid Kerberos ticket, which is used to authenticate a client to the server. The *server_name* must also be specified, as the client principal is constructed as `jgroups/$server_name@REALM`.

Example 9.7. Specifying the login module and server on the coordinator node

```
<SASL mech="GSSAPI"
    server_name="node0/clustered"
    login_module_name="krb-node0"

server_callback_handler_class="org.infinispan.test.integration.security.
utils.SaslPropCallbackHandler" />
```

On the coordinator node, the *server_callback_handler_class* must be specified for node authorization. This will determine if the authenticated joining node has permission to join the cluster.



NOTE

The server principal is always constructed as `jgroups/server_name`, therefore the server principal in Kerberos must also be `jgroups/server_name`. For example, if the server name in Kerberos is `jgroups/node1/mycache`, then the server name must be `node1/mycache`.

[Report a bug](#)

9.2.3. Node Authorization in Library Mode

The SASL protocol in JGroups is concerned only with the authentication process. To implement node authorization, you can do so within the server callback handler by throwing an Exception.

The following example demonstrates this.

Example 9.8. Implementing Node Authorization

```

public class AuthorizingServerCallbackHandler implements CallbackHandler
{
    @Override
    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            ...
            if (callback instanceof AuthorizeCallback) {
                AuthorizeCallback acb = (AuthorizeCallback) callback;
                if
                (!"myclusterrole".equals(acb.getAuthenticationID())) {
                    throw new SecurityException("Unauthorized node "
+user);
                }
                ...
            }
        }
    }
}

```

[Report a bug](#)

9.3. JGROUPS ENCRYPT

JGroups includes the **ENCRYPT** protocol to provide encryption for cluster traffic.

By default, encryption only encrypts the message body; it does not encrypt message headers. To encrypt the entire message, including all headers, as well as destination and source addresses, the property *encrypt_entire_message* must be `true`. **ENCRYPT** must also be below any protocols with headers that must be encrypted.

The **ENCRYPT** layer is used to encrypt and decrypt communication in JGroups. The JGroups **ENCRYPT** protocol can be used in two ways:

- Configured with a `secretKey` in a keystore.
- Configured with algorithms and key sizes.

Each message is identified as encrypted with a specific encryption header identifying the encrypt header and an MD5 digest identifying the version of the key being used to encrypt and decrypt messages.

[Report a bug](#)

9.3.1. ENCRYPT Configured with a `secretKey` in a Key Store

The **ENCRYPT** class can be configured with a `secretKey` in a keystore so it is usable at any layer in JGroups without requiring a coordinator. This also provides protection against passive monitoring without the complexity of key exchange.

This method can be used by inserting the encryption layer at any point in the JGroups stack, which will encrypt all events of a type `MSG` that have a non-null message buffer. The following is an example of the entry in this form:

■

```
<ENCRYPT key_store_name="defaultStore.keystore"
store_password="{VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::ENCRYPTED_VALUE}"
alias="myKey"/>
```

The `secretKey` must be already generated in a keystore file in order to use the **ENCRYPT** layer in this manner. The directory containing the keystore file must be on the application classpath.



NOTE

The `secretKey` keystore file cannot be created using the `keytool` application shipped with JDK. The `KeyStoreGenerator` java file is included in the demo package that can be used to generate a suitable keystore.

[Report a bug](#)

9.3.2. ENCRYPT Using a Key Store

ENCRYPT uses store type JCEKS. To generate a keystore compatible with JCEKS, use the following command line options:

```
$ keytool -genseckey -alias myKey -keypass changeit -storepass changeit -
keyalg Blowfish -keysize 56 -keystore defaultStore.keystore -storetype
JCEKS
```

ENCRYPT can then be configured by adding the following information to the JGroups file used by the application.

```
<ENCRYPT key_store_name="defaultStore.keystore"
store_password="{VAULT::VAULT_BLOCK::ATTRIBUTE_NAME::ENCRYPTED_VALUE}"
alias="myKey"/>
```

Standard Java properties can also be used in the configuration, and it is possible to pass the path to JGroups configuration via the `-D` option during start up.

The default, pre-configured JGroups files are packaged in `infinispan-embedded.jar`, alternatively, you can create your own configuration file. See the *Configure JGroups (Library Mode)* section in the *Red Hat JBoss Data Grid Administration and Configuration Guide* for instructions on how to set up JBoss Data Grid to use custom JGroups configurations in library mode.

In Remote Client-Server mode, the JGroups configuration is part of the main server configuration file.



NOTE

The `defaultStore.keystore` must be found in the classpath.

[Report a bug](#)

9.3.3. ENCRYPT Configured with Algorithms and Key Sizes

In this encryption mode, the coordinator selects the `secretKey` and distributes it to all peers. There is no keystore, and keys are distributed using a public/private key exchange. Instead, encryption occurs as follows:

1. The secret key is generated and distributed by the controller.
2. When a view change occurs, a peer requests the secret key by sending a key request with its own public key.
3. The controller encrypts the secret key with the public key, and sends it back to the peer.
4. The peer then decrypts and installs the key as its own secret key.

In this mode, the **ENCRYPT** layer must be placed above the **GMS** protocol in the configuration.

Example 9.9. ENCRYPT Layer

```
<config ... >
  <UDP />
  <PING />
  <MERGE2 />
  <FD />
  <VERIFY_SUSPECT />
  <ENCRYPT encrypt_entire_message="false"
           sym_init="128" sym_algorithm="AES/ECB/PKCS5Padding"
           asym_init="512" asym_algorithm="RSA"/>
  <pbcast.NAKACK />
  <UNICAST />
  <pbcast.STABLE />
  <FRAG2 />
  <pbcast.GMS />
</config>
```

In the provided example, the sequence numbers for the **NAKACK** and **UNICAST** protocols will be encrypted.

View changes that identify a new controller result in a new session key being generated and distributed to all peers. This is a substantial overhead in an application with high peer churn.

When encrypting an entire message, the message must be marshalled into a byte buffer before being encrypted, resulting in decreased performance.

[Report a bug](#)

9.3.4. ENCRYPT Configuration Parameters

The following table provides configuration parameters for the **ENCRYPT** JGroups protocol:

Table 9.1. Configuration Parameters

Name	Description
alias	Alias used for recovering the key. Change the default.

