Red Hat Data Grid 7.2

Developer Guide

For use with Red Hat JBoss Data Grid 7.2

Last Updated: 2020-05-06

Red Hat Data Grid 7.2 Developer Guide

For use with Red Hat JBoss Data Grid 7.2

Legal Notice

Abstract

An advanced guide intended for developers using Red Hat JBoss Data Grid 7.2

Table of Contents

PART I. PROGRAMMABLE APIS

CHAPTER 1. PROGRAMMABLE APIS

Red Hat JBoss Data Grid provides the following programmable APIs:

- Cache
- AdvancedCache
- MultimapCache
- Asynchronous
- Batching
- Grouping
- Persistence (formerly CacheStore)
- ConfigurationBuilder
- Externalizable
- Notification (also known as the Listener API because it deals with Notifications and Listeners)
- JSR-107 (JCache)
- Health Check
- REST

CHAPTER 2. THE CACHE API

2.1. 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 the Red Hat JBoss Data Grid cache.

JBoss Data Grid in Library, or Embedded, Mode

Use the **org.infinispan.Cache** API.

JBoss Data Grid in Remote Client-Server Mode

Use the **org.infinispan.client.hotrod.RemoteCache** API.
The **RemoteCache** interface implements the **Cache** API but does not support some operations given the difference between remote and local operations.

2.2. 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**:

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);
```

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

3. 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. 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();
```

2.3. PER-INVOCATION FLAGS

2.3.1. Per-Invocation Flags

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

2.3.2. Per-Invocation Flag Functions

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

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

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

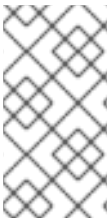
A detailed list of all flags available for JBoss Data Grid is in the JBoss Data Grid API Documentation's Flag class.

2.3.3. 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.

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.

2.3.4. 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.

Using the IGNORE_RETURN_VALUES Flag

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

2.4. THE ADVANCEDCACHE INTERFACE

2.4.1. 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();
```

2.4.2. 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.

Applying Flags to a Cache Invocation

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

2.4.3. GET and PUT Usage in Distribution Mode

2.4.3.1. GET and PUT Usage in Distribution Mode

In distribution mode, the cache performs a remote **GET** command before a write command. This occurs because certain methods (for example, **Cache.put()**) return the previous value associated with the specified key according to the **java.util.Map** contract. When this is performed on an instance that does not own the key and the entry is not found in the L1 cache, the only reliable way to elicit this return value is to perform a remote **GET** before the **PUT**.

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

2.4.3.2. Distributed GET and PUT Operation Resource Usage

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

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

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

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

2.4.4. 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.

The listed methods have a significant impact on performance. As a result, it is recommended that these methods are used for informational and debugging purposes only.

Performance Concerns

In JBoss Data Grid 7.2 the map methods **size()**, **values()**, **keySet()**, and **entrySet()** include entries in the cache loader by default. The cache loader in use will determine the performance of these commands; for instance, when using a database these methods will run a complete scan of the table where data is stored, which may result in slower processing. To not load entries from the cache loader, and avoid any potential performance hit, use **Cache.getAdvancedCache().withFlags(Flag.SKIP_CACHE_LOAD)** before executing the desired method.

Understanding the size() Method (Embedded Caches)

In JBoss Data Grid 7.2 the **Cache.size()** method provides a count of all elements in both this cache and cache loader across the entire cluster. When using a loader or remote entries, only a subset of entries is held in memory at any given time to prevent possible memory issues, and the loading of all entries may be slow.

In this mode of operation, the result returned by the **size()** method is affected by the flags **org.infinispan.context.Flag#CACHE_MODE_LOCAL**, to force it to return the number of entries present on the local node, and **org.infinispan.context.Flag#SKIP_CACHE_LOAD**, to ignore any passivated entries. Either of these flags may be used to increase performance of this method, at the cost of not returning a count of all elements across the entire cluster.

Understanding the size() Method (Remote Caches)

In JBoss Data Grid 7.2 the Hot Rod protocol contain a dedicated **SIZE** operation, and the clients use this operation to calculate the size of all entries.

CHAPTER 3. THE MULTIMAP CACHE

3.1. THE MULTIMAP CACHE

The **MultimapCache** is a cache that maps keys to values in which each key can contain multiple values. It currently only functions in Library Mode.

3.2. INSTALLING MULTIMAPCACHE USING MAVEN

To make the **MultimapCache** available in the Maven project configure the **pom.xml** as follows:

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-multimap</artifactId>
  <version>...</version> <!-- 7.2.0 or later -->
</dependency>
```

3.3. CREATING A MULTIMAP CACHE

Create a **MultimapCache** using code like the following:

```
// create or obtain your EmbeddedCacheManager
EmbeddedCacheManager cm = ... ;

// create or obtain a MultimapCacheManager passing the EmbeddedCacheManager
MultimapCacheManager multimapCacheManager =
EmbeddedMultimapCacheManagerFactory.from(cm);

// define the configuration for the multimap cache
multimapCacheManager.defineConfiguration(multimapCacheName, c.build());

// get the multimap cache
multimapCache = multimapCacheManager.get(multimapCacheName);
```

3.4. EXAMPLE MULTIMAPCACHE USAGE

Below is code demonstrating how to use **MultimapCache**:

```
MultimapCache<String, String> multimapCache = ...;

multimapCache.put("girlNames", "marie")
        .thenCompose(r1 -> multimapCache.put("girlNames", "oihana"))
        .thenCompose(r3 -> multimapCache.get("girlNames"))
        .thenAccept(names -> {
                if(names.contains("marie"))
                    System.out.println("Marie is a girl name");

                 if(names.contains("oihana"))
                    System.out.println("Oihana is a girl name");
               });
```

-

CHAPTER 4. THE ASYNCHRONOUS API

4.1. THE ASYNCHRONOUS API

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

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

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

4.2. ASYNCHRONOUS API BENEFITS

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

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

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

Using the Asynchronous API

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

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

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

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

4.3. ABOUT ASYNCHRONOUS PROCESSES

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

- Network calls
- Marshalling
- Writing to a cache store (optional)

- Locking

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

4.4. RETURN VALUES AND THE ASYNCHRONOUS API

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

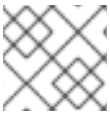
Call the following operation to obtain the result of an asynchronous operation. This operation blocks threads when called.

```
Future.get()
```

CHAPTER 5. THE BATCHING API

5.1. 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 may only be used in Red Hat JBoss Data Grid's Library Mode.

5.2. 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.

5.3. 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.