Name	Description
asymAlgorithm	Cipher engine transformation for asymmetric algorithm. Default is RSA.
asymInit	Initial public/private key length. Default is 512.
asymProvider	Cryptographic Service Provider. Default is Bouncy Castle Provider.
encrypt_entire_message	
id	Give the protocol a different ID if needed so we can have multiple instances of it in the same stack.
keyPassword	Password for recovering the key. Change the default.
keyStoreName	File on classpath that contains keystore repository.
level	Sets the logger level (see javadocs).
name	Give the protocol a different name if needed so we can have multiple instances of it in the same stack.
stats	Determines whether to collect statistics (and expose them via JMX). Default is true.
storePassword	Password used to check the integrity/unlock the keystore. Change the default. It is recommended that passwords are stored using Vault.
symAlgorithm	Cipher engine transformation for symmetric algorithm. Default is AES.
symInit	Initial key length for matching symmetric algorithm. Default is 128.

[Report a bug](#)

PART III. ADVANCED FEATURES IN RED HAT JBOSS DATA GRID

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

The advanced features explored in this guide are as follows:

- Transactions
- Marshalling
- Listeners and Notifications
- The Infinispan CDI Module
- MapReduce
- Distributed Execution
- Interoperability and Compatibility Mode

[Report a bug](#)

CHAPTER 10. TRANSACTIONS

10.1. ABOUT JAVA TRANSACTION API

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

JBoss Data Grid does the following for each cache operation:

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

[Report a bug](#)

10.2. TRANSACTIONS SPANNING MULTIPLE CACHE INSTANCES

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

[Report a bug](#)

10.3. THE TRANSACTION MANAGER

Use the following to obtain the `TransactionManager` from the cache:

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

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

Example 10.1. Performing Operations

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



NOTE

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

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


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

[Report a bug](#)

10.4. 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 10.1. Transaction Manager Lookup Classes

Class Name	Details
<code>org.infinispan.transaction.lookup.DummyTransactionManagerLookup</code>	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.
<code>org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup</code>	It is a fully functional JBoss Transactions based transaction manager that overcomes the functionality limits of the DummyTransactionManager .
<code>org.infinispan.transaction.lookup.GenericTransactionManagerLookup</code>	<code>GenericTransactionManagerLookup</code> is used by default when no transaction lookup class is specified. This lookup class is recommended when using JBoss Data Grid with Java EE-compatible environment that provides a <code>TransactionManager</code> interface, and is capable of locating the Transaction Manager in most Java EE application servers. If no transaction manager is located, it defaults to DummyTransactionManager .

It is important to note that when using Red Hat JBoss Data Grid with Tomcat or an ordinary Java Virtual Machine (JVM), the recommended Transaction Manager Lookup class is `JBossStandaloneJTAManagerLookup`, which uses JBoss Transactions.

[Report a bug](#)

CHAPTER 11. MARSHALLING

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

Red Hat JBoss Data Grid uses marshalling and unmarshalling to:

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

[Report a bug](#)

11.1. ABOUT MARSHALLING FRAMEWORK

Red Hat JBoss Data Grid uses the JBoss Marshalling Framework to marshal and unmarshal 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 marshal 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](#)

11.2. SUPPORT FOR NON-SERIALIZABLE OBJECTS

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

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



NOTE

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

[Report a bug](#)

11.3. HOT ROD AND MARSHALLING

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

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

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

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

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

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

[Report a bug](#)

11.4. CONFIGURING THE MARSHALLER USING THE REMOTECACHEMANAGER

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

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

Procedure 11.1. Define a Marshaller

1. Create a ConfigurationBuilder

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

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

2. Add a Marshaller Class

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

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

- Alternatively, specify a custom *Marshaller* instance.

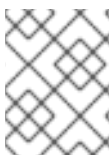
```
builder.marshaller(new GenericJBossMarshaller());
```

3. Start the RemoteCacheManager

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

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

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



NOTE

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

[Report a bug](#)

11.5. TROUBLESHOOTING

11.5.1. Marshalling Troubleshooting

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

Example 11.1. Exception 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.java:407)
at
org.infinispan.marshall.exts.ReplicableCommandExternalizer.writeObject(ReplicableCommandExternalizer.java:54)
at
org.infinispan.marshall.jboss.ConstantObjectTable$ExternalizerAdapter.writeObject(ConstantObjectTable.java:267)
at
org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.java:143)
at
org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.java:407)
at
org.infinispan.marshall.jboss.JBossMarshaller.objectToObjectStream(JBossMarshaller.java:167)
at
org.infinispan.marshall.VersionAwareMarshaller.objectToBuffer(VersionAwareMarshaller.java:92)
at
org.infinispan.marshall.VersionAwareMarshaller.objectToByteBuffer(VersionAwareMarshaller.java:170)
at
org.infinispan.marshall.VersionAwareMarshallerTest.testNestedNonSerializable(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
```

Messages starting with **in object** and stack traces are read in the same way: the highest **in object** message is the innermost one and the outermost **in object** message 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:

Example 11.2. Exceptions with Logging Levels Enabled

```
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:

Example 11.3. Unmarshalling Exceptions

```
java.io.IOException: Injected failue!
at
org.infinispan.marshall.VersionAwareMarshallerTest$1.readExternal(VersionAwareMarshallerTest.java:426)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadNewObject(RiverUnmarshaller.java:1172)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshaller.java:273)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshaller.java:210)
at
org.jboss.marshalling.AbstractUnmarshaller.readObject(AbstractUnmarshaller.java:85)
at
org.infinispan.marshall.jboss.JBossMarshaller.objectFromObjectStream(JBossMarshaller.java:210)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(VersionAwareMarshaller.java:104)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(VersionAwareMarshaller.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:

Example 11.4. Classloader Information

```

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/e
clipse-testng.jar
-
>...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/l
ib/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/easymockclassexension
/2.4/easymockclassexension-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/act
ivation-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.C
R4-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/1.1.3.4.0/xpp3_min-
1.1.3.4.0.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](#)

11.5.2. Other Marshalling Related Issues

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

Example 11.5. State Generator Log

```

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(Stat
eTransferManagerImpl.java:175)
    at
org.infinispan.remoting.InboundInvocationHandlerImpl.generateState(Inbou
ndInvocationHandlerImpl.java:119)
    at
org.infinispan.remoting.transport.jgroups.JGroupsTransport.getState(JGro
upsTransport.java:586)
    at
org.jgroups.blocks.MessageDispatcher$ProtocolAdapter.handleUpEvent(Messa
geDispatcher.java:691)
    at
org.jgroups.blocks.MessageDispatcher$ProtocolAdapter.up(MessageDispatche
r.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$StateProviderHandl
er.process(STREAMING_STATE_TRANSFER.java:653)
    at
org.jgroups.protocols.pbcast.STREAMING_STATE_TRANSFER$StateProviderThrea
dSpawner$1.run(STREAMING_STATE_TRANSFER.java:582)
    at
java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecuto
r.java:886)
    at
java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.ja
va: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.acquireProcess
ingLock(JGroupsDistSync.java:71)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.generateTransactio
nLog(StateTransferManagerImpl.java:202)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.generateState(Stat
eTransferManagerImpl.java:165)
    ... 12 more

```

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

Example 11.6. *EOFException*

```

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(AbstractUnmarshalle
r.java:184)
    at
org.jboss.marshalling.AbstractUnmarshaller.readUnsignedByteDirect(Abstra
ctUnmarshaller.java:319)
    at
org.jboss.marshalling.AbstractUnmarshaller.readUnsignedByte(AbstractUnma
rshaller.java:280)
    at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarsha
ller.java:207)
    at
org.jboss.marshalling.AbstractUnmarshaller.readObject(AbstractUnmarshall
er.java:85)
    at

```



```
org.infinispan.marshall.jboss.GenericJBossMarshaller.objectFromObjectStream(GenericJBossMarshaller.java:175)
    at
org.infinispan.marshall.VersionAwareMarshaller.objectFromObjectStream(VersionAwareMarshaller.java:184)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.processCommitLog(StateTransferManagerImpl.java:228)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.applyTransactionLog(StateTransferManagerImpl.java:250)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.applyState(StateTransferManagerImpl.java:320)
    at
org.infinispan.remoting.InboundInvocationHandlerImpl.applyState(InboundInvocationHandlerImpl.java:102)
    at
org.infinispan.remoting.transport.jgroups.JGroupsTransport.setState(JGroupsTransport.java:603)
    ...
```

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

[Report a bug](#)

CHAPTER 12. LISTENERS AND NOTIFICATIONS

12.1. THE NOTIFICATION/LISTENER API

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

[Report a bug](#)

12.2. LISTENER NOTIFICATIONS

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

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

[Report a bug](#)

12.2.1. About Cache-level Notifications

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

[Report a bug](#)

12.2.2. Cache Manager-level Notifications

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

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

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

[Report a bug](#)

12.2.3. About Synchronous and Asynchronous Notifications

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

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

Annotate listeners using the following:

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

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

[Report a bug](#)

12.3. MODIFYING CACHE ENTRIES

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

[Report a bug](#)

12.3.1. Cache Entry Modified Listener Configuration

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

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

[Report a bug](#)

12.3.2. Listener Example

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

Example 12.1. Configuring a Listener

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

[Report a bug](#)

12.3.3. Cache Entry Modified Listener Example

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

Example 12.2. Modified Listener

```
@Listener
public class PrintWhenModified {
    @CacheEntryModified
    public void print(CacheEntryModifiedEvent event) {
        System.out.println("Cache entry modified. Details = " +
event");
    }
}
```

[Report a bug](#)

12.4. NOTIFYINGFUTURES

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

**NOTE**

NotifyingFutures are only available in JBoss Data Grid Library mode.

[Report a bug](#)

12.4.1. NotifyingFutures Example

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

Example 12.3. Configuring NotifyingFutures

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

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

[Report a bug](#)

CHAPTER 13. THE INFINISPAN CDI MODULE

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

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

[Report a bug](#)

13.1. USING INFINISPAN CDI

13.1.1. Infinispan CDI Prerequisites

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

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

[Report a bug](#)

13.1.2. Set the CDI Maven Dependency

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

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

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

[Report a bug](#)

13.2. USING THE INFINISPAN CDI MODULE

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

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

[Report a bug](#)

13.2.1. Configure and Inject Infinispan Caches

13.2.1.1. Inject an Infinispan Cache

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

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

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

[Report a bug](#)

13.2.1.2. Inject a Remote Infinispan Cache

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

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

[Report a bug](#)

13.2.1.3. Set the Injection's Target Cache

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

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

[Report a bug](#)

13.2.1.3.1. Create a Qualifier Annotation

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

Example 13.1. Custom Cache Qualifier

```
@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](#)

13.2.1.3.2. Add a Producer Class

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

Example 13.2. Using the `@SmallCache` Qualifier

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

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](#)

13.2.1.3.3. Inject the Desired Class

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

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

[Report a bug](#)

13.2.2. Configure Cache Managers with CDI

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

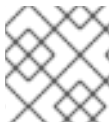
[Report a bug](#)

13.2.2.1. Specify the Default Configuration

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

Example 13.3. Specifying the Default Configuration

```
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](#)

13.2.2.2. Override the Creation of the Embedded Cache Manager

Prerequisites

[Section 13.2.2.1, “Specify the Default Configuration”](#)

Creating Non Clustered Caches

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

Example 13.4. Create a Non Clustered Cache

```
public class Config {
    @Produces
    @ApplicationScoped
```



```

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

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

```

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

Creating Clustered Caches

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

Example 13.5. Create Clustered Caches

```

public class Config {

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

```

Invoke the Method to Generate an `EmbeddedCacheManager`

The method annotated with `@Produces` in the non clustered method generates `Configuration` objects. The methods in the clustered cache example annotated 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.