5.4. USING THE BATCHING API

5.4.1. Configure the Batching API

To use the Batching API, enable invocation batching in the cache configuration, as seen in the following example:

```
Configuration c = new
ConfigurationBuilder().transaction().transactionMode(TransactionMode.TRANSACTIONAL).invocationB
atching().enable().build();
```

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

5.4.2. 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();
```

Without Using Batch

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

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

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 applied.

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

CHAPTER 6. THE GROUPING API

6.1. 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.

6.2. 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.

6.3. 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.

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.

6.4. CONFIGURE THE GROUPING API

6.4.1. 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 Specify an Intrinsic Group and Specify an Extrinsic Group .
3. Register all specified groupers.

6.4.2. Enable Groups

The first step to set up the Grouping API is to enable groups. The following example demonstrates how to enable Groups:

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

6.4.3. 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:

Specifying an Intrinsic Group Example

```
class User {

  <!-- Additional configuration information here -->
  String office;
  <!-- Additional configuration information here -->

  public int hashCode() {
    // Defines the hash for the key, normally used to determine location
    <!-- Additional configuration information here -->
  }

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

}
```

6.4.4. 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.

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;
    }

}
```

6.4.5. Register Groupers

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

Programmatically Register a Grouper

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

CHAPTER 7. THE PERSISTENCE SPI

7.1. THE PERSISTENCE SPI

In Red Hat JBoss Data Grid, persistence can configure external (persistent) storage engines. These storage engines complement Red Hat 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.

NOTE

Programmatically configuring persistence can only be accomplished in Red Hat JBoss Data Grid's Library Mode.

7.2. 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 deserialized (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.

7.3. PROGRAMMATICALLY CONFIGURE THE PERSISTENCE SPI

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

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);
```

7.4. PERSISTENCE EXAMPLES

7.4.1. Persistence Examples

The following examples demonstrate how to configure various cache stores implementations programmatically. For a comparison of these stores, along with additional information on each, refer to the Administration and Configuration Guide.

7.4.2. Configure the Cache Store Programmatically

The following example demonstrates how to configure the cache store programmatically:

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
   .passivation(false)
   .addSingleFileStore()
      .shared(false)
      .preload(true)
      .fetchPersistentState(true)
      .purgeOnStartup(false)
      .location(System.getProperty("java.io.tmpdir"))
      .async()
        .enabled(true)
        .threadPoolSize(5)
      .singleton()
        .enabled(true)
        .pushStateWhenCoordinator(true)
        .pushStateTimeout(20000);
```

NOTE

This configuration is for a single-file cache store. Some attributes, such as **location** are specific to the single-file cache store and are not used for other types of cache stores.

Configure the Cache store Programatically

1. Use the **ConfigurationBuilder** to create a new configuration object.

2. The **passivation** elements affects the way Red Hat JBoss Data Grid interacts with stores. Passivation removes an object from an in-memory cache and writes it to a secondary data store,

such as a system or database. If no secondary data store exists, then the object will only be removed from the in-memory cache. Passivation is **false** by default.

3. The **addSingleFileStore()** elements adds the SingleFileStore as the cache store for this configuration. It is possible to create other stores, such as a JDBC Cache Store, which can be added using the **addStore** method.

4. The **shared** parameter indicates that the cache store is shared by different cache instances. For example, where all instances in a cluster use the same JDBC settings to talk to the same remote, shared database. **shared** is **false** by default. When set to **true**, it prevents duplicate data being written to the cache store by different cache instances.

5. The **preload** element is set to **false** by default. When set to **true** the data stored in the cache store is preloaded into the memory when the cache starts. This allows data in the cache store to be available immediately after startup and avoids cache operations delays as a result of loading data lazily. Preloaded data is only stored locally on the node, and there is no replication or distribution of the preloaded data. JBoss Data Grid will only preload up to the maximum configured number of entries in eviction.

6. The **fetchPersistentState** element determines whether or not to fetch the persistent state of a cache and apply it to the local cache store when joining the cluster. If the cache store is shared the fetch persistent state is ignored, as caches access the same cache store. A configuration exception will be thrown when starting the cache service if more than one cache store has this property set to **true**. The **fetchPersistentState** property is **false** by default.

7. The **purgeOnStartup** element controls whether cache store is purged when it starts up and is **false** by default.

8. The **location** element configuration element sets a location on disk where the store can write.

9. These attributes configure aspects specific to each cache store. For example, the **location** attribute points to where the SingleFileStore will keep files containing data. Other stores may require more complex configuration.

10. The **singleton** element enables modifications to be stored by only one node in the cluster. This node is called the coordinator. The coordinator pushes the caches in-memory states to disk. This function is activated by setting the **enabled** attribute to **true** in all nodes. The **shared** parameter cannot be defined with **singleton** enabled at the same time. The **enabled** attribute is **false** by default.

11. The **pushStateWhenCoordinator** element is set to **true** by default. If **true**, this property will cause a node that has become the coordinator to transfer in-memory state to the underlying cache store. This parameter is useful where the coordinator has crashed and a new coordinator is elected.

7.4.3. LevelDB Cache Store Programmatic Configuration

The following is a sample programmatic configuration of LevelDB Cache Store:

```
Configuration cacheConfig = new ConfigurationBuilder().persistence()
        .addStore(LevelDBStoreConfigurationBuilder.class)
        .location("/tmp/leveldb/data")
        .expiredLocation("/tmp/leveldb/expired").build();
```

LevelDB Cache Store programmatic configuration

1. Use the **ConfigurationBuilder** to create a new configuration object.
2. Add the store using **LevelDBCacheStoreConfigurationBuilder** class to build its configuration.
3. Set the LevelDB Cache Store location path. The specified path stores the primary cache store data. The directory is automatically created if it does not exist.
4. Specify the location for expired data using the **expiredLocation** parameter for the LevelDB Store. The specified path stores expired data before it is purged. The directory is automatically created if it does not exist.

7.4.4. JdbcBinaryStore Programmatic Configuration

The **JdbcBinaryStore** supports all key types by storeing all keys with the same hash value (**hashCode** method on the key) in the same table row/blob.

IMPORTANT

Binary JDBC stores are deprecated in JBoss Data Grid 7.2, and are not recommended for production use. It is recommended to utilize a String Based store instead.