Example 13.6. Generate an `EmbeddedCacheManager`

```

...
@Inject
EmbeddedCacheManager cacheManager;

```



[Report a bug](#)

13.2.2.3. Configure a Remote Cache Manager

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

Example 13.7. Configuring the Remote Cache Manager

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

[Report a bug](#)

13.2.2.4. Configure Multiple Cache Managers with a Single Class

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

Example 13.8. Configure Multiple Cache Managers

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

    @Produces
    @ApplicationScoped
    @DefaultClustered
    public org.infinispan.manager.EmbeddedCacheManager
    defaultClusteredCacheManager() {
        GlobalConfiguration g = new GlobalConfigurationBuilder()
        .clusteredDefault()
        .transport()
        .clusterName("InfinispanCluster")
    }
}
```

```

        .build();
    Configuration cfg = new ConfigurationBuilder()
        .eviction()
        .strategy(EvictionStrategy.LRU)
        .maxEntries(150)
        .build();
    return new DefaultCacheManager(g, cfg);
}

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

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

```

[Report a bug](#)

13.2.3. Storage and Retrieval Using CDI Annotations

13.2.3.1. Configure Cache Annotations

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

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

[Report a bug](#)

13.2.3.2. Enable Cache Annotations

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

Example 13.9. Adding Interceptors

```

<beans xmlns="http://java.sun.com/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>

```

**NOTE**

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

[Report a bug](#)

13.2.3.3. Caching the Result of a Method Invocation

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

```

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

```

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

```

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

```

```

    celsius = toCelsiusFormatted(f);
    fahrenheitToCelsiusCache.put(f, celsius);
}

```

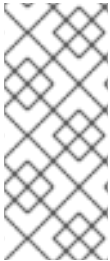
In such cases, the Infinispan CDI module can be used to eliminate all the extra code in the related examples. Annotate the method with the `@CacheResult` annotation instead, as follows:

```

@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](#)

13.2.3.3.1. Specify the Cache Used

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

```

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

```

[Report a bug](#)

13.2.3.3.2. Cache Keys for Cached Results

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

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

Example 13.10. Create a Custom Key

```

@CacheResult
public String doSomething
    (@CacheKey String p1,
    @CacheKey String p2,

```

```
String dontCare) {
    ...
}
```

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

[Report a bug](#)

13.2.3.3.3. 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](#)

13.2.4. Cache Operations

13.2.4.1. Update a Cache Entry

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

Example 13.11. Sample @CachePut Annotated Method

```
import javax.cache.annotation.CachePut
@CachePut (cacheName = "personCache")
public void updatePerson
    (@CacheKey long personId,
    @CacheValue Person newPerson) {
    ...
}
```

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

See Also:

- [Section 13.2.3.3.2, “Cache Keys for Cached Results”](#)

[Report a bug](#)

13.2.4.2. Remove an Entry from the Cache

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

Example 13.12. Removing an Entry from the Cache

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

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

[Report a bug](#)

13.2.4.3. Clear the Cache

Invoke the `@CacheRemoveAll` method to clear all entries from the cache.

Example 13.13. Clear All Entries from the Cache with @CacheRemoveAll

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



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

[Report a bug](#)

CHAPTER 14. MAPREDUCE

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

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

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

1. The user initiates a task on a cache instance, which runs on a cluster node (the master node).
2. The master node receives the task input, divides the task, and sends tasks for map phase execution on the grid.
3. Each node executes a **Mapper** function on its input, and returns intermediate results back to the master node.
 - If the *useIntermediateSharedCache* parameter is set to "true", the map results are inserted in an intermediary cache, rather than being returned to the master node.
 - If a **Combiner** has been specified with *task.combinedWith(combiner)*, the **Combiner** is called on the **Mapper** results and the combiner's results are returned to the master node or inserted in the intermediary cache.



NOTE

Combiners are not required but can only be used when the function is both commutative (changing the order of the operands does not change the results) and associative (the order in which the operations are performed does not matter as long as the sequence of the operands is not changed). Combiners are advantageous to use because they can improve the speeds of MapReduceTask executions.

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. Optionally specify the target cache for the results using the instructions in [Section 14.1.2, "Specify the Target Cache"](#).
 - 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.

- o If no target cache is specified and no collator is specified (using `task.execute(Collator)`), the result map is returned to the master node.

[Report a bug](#)

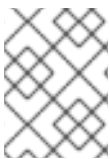
14.1. THE MAPREDUCE API

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

- **Mapper**
- **Reducer**
- **Collator**
- **MapReduceTask**
- **Combiners**

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

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



NOTE

The **MapReduceTask** requires a **Mapper** and a **Reducer** but using a **Collator** or **Combiner** is optional.

Example 14.1. Executing the Mapper

```
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.

Example 14.2. Reducer

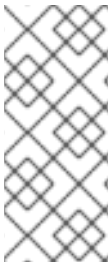
```
public interface Reducer<KOut, VOut> extends Serializable {
    /**
     * Combines/reduces all intermediate values for a particular
     * intermediate key to a single value.
     */
}
```

```

* <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.



NOTE

Combiners are not required but can only be used when the function is both commutative (changing the order of the operands does not change the results) and associative (the order in which the operations are performed does not matter as long as the sequence of the operands is not changed). Combiners are advantageous to use because they can improve the speeds of **MapReduceTask** executions.

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

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

Example 14.3. Assembling the Result

```

public interface Collator<KOut, VOut, R> {
    /**
     * Collates all reduced results and returns R to invoker of
     * distributed task.
     *
     * @return final result of distributed task computation
     */
    R collate(Map<KOut, VOut> reducedResults);
}

```

[Report a bug](#)

14.1.1. MapReduceTask

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

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

Example 14.4. Specifying MapReduceTask Components

```
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.



NOTE

The default timeout for **MapReduceTask** is 0 (zero). That is, the task will wait indefinitely for its completion by default.

[Report a bug](#)

14.1.2. Specify the Target Cache

Red Hat JBoss Data Grid's **MapReduce** implementation allows users to specify a target cache to store the results of an executed task. The results are available after the `execute` method (which is synchronous) is complete.

This variant of the `execute` method prevents the master JVM node from exceeding its allowed maximum heap size. This is especially relevant if objects that are the results of the reduce phase have a large memory footprint or if multiple **MapReduceTasks** are concurrently executing on the master task node.

Use the following method of **MapReduceTask** to specify a **Cache** object to store the results:

```
public void execute(Cache<KOut, VOut> resultsCache) throws CacheException
```

Use the following method of **MapReduceTask** to specify a name for the target cache:

```
public void execute(String resultsCache) throws CacheException
```

[Report a bug](#)

14.1.3. Mapper and CDI

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

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

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

Example 14.5. Using a CDI Injection

```
public class WordCountCacheInjectorMapper 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](#)

14.2. MAPREDUCETASK DISTRIBUTED EXECUTION

MapReduceTask is a distributed task that allows a large-scale computation to be transparently parallelized across Red Hat JBoss Data Grid cluster nodes. **MapReduceTask** can be instantiated with a reference to a cache containing data that is used as input for this task. JBoss Data Grid's execution environment can migrate and execute instances of provided **Mapper** and **Reducer** seamlessly across JBoss Data Grid nodes.

MapReduceTask distributed execution distributes the reduce phase execution. Previously, the reduce phase was performed on a single master task node. This limitation has been removed, and the reduce phase execution can now be distributed across the cluster also. The distribution of reduce phase is achieved by relying on consistent hashing.

It is still possible to use MapReduceTask with the reduce phase performed on a single node, and this is recommended for smaller input tasks.

Distributed Execution of the MapReduceTask occurs in three phases:

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

Map Phase

MapReduceTask hashes task input keys and groups them by the execution node that they are hashed to. After key node mapping, MapReduceTask sends a map function and inputs keys to each node. The map function is invoked using given keys and locally loaded corresponding values.

Results are collected with a Red Hat JBoss Data Grid supplied Collector, and the combine phase is initiated. A Combiner, if specified, takes KOut keys and immediately invokes the reduce phase on keys. The result of mapping phase executed on each node is KOut/VOut map. There is one resulting map per execution node per launched MapReduceTask.

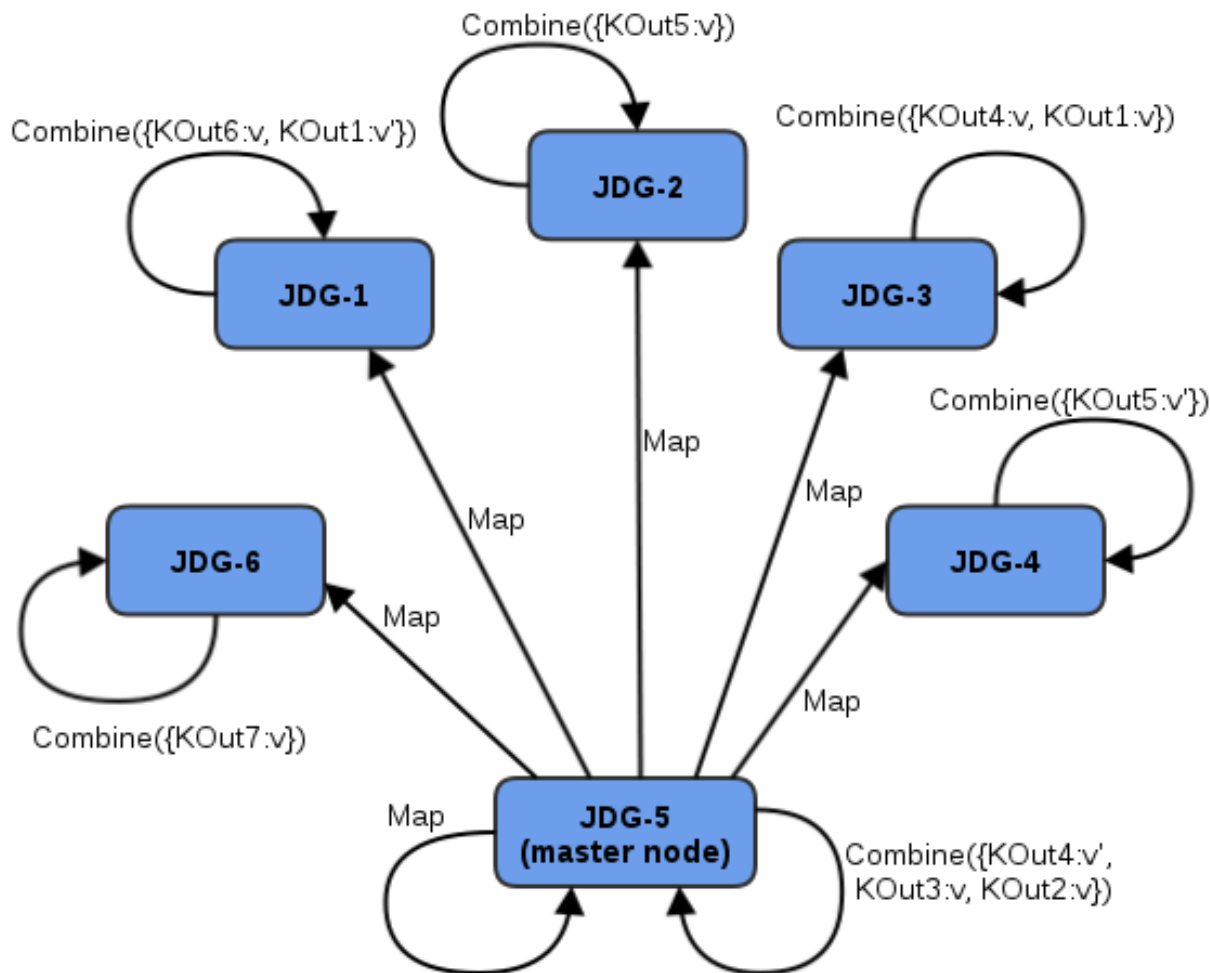


Figure 14.1. Map Phase

Intermediate KOut/VOut migration phase

In order to proceed with reduce phase, all intermediate keys and values must be grouped by

intermediate KOut keys. As map phases around the cluster can produce identical intermediate keys, all identical intermediate keys and their values must be grouped before reduce is executed on any particular intermediate key.

At the end of the combine phase, each intermediate KOut key is hashed and migrated with its VOut values to the JBoss Data Grid node where keys KOut are hashed to. This is achieved using a temporary distributed cache and underlying consistent hashing mechanism.