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

```
ConfigurationBuilder builder = new ConfigurationBuilder();
 builder.persistence()
    .addStore(JdbcBinaryStoreConfigurationBuilder.class)
    .fetchPersistentState(false)
    .ignoreModifications(false)
    .purgeOnStartup(false)
    .table()
      .dropOnExit(true)
      .createOnStart(true)
      .tableNamePrefix("ISPN_BUCKET_TABLE")
      .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
      .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
      .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
    .connectionPool()
      .connectionUrl("jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1")
      .username("sa")
      .driverClass("org.h2.Driver");
```

JdbcBinaryStore Programmatic Configuration (Library Mode)

1. Use the **ConfigurationBuilder** to create a new configuration object.
2. Add the **JdbcBinaryStore** configuration builder to build a specific configuration related to this store.
3. The **fetchPersistentState** element determines whether or not to fetch the persistent state of a cache and apply it to the local cache store when joining the cluster. If the cache store is shared the fetch persistent state is ignored, as caches access the same cache store. A configuration exception will be thrown when starting the cache service if more than one cache loader has this property set to **true**. The **fetchPersistentState** property is **false** by default.
4. The **ignoreModifications** element determines whether write methods are pushed to the

specific cache loader by allowing write operations to the local file cache loader, but not the shared cache loader. In some cases, transient application data should only reside in a file-based cache loader on the same server as the in-memory cache. For example, this would apply with a further JDBC based cache loader used by all servers in the network. **ignoreModifications** is **false** by default.

5. The **purgeOnStartup** element specifies whether the cache is purged when initially started.
6. Configure the table as follows:
    a. **dropOnExit** determines if the table will be dropped when the cache store is stopped. This is set to **false** by default.
    b. **createOnStart** creates the table when starting the cache store if no table currently exists. This method is **true** by default.
    c. **tableNamePrefix** sets the prefix for the name of the table in which the data will be stored.
    d. The **idColumnName** property defines the column where the cache key or bucket ID is stored.
    e. The **dataColumnName** property specifies the column where the cache entry or bucket is stored.
    f. The **timestampColumnName** element specifies the column where the time stamp of the cache entry or bucket is stored.
7. The **connectionPool** element specifies a connection pool for the JDBC driver using the following parameters:
    a. The **connectionUrl** parameter specifies the JDBC driver-specific connection URL.
    b. The **username** parameter contains the user name used to connect via the **connectionUrl**.
    c. The **driverClass** parameter specifies the class name of the driver used to connect to the database.

7.4.5. JdbcStringBasedStore Programmatic Configuration

The **JdbcStringBasedStore** stores each entry in its own row in the table, instead of grouping multiple entries into each row, resulting in increased throughput under a concurrent load.

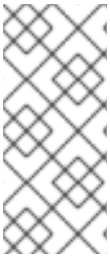
The following is a sample configuration for the *JdbcStringBasedStore* :

```
ConfigurationBuilder builder = new ConfigurationBuilder();
 builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
    .fetchPersistentState(false)
    .ignoreModifications(false)
    .purgeOnStartup(false)
    .table()
      .dropOnExit(true)
      .createOnStart(true)
      .tableNamePrefix("ISPN_STRING_TABLE")
      .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
      .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
```

```
    .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
  .dataSource()
    .jndiUrl("java:jboss/datasources/JdbcDS");
```

Configure the JdbcStringBasedStore Programmatically

1. Use the **ConfigurationBuilder** to create a new configuration object.
2. Add the **JdbcStringBasedStore** configuration builder to build a specific configuration related to this store.
3. The **fetchPersistentState** parameter determines whether or not to fetch the persistent state of a cache and apply it to the local cache store when joining the cluster. If the cache store is shared the fetch persistent state is ignored, as caches access the same cache store. A configuration exception will be thrown when starting the cache service if more than one cache loader has this property set to **true**. The **fetchPersistentState** property is **false** by default.
4. The **ignoreModifications** parameter determines whether write methods are pushed to the specific cache loader by allowing write operations to the local file cache loader, but not the shared cache loader. In some cases, transient application data should only reside in a file-based cache loader on the same server as the in-memory cache. For example, this would apply with a further JDBC based cache loader used by all servers in the network. **ignoreModifications** is **false** by default.
5. The **purgeOnStartup** parameter specifies whether the cache is purged when initially started.
6. Configure the Table
    a. **dropOnExit** determines if the table will be dropped when the cache store is stopped. This is set to **false** by default.
    b. **createOnStart** creates the table when starting the cache store if no table currently exists. This method is **true** by default.
    c. **tableNamePrefix** sets the prefix for the name of the table in which the data will be stored.
    d. The **idColumnName** property defines the column where the cache key or bucket ID is stored.
    e. The **dataColumnName** property specifies the column where the cache entry or bucket is stored.
    f. The **timestampColumnName** element specifies the column where the time stamp of the cache entry or bucket is stored.
7. The **dataSource** element specifies a data source using the following parameters:
    - The **jndiUrl** specifies the JNDI URL to the existing JDBC.

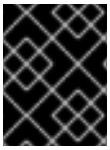


NOTE

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

7.4.6. JdbcMixedStore Programmatic Configuration

The **JdbcMixedStore** is a hybrid implementation that delegates keys based on their type to either the **JdbcBinaryStore** or **JdbcStringBasedStore**.



IMPORTANT

Mixed JDBC stores are deprecated in JBoss Data Grid 7.2, and are not recommended for production use. It is recommended to utilize a String Based store instead.

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

```
ConfigurationBuilder builder = new ConfigurationBuilder();
 builder.persistence().addStore(JdbcMixedStoreConfigurationBuilder.class)
    .fetchPersistentState(false)
    .ignoreModifications(false)
    .purgeOnStartup(false)
    .stringTable()
      .dropOnExit(true)
      .createOnStart(true)
      .tableNamePrefix("ISPN_MIXED_STR_TABLE")
      .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
      .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
      .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
    .binaryTable()
      .dropOnExit(true)
      .createOnStart(true)
      .tableNamePrefix("ISPN_MIXED_BINARY_TABLE")
      .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
      .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
      .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
    .connectionPool()
      .connectionUrl("jdbc:h2:mem:infinispan_binary_based;DB_CLOSE_DELAY=-1")
      .username("sa")
      .driverClass("org.h2.Driver");
```

Configure JdbcMixedStore Programmatically

1. Use the **ConfigurationBuilder** to create a new configuration object.
2. Add the **JdbcMixedStore** configuration builder to build a specific configuration related to this store.
3. The **fetchPersistentState** parameter determines whether or not to fetch the persistent state of a cache and apply it to the local cache store when joining the cluster. If the cache store is shared the fetch persistent state is ignored, as caches access the same cache store. A configuration exception will be thrown when starting the cache service if more than one cache loader has this property set to **true**. The **fetchPersistentState** property is **false** by default.
4. The **ignoreModifications** parameter determines whether write methods are pushed to the specific cache loader by allowing write operations to the local file cache loader, but not the shared cache loader. In some cases, transient application data should only reside in a file-based cache loader on the same server as the in-memory cache. For example, this would apply with a further JDBC based cache loader used by all servers in the network. **ignoreModifications** is **false** by default.