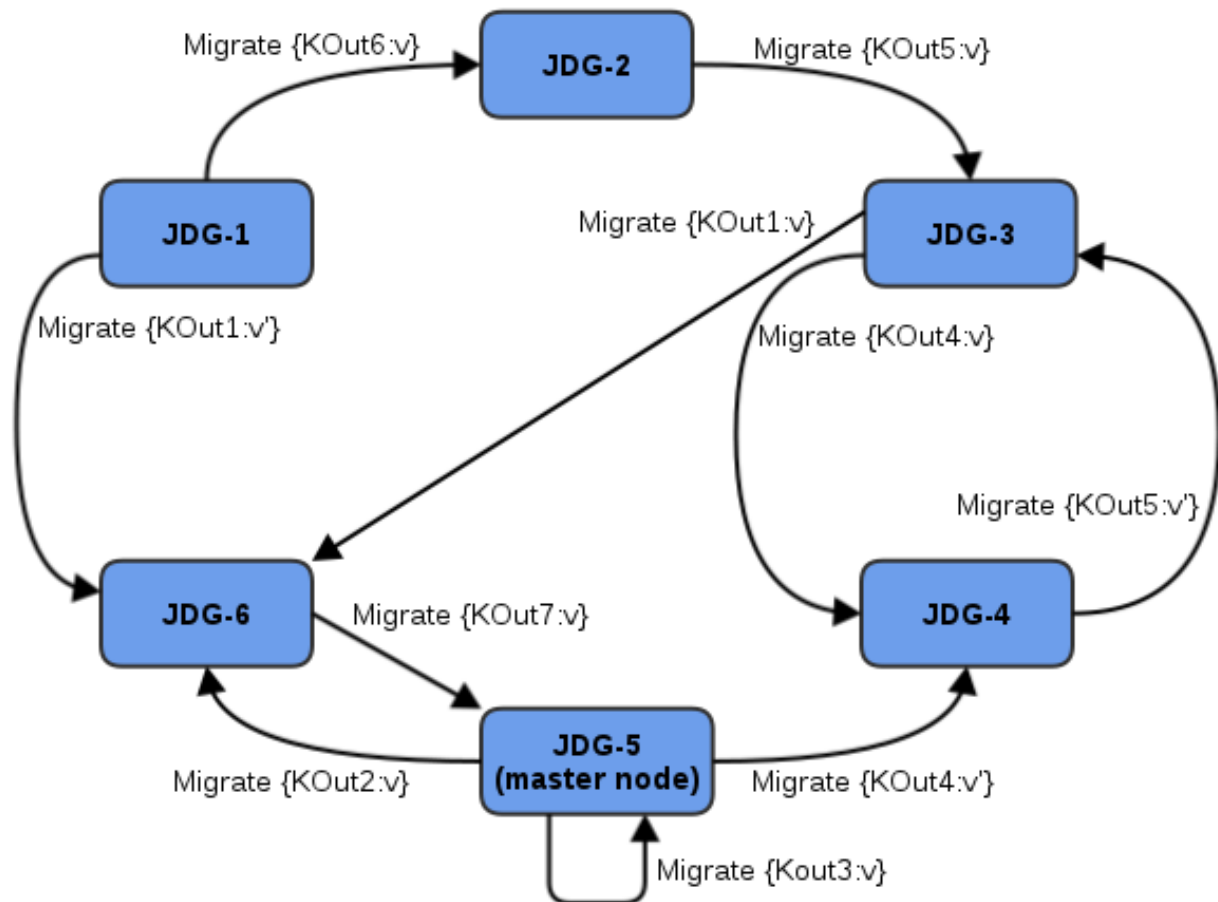


Figure 14.2. Kout/VOut Migration

Once Map and Combine phase have finished execution, a list of KOut keys is returned to a master node and it is initiating MapReduceTask. VOut values are not returned as they are not required at the master task node. MapReduceTask is ready to start with reduce phase.

Reduce Phase

To complete reduce phase, MapReduceTask groups KOut keys by execution node N they are hashed to. For each node and its grouped input KOut keys, MapReduceTask sends a reduce command to a node where KOut keys are hashed. Once the reduce command is executed on the target execution node, it locates the temporary cache belonging to MapReduce task. For each KOut key, the reduce command obtains a list of VOut values, wraps it with an Iterator, and invokes reduce on it.

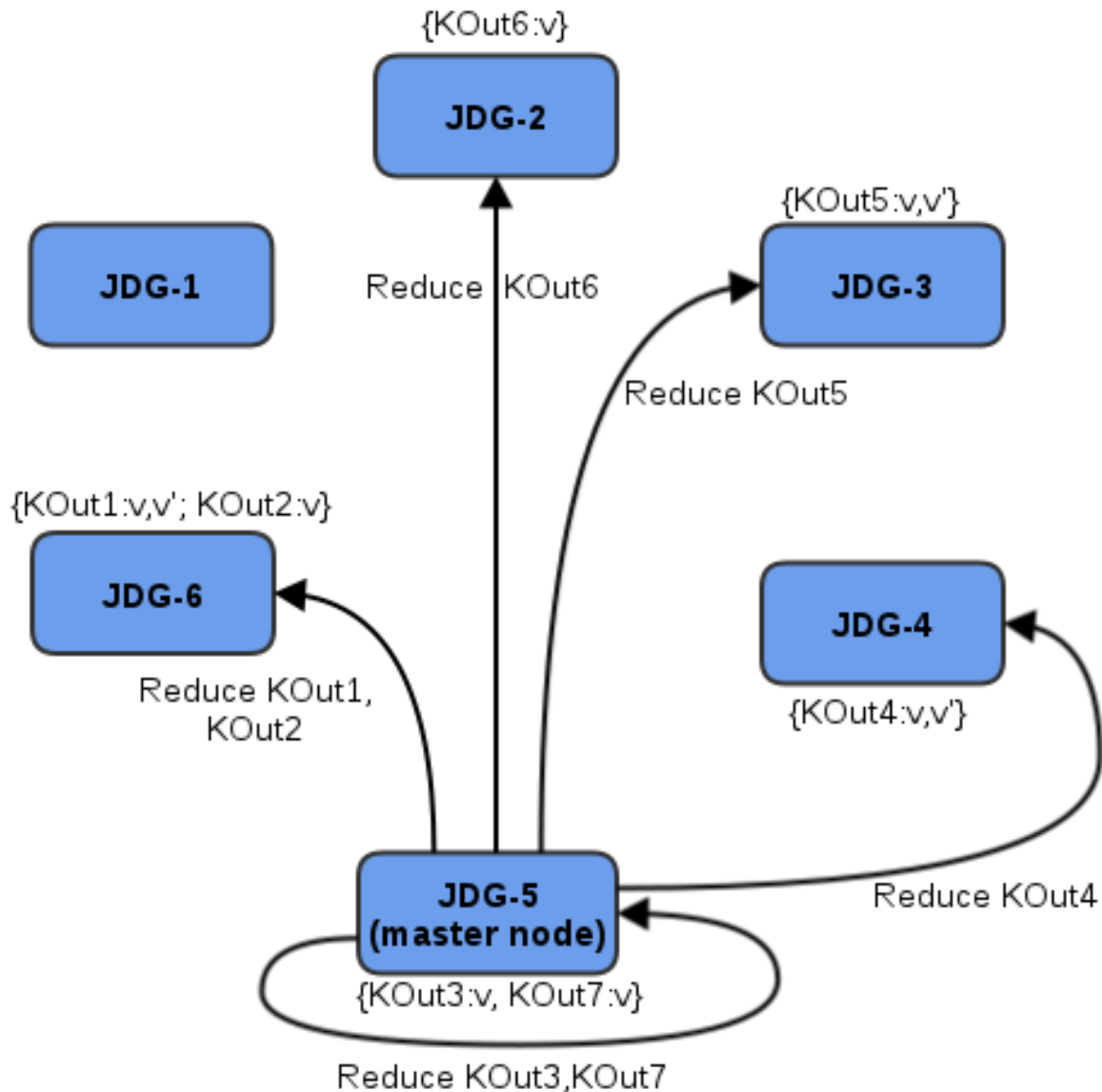


Figure 14.3. Reduce Phase

The result of each reduce is a map where each key is KOut and value is VOut. Each JBoss Data Grid execution node returns one map with KOut/VOut result values. As all initiated reduce commands return to a calling node, MapReduceTask combines all resulting maps into a map and returns the map as a result of MapReduceTask.

Distributed reduce phase is enabled by using a MapReduceTask constructor specifying the cache to use as input data for the task and boolean parameter *distributeReducePhase* set to `true`. For more information, see the Map/Reduce section of the *Red Hat JBoss Data Grid API Documentation*

[Report a bug](#)

14.3. MAP REDUCE EXAMPLE

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

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

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

All words that appear in all sentences must be counted.

Example 14.6. Implementing the Distributed Task

```
public class WordCountExample {

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

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

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

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

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

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

```

        collector.emit(token, 1);
    }
}

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

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

```

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

Example 14.7. Defining the *Collator*

```

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);

```

CHAPTER 15. DISTRIBUTED EXECUTION

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

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

In JBoss Data Grid's distributed execution framework:

- Each `DistributedExecutorService` is bound to a single cache. Tasks submitted have access to key/value pairs from that particular cache if the task submitted is an instance of `DistributedCallable`.
- Every `Callable`, `Runnable`, and/or `DistributedCallable` submitted must be either `Serializable` or `Externalizable`, in order to prevent task migration to other nodes each time one of these tasks is performed. The value returned from a `Callable` must also be `Serializable` or `Externalizable`.

[Report a bug](#)

15.1. DISTRIBUTEDCALLABLE API

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

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

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

Example 15.1. Using the `DistributedCallable` API

```
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](#)

15.2. CALLABLE AND CDI

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

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

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

1. Declare a `Cache` field in `Callable` and annotate it with `org.infinispan.cdi.Input`
2. Include the mandatory `@Inject` annotation.

Example 15.2. Using Callable and the CDI

```
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](#)

15.3. DISTRIBUTED TASK FAILOVER

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

- Failover due to a node failure where a task is executing.
- Failover due to a task failure; for example, if a `Callable` task throws an exception.

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

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

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

Example 15.3. Random Failover Execution Policy

```
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.

Example 15.4. Distributed Task Failover Policy Interface

```
/**
 * 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](#)

15.4. DISTRIBUTED TASK EXECUTION POLICY

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

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

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

Example 15.5. Using Rack Nodes to Execute a Specific Task

```
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](#)