5. The **purgeOnStartup** parameter specifies whether the cache is purged when initially started.

6. Configure the table as follows:

    a. **dropOnExit** determines if the table will be dropped when the cache store is stopped. This is set to **false** by default.

    b. **createOnStart** creates the table when starting the cache store if no table currently exists. This method is **true** by default.

    c. **tableNamePrefix** sets the prefix for the name of the table in which the data will be stored.

    d. The **idColumnName** property defines the column where the cache key or bucket ID is stored.

    e. The **dataColumnName** property specifies the column where the cache entry or bucket is stored.

    f. The **timestampColumnName** element specifies the column where the time stamp of the cache entry or bucket is stored.

7. The **connectionPool** element specifies a connection pool for the JDBC driver using the following parameters:

    a. The **connectionUrl** parameter specifies the JDBC driver-specific connection URL.

    b. The **username** parameter contains the username used to connect via the **connectionUrl**.

    c. The **driverClass** parameter specifies the class name of the driver used to connect to the database.

7.4.7. JPA Cache Store Sample Programmatic Configuration

To configure JPA Cache Stores programatically in Red Hat JBoss Data Grid, use the following:

```
Configuration cacheConfig = new ConfigurationBuilder().persistence()
      .addStore(JpaStoreConfigurationBuilder.class)
      .persistenceUnitName("org.infinispan.loaders.jpa.configurationTest")
      .entityClass(User.class)
   .build();
```

The parameters used in this code sample are as follows:

- The **persistenceUnitName** parameter specifies the name of the JPA cache store in the configuration file (*persistence.xml*) that contains the JPA entity class.

- The **entityClass** parameter specifies the JPA entity class that is stored in this cache. Only one class can be specified for each configuration.

7.4.8. Cassandra Cache Store Sample Programmatic Configuration

The Cassandra cache store is not part of the Red Hat JBoss Data Grid's core libraries, and must be added to the classpath. For Maven projects this may be added with the following addition to your **pom.xml**:

```
<dependency>
```

```
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-cachestore-cassandra</artifactId>
    <version>...</version> <!-- 7.2.0 or later -->
</dependency>
```

The following configuration snippet provides an example on how to define a Cassandra Cache Store programmatically:

```
Configuration cacheConfig = new ConfigurationBuilder()
    .persistence()
    .addStore(CassandraStoreConfigurationBuilder.class)
    .addServer()
        .host("127.0.0.1")
        .port(9042)
    .addServer()
        .host("127.0.0.1")
        .port(9041)
    .autoCreateKeyspace(true)
    .keyspace("TestKeyspace")
    .entryTable("TestEntryTable")
    .consistencyLevel(ConsistencyLevel.LOCAL_ONE)
    .serialConsistencyLevel(ConsistencyLevel.SERIAL)
    .connectionPool()
        .heartbeatIntervalSeconds(30)
        .idleTimeoutSeconds(120)
        .poolTimeoutMillis(5)
    .build();
```

CHAPTER 8. THE CONFIGURATIONBUILDER API

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

NOTE

Programmatic configuration can only be accomplished in Red Hat JBoss Data Grid's Library Mode.

8.2. USING THE CONFIGURATIONBUILDER API

8.2.1. 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:

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);
```

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.

2. Create a new synchronously replicated cache programmatically.

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

   - Define or register the configuration with a manager:

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.

- 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.

8.2.2. Cluster-Wide Dynamic Cache Creation

When using the **getCache()** method, like in the above example, a cache will be created only on a single node. If the cache needs to be created dynamically on any new nodes that join the cluster, use the **createCache()** method instead:

```
Cache<String, String> cache = manager.administration().createCache("newCacheName",
"newTemplate");
```

While a cache created this way will be available on all nodes in the cluster, it will also be ephemeral: shutting down the entire cluster and restarting it will not automatically recreate the caches. To make the caches persistent, use the **PERMANENT** flag as follows:

```
Cache<String, String> cache =
manager.administration().withFlags(AdminFlag.PERMANENT).createCache("newCacheName",
"newTemplate");
```

In order for the above to work, global state must be enabled and a suitable configuration storage selected. The available configuration stores are:

- **VOLATILE**: as the name implies, this configuration storage does not support **PERMANENT** caches.
- **OVERLAY**: this stores configurations in the global shared state persistent path in a file named *caches.xml*.
- **MANAGED**: this is only supported in server deployments, and will store **PERMANENT** caches in the server model.
- **CUSTOM**: a custom configuration store.

8.2.3. 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.

Customize the Default Cache

```
String newCacheName = "newCache";
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration dcc = manager.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);
```

1. Read an instance of a default **Configuration** object to get the default configuration.
2. Use the **ConfigurationBuilder** to construct and modify the cache mode and L1 cache lifespan on a new configuration object.
3. Register/define your cache configuration with a cache manager.
4. Obtain a reference to **newCache**, containing the specified configuration.

8.2.4. 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.

Creating a Customized Cache Using a Non-Default Named Cache

```
String newCacheName = "newCache";
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-config-file.xml");
Configuration rc = manager.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);
```

1. Read the **replicatedCache** to get the default configuration.
2. Use the **ConfigurationBuilder** to construct and modify the desired configuration on a new configuration object.
3. Register/define your cache configuration with a cache manager.
4. Obtain a reference to **newCache**, containing the specified configuration.

8.2.5. 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.

8.2.6. Global Configuration Examples

8.2.6.1. 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:

Configuring the Transport Layer

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .transport().defaultTransport()
  .build();
```

8.2.6.2. 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.

Configuring the Cache Manager Name

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

8.2.6.3. Globally Configure JGroups

Red Hat JBoss Data Grid must have an appropriate JGroups configuration in order to operate in clustered mode; the following sample configuration demonstrates how to pass a predefined JGroups configuration file into the configuration:

JGroups Programmatic Configuration

```
GlobalConfiguration gc = new GlobalConfigurationBuilder()
    .transport()
        .defaultTransport()
    .addProperty("configurationFile","jgroups.xml")
    .build();
```

Red Hat JBoss Data Grid will first search for *jgroups.xml* in the classpath; if no instances are found in the classpath it will then search for an absolute path name.