15.5. DISTRIBUTED EXECUTION EXAMPLE

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

1. As shown below, the area of a square is:

$$\text{Area of a Square (S)} = 4r^2$$

2. The following is an equation for the area of a circle:

$$\text{Area of a Circle (C)} = \pi \times r^2$$

3. Isolate r^2 from the first equation:

$$r^2 = S/4$$

4. Inject this value of r^2 into the second equation to find a value for Pi:

$$C = S\pi/4$$

5. Isolating π in the equation results in:

$$C = S\pi/4$$

$$4C = S\pi$$

$$4C/S = \pi$$

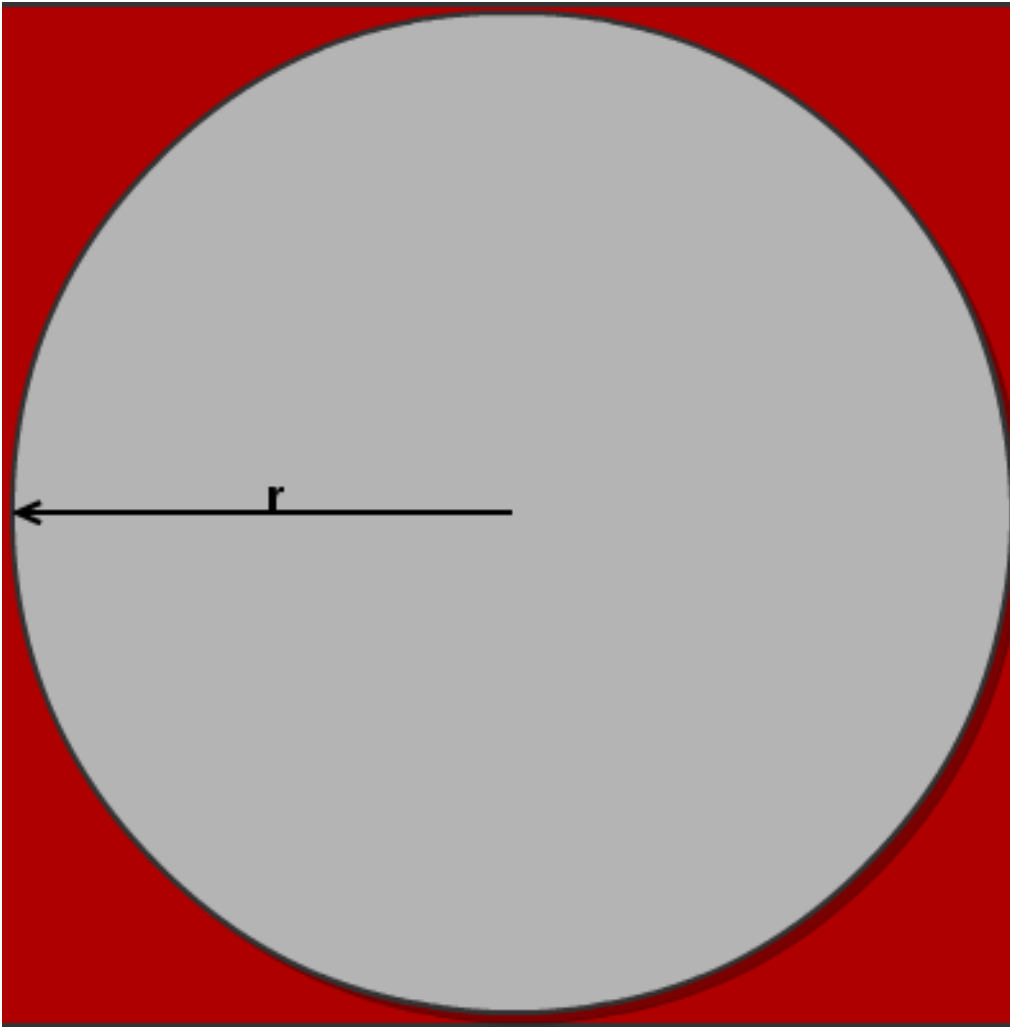


Figure 15.1. Distributed Execution Example

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

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

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

Example 15.6. Distributed Execution Example

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

        int numPoints = 100000000;
        int numServers = caches.size();
        int numberPerWorker = numPoints / numServers;

        DistributedExecutorService des = new
        DefaultExecutorService(cache);
        long start = System.currentTimeMillis();
        CircleTest ct = new CircleTest(numberPerWorker);
```

```
List<Future<Integer>> results = des.submitEverywhere(ct);
int countCircle = 0;
for (Future<Integer> f : results) {
    countCircle += f.get();
}
double appxPi = 4.0 * countCircle / numPoints;

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

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

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

    private final int loopCount;

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

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

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

[Report a bug](#)

CHAPTER 16. DATA INTEROPERABILITY

16.1. INTEROPERABILITY BETWEEN LIBRARY AND REMOTE CLIENT-SERVER ENDPOINTS

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

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

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

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

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

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

[Report a bug](#)

16.2. USING COMPATIBILITY MODE

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

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

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

[Report a bug](#)

16.3. PROTOCOL INTEROPERABILITY

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

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

Enable Compatibility Mode declaratively in Client-Server mode

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

Example 16.1. Compatibility Mode Enabled

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

Enable Compatibility Mode programmatically in Library mode

Use a `configurationBuilder` with the compatibility mode enabled as follows:

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

Enable Compatibility Mode declaratively in Library mode

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

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

[Report a bug](#)

16.3.1. Use Cases and Requirements

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

Table 16.1. Compatibility Mode Use Cases

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

Use Case	Client A (Reader or Writer)	Client B (Write/Read Counterpart of Client A)
7	Memcached	Hot Rod C++

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

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

The following steps apply to all use cases:

Client A Side

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

Client B Side

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



NOTE

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

[Report a bug](#)

16.3.2. Protocol Interoperability Over REST

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

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

[Report a bug](#)

APPENDIX A. REVISION HISTORY

Revision 6.3.0-20	Mon Jan 12 2015	Gemma Sheldon
BZ-1021819: Updated ENCRYPT configuration examples to include Vault passwords.		
Revision 6.3.0-19	Mon Jan 05 2015	Gemma Sheldon
BZ-1176192: Updated JGroups SASL configuration example to include both server and client callback handler classes in cluster transport security example for library mode.		
BZ-1021819: Updated AUTH configuration examples with VAULT, required to prevent passwords being stored as plain text.		
Revision 6.3.0-18	Fri Dec 05 2014	Misha Husnain Ali
Updated for 6.3.2.		
Revision 6.3.0-17	Wed Dec 03 2014	Gemma Sheldon
BZ-1163946: Clarified implementation of CallbackHandlers.		
Revision 6.3.0-16	Mon Nov 17 2014	Misha Husnain Ali
BZ-1162244: Removed a chapter that was not relevant for developers.		
Revision 6.3.0-15	Fri Nov 07 2014	Gemma Sheldon
BZ-1159714: Added AUTH permission to list of available security permissions.		
Revision 6.3.0-14	Wed Oct 22 2014	Gemma Sheldon
BZ-1154943: Updated configuration example for Node Authorization in Library Mode.		
Revision 6.3.0-13	Wed Oct 08 2014	Gemma Sheldon
BZ-1148295: Included specific information about jgroups configuration file location for ENCRYPT.		
BZ-1151317: Changed name of clusteringRealm example.		
Revision 6.3.0-12	Wed Sep 10 2014	Rakesh Ghatvisave
BZ-1128958: Added note admonition for MapReduceTask timeout.		
BZ-1123898: Added topic about LDAP roles.		
Revision 6.3.0-11	Tue Sep 02 2014	Misha Husnain Ali
BZ-1136108: Fixed typo in chapter name.		
BZ-1119510: Changes to code.		
Revision 6.3.0-10	Tue Aug 26 2014	Gemma Sheldon
BZ-1119510: Added EXTERNAL auth certificate topic back to security information.		
BZ-1109191: Multiple changes to structure and content of Chapters 1-2. Added section 2.3.		
Revision 6.3.0-9	Mon Aug 18 2014	Rakesh Ghatvisave
BZ-1127098: Modified Audit log server configuration and Audit logger programmatic config Library mode.		
BZ-1127098: Modified Audit log server config snippet.		
Revision 6.3.0-8	Mon Aug 11 2014	Rakesh Ghatvisave
BZ-1127098: Commented out incomplete Audit log server config topic for async release.		
BZ-1127098: Added Audit log server configuration topic.		
BZ-1128956: Added a translation manager topic back in.		
Revision 6.3.0-7	Mon Aug 04 2014	Misha Husnain Ali
BZ-1126031: Added start information to permissions.		
BZ-1119746: Rolled back content for jcache.		
BZ-1114813: Updated MapReduceTask Distributed Execution topic with diagrams and clearer description.		
Revision 6.3.0-6	Wed Jul 02 2014	Rakesh Ghatvisave