8.2.7. Cache Level Configuration Examples

8.2.7.1. 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:

Configure Cluster Mode at Cache Level

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

```
  .clustering()
    .cacheMode(CacheMode.DIST_SYNC)
    .sync()
    .l1().lifespan(25000L).enable()
    .hash().numOwners(3)
  .build();
```

8.2.7.2. Cache Level Eviction and Expiration Configuration

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

Configuring Expiration and Eviction at the Cache Level

```
Configuration config = new ConfigurationBuilder()
        .memory()
          .size(20000)
        .expiration()
          .wakeUpInterval(5000L)
          .maxIdle(120000L)
        .build();
```

8.2.7.3. 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.

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();
```

8.2.7.4. 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:

Configuring Chained Persistent Stores at Cache Level

```
Configuration conf = new ConfigurationBuilder()
        .persistence()
          .passivation(false)
          .addSingleFileStore()
            .location("/tmp/firstDir")
```

```
        .persistence()
          .passivation(false)
          .addSingleFileStore()
            .location("/tmp/secondDir")
        .build();
```

8.2.7.5. 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:

Configuring Advanced Externalizers at Cache Level

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

8.2.7.6. Cache Level Configuration for Partition Handling (Library Mode)

In the event of a split brain scenario a partition handling strategy can be selected to provide either consistency or availability of data. If availability is chosen and data becomes inconsistent a merge policy can also be selected to define how data is merged upon node rejoins. An example configuration is shown below.

```
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
                .whenSplit(PartitionHandling.DENY_READ_WRITES)
                .mergePolicy(MergePolicies.REMOVE_ALL);
```

Additional information regarding partition handling is found in the Administration and Configuration Guide.

NOTE

To configure Partition Handling in Client-Server Mode it must be enabled declaratively as described in the Administration and Configuration Guide.

CHAPTER 9. THE EXTERNALIZABLE API

9.1. 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 Red Hat 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 Red Hat JBoss Data Grid.

9.2. 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 Red Hat JBoss Data Grid can be used in one of two ways:

- Use an Externalizable Interface.
- Use an advanced externalizer.

9.3. 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.

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;
  }
```

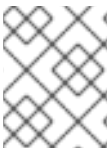
```
    public static class PersonExternalizer implements Externalizer<Person> {
      @Override
      public void writeObject(ObjectOutput output, Person person)
            throws IOException {
        output.writeObject(person.name);
        output.writeInt(person.age);
      }

      @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 marshall the object using the Externalizer class passed.

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

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

Advanced Externalizers are available for users affected by these disadvantages.



NOTE

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

9.4. USING AN ADVANCED EXTERNALIZER

9.4.1. Using an Advanced Externalizer

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

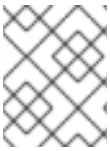
1. Define and implement the **readObject()** and **writeObject()** methods.
2. Link externalizers with marshaller classes.
3. Register the advanced externalizer.

9.4.2. Implement the Methods

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

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.

9.4.3. Link Externalizers with Marshaller Classes

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

The following is a sample implementation:

```
import org.infinispan.util.Util;
<!-- Additional configuration information here -->
```

```
@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));
}
```

9.4.4. 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:

Registering the Advanced Externalizer Programmatically

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

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

9.4.5. 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.

Registering Multiple Externalizers

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

9.5. CUSTOM EXTERNALIZER ID VALUES

9.5.1. 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 9.1. 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
1800-1849	JBoss Data Grid Scripting Module
1850-1899	JBoss Data Grid Server Event Logger Module
1900-1999	JBoss Data Grid Remote Store

9.5.2. Customize the Externalizer ID (Programmatically)

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

Assign an ID to the Externalizer

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

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

CHAPTER 10. THE NOTIFICATION/LISTENER API

10.1. 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.

10.2. 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:

Configuring a Listener

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

10.3. LISTENER NOTIFICATIONS

10.3.1. 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.

10.3.2. About Cache-level Notifications

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

10.3.3. Cache Manager-level Notifications

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

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

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

The first two types of events, **CacheStarted** and **CacheStopped** are highly similar, and the following example demonstrates printing out the name of the cache that has started or stopped:

```
@CacheStarted
public void cacheStarted(CacheStartedEvent event){
   // Print the name of the Cache that started
   log.info("Cache Started: " + event.getCacheName());
}

@CacheStopped
public void cacheStopped(CacheStoppedEvent event){
   // Print the name of the Cache that stopped
   log.info("Cache Stopped: " + event.getCacheName());
}
```

When receiving a **ViewChangedEvent** or **MergeEvent** note that the list of old and new members is from the node that generated the event. For instance, consider the following scenario:

- A JDG Cluster currently consists of nodes A, B, and C.
- Node D joins the cluster.
- Nodes A, B, and C will receive a **ViewChangedEvent** with [A,B,C] as the list of old members, and [A,B,C,D] as the list of new members.
- Node D will receive a **ViewChangedEvent** with [D] as the list of old members, and [A,B,C,D] as the list of new members.

Therefore, a set intersection may be used to determine if a node has recently joined or left a cluster. By using **getOldMembers()** in conjunction with **getNewMembers()**, we may determine the set of nodes that have joined or left the cluster, as seen below:

```
@ViewChanged
public void viewChanged(ViewChangedEvent event){
   HashSet<Address> oldMembers = new HashSet(event.getOldMembers());
   HashSet<Address> newMembers = new HashSet(event.getNewMembers());
   HashSet<Address> oldCopy = (HashSet<Address>)oldMembers.clone();

   // Remove all new nodes from the old view.
   // The resulting set indicates nodes that have left the cluster.
   oldCopy.removeAll(newMembers);
   if(oldCopy.size() > 0){
      for (Address oldAdd : oldCopy){
         log.info("Node left:" + oldAdd.toString());
      }
   }

   // Remove all old nodes from the new view.
   // The resulting set indicates nodes that have joined the cluster.
   newMembers.removeAll(oldMembers);
   if(newMembers.size() > 0){
      for(Address newAdd : newMembers){
         log.info("Node joined: " + newAdd.toString());
      }
   }
}
```

Similar logic may be used during a **MergeEvent** to determine the new set of members in the cluster.

10.3.4. 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 XML configuration file to tune the thread pool that is used to dispatch asynchronous notifications.

IMPORTANT

When using a synchronous, non-clustered listener that handles the **CacheEntryExpiredEvent** ensure that this listener does not block execution, as the expiration reaper is also synchronous in a non-clustered environment.

10.4. MODIFYING CACHE ENTRIES

10.4.1. Modifying Cache Entries

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

10.4.2. 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).

10.4.3. 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:

Modified Listener

```
@Listener
public class PrintWhenModified {
```

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

}
```

10.5. CLUSTERED LISTENERS

10.5.1. Clustered Listeners

Clustered listeners allow listeners to be used in a distributed cache configuration. In a distributed cache environment, registered local listeners are only notified of events that are local to the node where the event has occurred. Clustered listeners resolve this issue by allowing a single listener to receive any write notification that occurs in the cluster, regardless of where the event occurred. As a result, clustered listeners perform slower than non-clustered listeners, which only provide event notifications for the node on which the event occurs.

When using clustered listeners, client applications are notified when an entry is added, updated, expired, or deleted in a particular cache. The event is cluster-wide so that client applications can access the event regardless of the node on which the application resides or connects with.

The event will always be triggered on the node where the listener was registered, while disregarding where the cache update originated.

10.5.2. Configuring Clustered Listeners

In the following use case, listener stores events as it receives them.

Procedure: Clustered Listener Configuration

```
@Listener(clustered = true)
  protected static class ClusterListener {
     List<CacheEntryEvent> events = Collections.synchronizedList(new ArrayList<CacheEntryEvent>
());

     @CacheEntryCreated
     @CacheEntryModified
     @CacheEntryExpired
     @CacheEntryRemoved
     public void onCacheEvent(CacheEntryEvent event) {
        log.debugf("Adding new cluster event %s", event);
        events.add(event);
     }
  }

  public void addClusterListener(Cache<?, ?> cache) {
     ClusterListener clusterListener = new ClusterListener();
     cache.addListener(clusterListener);
  }
```

1. Clustered listeners are enabled by annotating the **@Listener** class with **clustered=true**.

2. The following methods are annotated to allow client applications to be notified when entries are added, modified, expired, or removed.
   - **@CacheEntryCreated**
   - **@CacheEntryModified**
   - **@CacheEntryExpired**
   - **@CacheEntryRemoved**
3. The listener is registered with a cache, with the option of passing on a filter or converter.

The following limitations occur when using clustered listeners, that do not apply to non-clustered listeners:

- A cluster listener can only listen to entries that are created, modified, expired, or removed. No other events are listened to by a clustered listener.
- Only post events are sent to a clustered listener, pre events are ignored.

10.5.3. The Cache Listener API

Clustered listeners can be added on top of the existing **@CacheListener API** via the **addListener** method.

The Cache Listener API

```
cache.addListener(Object listener, Filter filter, Converter converter);
```

```
public @interface Listener {
  boolean clustered() default false;
  boolean includeCurrentState() default false;
  boolean sync() default true;
}
```

```
interface CacheEventFilter<K,V> {
  public boolean accept(K key, V oldValue, Metadata oldMetadata, V newValue, Metadata
newMetadata, EventType eventType);
}
```

```
interface CacheEventConverter<K,V,C> {
  public C convert(K key, V oldValue, Metadata oldMetadata, V newValue, Metadata newMetadata,
EventType eventType);
}
```

The Cache API

The local or clustered listener can be registered with the **cache.addListener** method, and is active until one of the following events occur.

- The listener is explicitly unregistered by invoking **cache.removeListener**.
- The node on which the listener was registered crashes.

Listener Annotation

The listener annotation is enhanced with three attributes:

- **clustered()**:This attribute defines whether the annotated listener is clustered or not. Note that clustered listeners can only be notified for **@CacheEntryRemoved**, **@CacheEntryCreated**, **@CacheEntryExpired**, and **@CacheEntryModified** events. This attribute is false by default.

- **includeCurrentState()**: This attribute applies to clustered listeners only, and is false by default. When set to **true**, the entire existing state within the cluster is evaluated. When being registered, a listener will immediately be sent a **CacheCreatedEvent** for every entry in the cache.

- Refer to About Synchronous and Asynchronous Notifications for information regarding **sync()**.

oldValue and oldMetadata

The **oldValue** and **oldMetadata** values are extra methods on the accept method of **CacheEventFilter** and **CacheEventConverter** classes. They values are provided to any listener, including local listeners. For more information about these values, see the *JBoss Data Grid API Documentation* .

EventType

The **EventType** includes the type of event, whether it was a retry, and if it was a pre or post event.

When using clustered listeners, the order in which the cache is updated is reflected in the sequence of notifications received.

The clustered listener does not guarantee that an event is sent only once. The listener implementation must be idempotent in order to prevent situations where the same event is sent more than once. Implementors can expect singularity to be honored for stable clusters and outside of the time span in which synthetic events are generated as a result of **includeCurrentState**.

10.5.4. Clustered Listener Example

The following use case demonstrates a listener that wants to know when orders are generated that have a destination of New York, NY. The listener requires a Filter that filters all orders that come in and out of New York. The listener also requires a Converter as it does not require the entire order, only the date it is to be delivered.

Use Case: Filtering and Converting the New York orders

```
class CityStateFilter implements CacheEventFilter<String, Order> {
   private String state;
   private String city;

   public boolean accept(String orderId, Order oldOrder,
                         Metadata oldMetadata, Order newOrder,
                         Metadata newMetadata, EventType eventType) {
      switch (eventType.getType()) {
         // Only send update if the order is going to our city
         case CACHE_ENTRY_CREATED:
            return city.equals(newOrder.getCity()) &&
                  state.equals(newOrder.getState());
         // Only send update if our order has changed from our city to elsewhere or if is now going to
```

```
our city
            case CACHE_ENTRY_MODIFIED:
               if (city.equals(oldOrder.getCity()) &&
                     state.equals(oldOrder.getState())) {
                  // If old city matches then we have to compare if new order is no longer going to our city
                  return !city.equals(newOrder.getCity()) ||
                        !state.equals(newOrder.getState());
               } else {
                  // If the old city doesn't match ours then only send update if new update does match ours
                  return city.equals(newOrder.getCity()) &&
                        state.equals(newOrder.getState());
               }
               // On remove we have to send update if our order was originally going to city
            case CACHE_ENTRY_REMOVED:
               return city.equals(oldOrder.getCity()) &&
                     state.equals(oldOrder.getState());
         }
         return false;
      }
   }

   class OrderDateConverter implements CacheEventConverter<String, Order, Date> {
      private String state;
      private String city;

      public Date convert(String orderId, Order oldValue,
                          Metadata oldMetadata, Order newValue,
                          Metadata newMetadata, EventType eventType) {
         // If remove we do not care about date - this tells listener to remove its data
         if (eventType.isRemove()) {
            return null;
         } else if (eventType.isModified()) {
            if (state.equals(newValue.getState()) &&
                  city.equals(newValue.getCity())) {
               // If it is a modification meaning the destination has changed to ours then we allow it
               return newValue.getDate();
            } else {
               // If destination is no longer our city it means it was changed from us so send null
               return null;
            }
         } else {
            // This was a create so we always send date
            return newValue.getDate();
         }
      }
   }
```

10.5.5. Optimized Cache Filter Converter

The example provided in Clustered Listener Example could use the optimized **CacheEventFilterConverter**, in order to perform the filtering and converting of results into one step.

The **CacheEventFilterConverter** is an optimization that allows the event filter and conversion to be performed in one step. This can be used when an event filter and converter are most efficiently used as the same object, composing the filtering and conversion in the same method. This can only be used in

situations where your conversion will not return a null value, as a returned value of null indicates that the value did not pass the filter. To convert a null value, use the **CacheEventFilter** and the **CacheEventConverter** interfaces independently.

The following is an example of the New York orders use case using the **CacheEventFilterConverter**:

CacheEventFilterConverter

```
class OrderDateFilterConverter extends AbstractCacheEventFilterConverter<String, Order, Date> {
   private final String state;
   private final String city;

   public Date filterAndConvert(String orderId, Order oldValue,
                                Metadata oldMetadata, Order newValue,
                                Metadata newMetadata, EventType eventType) {
      // Remove if the date is not required - this tells listener to remove its data
      if (eventType.isRemove()) {
         return null;
      } else if (eventType.isModified()) {
         if (state.equals(newValue.getState()) &&
            city.equals(newValue.getCity())) {
            // If it is a modification meaning the destination has changed to ours then we allow it
            return newValue.getDate();
         } else {
            // If destination is no longer our city it means it was changed from us so send null
            return null;
         }
      } else {
         // This was a create so we always send date
         return newValue.getDate();
      }
   }
}
```

When registering the listener, provide the **FilterConverter** as both arguments to the filter and converter:

```
OrderDateFilterConverter filterConverter = new OrderDateFilterConverter("NY", "New York");
cache.addListener(listener, filterConveter, filterConverter);
```

10.6. REMOTE EVENT LISTENERS (HOT ROD)

10.6.1. Remote Event Listeners (Hot Rod)

Event listeners allow Red Hat JBoss Data Grid Hot Rod servers to be able to notify remote clients of events such as **CacheEntryCreated**, **CacheEntryModified**, **CacheEntryExpired** and **CacheEntryRemoved**. Clients can choose whether or not to listen to these events to avoid flooding connected clients. This assumes that clients maintain persistent connections to the servers.

Client listeners for remote events can be added similarly to clustered listeners in library mode. The following example demonstrates a remote client listener that prints out each event it receives.

Event Print Listener

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class EventLogListener {

  @ClientCacheEntryCreated
  public void handleCreatedEvent(ClientCacheEntryCreatedEvent e) {
    System.out.println(e);
  }

  @ClientCacheEntryModified
  public void handleModifiedEvent(ClientCacheEntryModifiedEvent e) {
    System.out.println(e);
  }

  @ClientCacheEntryExpired
  public void handleExpiredEvent(ClientCacheEntryExpiredEvent e) {
    System.out.println(e);
  }

  @ClientCacheEntryRemoved
  public void handleRemovedEvent(ClientCacheEntryRemovedEvent e) {
    System.out.println(e);
  }

}
```

- **ClientCacheEntryCreatedEvent** and **ClientCacheEntryModifiedEvent** instances provide information on the key and version of the entry. This version can be used to invoke conditional operations on the server, such a **replaceWithVersion** or **removeWithVersion**.
- **ClientCacheEntryExpiredEvent** events are sent when either a **get()** is called on an expired entry, or when the expiration reaper detects that an entry has expired. Once the entry has expired the cache will nullify the entry, and adjust its size appropriately; however, the event will only be generated in the two scenarios listed.
- **ClientCacheEntryRemovedEvent** events are only sent when the remove operation succeeds. If a remove operation is invoked and no entry is found or there are no entries to remove, no event is generated. If users require remove events regardless of whether or not they are successful, a customized event logic can be created.
- All client cache entry created, modified, and removed events provide a **boolean isCommandRetried()** method that will return **true** if the write command that caused it has to be retried due to a topology change. This indicates that the event has been duplicated or that another event was dropped and replaced, such as where a Modified event replaced a Created event.

IMPORTANT

If the expected workload favors writes over reads it will be necessary to filter the events sent to prevent a large amount of excessive traffic being generated which may cause issues on either the client or the network. For more details on filtering events refer to .

10.6.2. Adding and Removing Event Listeners

Registering an Event Listener with the Server

The following example registers the Event Print Listener with the server. See Event Print Listener .

Adding an Event Listener

```
RemoteCache<Integer, String> cache = rcm.getCache();
cache.addClientListener(new EventLogListener());
```

Removing a Client Event Listener

A client event listener can be removed as follows

```
EventLogListener listener = ...
cache.removeClientListener(listener);
```

10.6.3. Remote Event Client Listener Example

The following procedure demonstrates the steps required to configure a remote client listener to interact with the remote cache via Hot Rod.

Configuring Remote Event Listeners

1. Download the Red Hat JBoss Data Grid distribution from the Red Hat Customer Portal
   The latest JBoss Data Grid distribution includes the Hot Rod server with which the client will communicate.

2. Start the server
   Start the JBoss Data Grid server by using the following command from the root of the server.

   ```
   $ ./bin/standalone.sh
   ```

3. Write the application to interact with the Hot Rod server

   a. Maven Users
      Create an application with the following dependency and change the version to **8.5.3.Final-redhat-00002** or later:

      ```
      <properties>
        <infinispan.version>8.5.3.Final-redhat-00002</infinispan.version>
      </properties>
      [...]
      <dependency>
        <groupId>org.infinispan</groupId>
        <artifactId>infinispan-remote</artifactId>
        <version>${infinispan.version}</version>
      </dependency>
      ```

   b. Non-Maven users, adjust according to your chosen build tool or download the distribution containing all JBoss Data Grid jars.

4. Write the client application
   The following demonstrates a simple remote event listener that logs all events received.

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class EventLogListener {

 @ClientCacheEntryCreated
 @ClientCacheEntryModified
 @ClientCacheEntryRemoved
 public void handleRemoteEvent(ClientEvent event) {
   System.out.println(event);
  }

}
```

5. Use the remote event listener to execute operations against the remote cache
   The following example demonstrates a simple main java class, which adds the remote event listener and executes some operations against the remote cache.

```
RemoteCacheManager rcm = new RemoteCacheManager();
RemoteCache<Integer, String> cache = rcm.getCache();
EventLogListener listener = new EventLogListener();
try {
 cache.addClientListener(listener);
 cache.put(1, "one");
 cache.put(1, "new-one");
 cache.remove(1);
} finally {
 cache.removeClientListener(listener);
}
```

Result

Once executed, the console output should appear similar to the following:

```
ClientCacheEntryCreatedEvent(key=1,dataVersion=1)
ClientCacheEntryModifiedEvent(key=1,dataVersion=2)
ClientCacheEntryRemovedEvent(key=1)
```

The output indicates that by default, events come with the key and the internal data version associated with current value. The actual value is not sent back to the client for performance reasons. Receiving remote events has a performance impact, which is increased with cache size, as more operations are executed. To avoid inundating Hot Rod clients, filter remote events on the server side, or customize the event contents.

10.6.4. Filtering Remote Events

10.6.4.1. Filtering Remote Events

To prevent clients being inundated with events, Red Hat JBoss Data Grid Hot Rod remote events can be filtered by providing key/value filter factories that create instances that filter which events are sent to clients, and how these filters can act on client provided information.

Sending events to remote clients has a performance cost, which increases with the number of clients with registered remote listeners. The performance impact also increases with the number of modifications that are executed against the cache.

The performance cost can be reduced by filtering the events being sent on the server side. Custom code can be used to exclude certain events from being broadcast to the remote clients to improve performance.

Filtering can be based on either key or value information, or based on cache entry metadata. To enable filtering, a cache event filter factory that produces filter instances must be created. The following is a sample implementation that filters key "2" out of the events sent to clients.

KeyValueFilter

```
package sample;

import java.io.Serializable;
import org.infinispan.notifications.cachelistener.filter.*;
import org.infinispan.metadata.*;

@NamedFactory(name = "basic-filter-factory")
public class BasicKeyValueFilterFactory implements CacheEventFilterFactory {
  @Override public CacheEventFilter<Integer, String> getFilter(final Object[] params) {
    return new BasicKeyValueFilter();
  }

    static class BasicKeyValueFilter implements CacheEventFilter<Integer, String>, Serializable {
      @Override public boolean accept(Integer key, String oldValue, Metadata oldMetadata, String
newValue, Metadata newMetadata, EventType eventType) {
        return !"2".equals(key);
      }
    }
}
```

In order to register a listener with this key value filter factory, the factory must be given a unique name, and the Hot Rod server must be plugged with the name and the cache event filter factory instance.

10.6.4.2. Custom Filters for Remote Events

Custom filters can improve performance by excluding certain event information from being broadcast to the remote clients.

To plug the JBoss Data Grid Server with a custom filter use the following procedure:

Using a Custom Filter

1. Create a *JAR* file with the filter implementation within it. Each factory must have a name assigned to it via the **org.infinispan.filter.NamedFactory** annotation. The example uses a **KeyValueFilterFactory**.

2. Create a *META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory* file within the *JAR* file, and within it write the fully qualified class name of the filter class implementation.

3. Deploy the *JAR* file in the JBoss Data Grid Server by performing any of the following options:

- Option 1: Deploy the JAR through the deployment scanner
  - Copy the *JAR* to the **$JDG_HOME/standalone/deployments/** directory. The deployment scanner actively monitors this directory and will deploy the newly placed file.
- Option 2: Deploy the JAR through the CLI
  - Connect to the desired instance with the CLI:

```
[$JDG_HOME] $ bin/cli.sh --connect=$IP:$PORT
```

  - Once connected execute the **deploy** command:

```
deploy /path/to/artifact.jar
```

- Option 3: Deploy the JAR as a custom module
  - Connect to the JDG server by running the below command:

```
[$JDG_HOME] $ bin/cli.sh --connect=$IP:$PORT
```

  - The jar containing the Custom Filter must be defined as a module for the Server; to add this substitute the desired name of the module and the .jar name in the below command, adding additional dependencies as necessary for the Custom Filter:

```
module add --name=$MODULE-NAME --resources=$JAR-NAME.jar --
dependencies=org.infinispan
```

  - In a different window add the newly added module as a dependency to the **org.infinispan** module by editing **$JDG_HOME/modules/system/layers/base/org/infinispan/main/module.xml**. In this file add the following entry:

```
<dependencies>
 [...]
 <module name="$MODULE-NAME">
</dependencies>
```

  - Restart the JDG server.

Once the server is plugged with the filter, add a remote client listener that will use the filter. The following example extends the EventLogListener implementation provided in Remote Event Client Listener Example (See Remote Event Client Listener Example), and overrides the **@ClientListener** annotation to indicate the filter factory to use with the listener.

Add Filter Factory to the Listener

```
@org.infinispan.client.hotrod.annotation.ClientListener(filterFactoryName = "basic-filter-factory")
public class BasicFilteredEventLogListener extends EventLogListener {}
```

The listener can now be added via the RemoteCacheAPI. The following example demonstrates this, and executes some operations against the remote cache.

Register the Listener with the Server

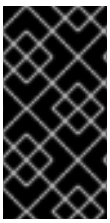
```
import org.infinispan.client.hotrod.*;

RemoteCacheManager rcm = new RemoteCacheManager();
RemoteCache<Integer, String> cache = rcm.getCache();
BasicFilteredEventLogListener listener = new BasicFilteredEventLogListener();
try {
  cache.addClientListener(listener);
  cache.putIfAbsent(1, "one");
  cache.replace(1, "new-one");
  cache.putIfAbsent(2, "two");
  cache.replace(2, "new-two");
  cache.putIfAbsent(3, "three");
  cache.replace(3, "new-three");
  cache.remove(1);
  cache.remove(2);
  cache.remove(3);
} finally {
  cache.removeClientListener(listener);
}
```

The system output shows that the client receives events for all keys except those that have been filtered.

Result

The following demonstrates the resulting system output from the provided example.

```
ClientCacheEntryCreatedEvent(key=1,dataVersion=1)
ClientCacheEntryModifiedEvent(key=1,dataVersion=2)
ClientCacheEntryCreatedEvent(key=3,dataVersion=5)
ClientCacheEntryModifiedEvent(key=3,dataVersion=6)
ClientCacheEntryRemovedEvent(key=1)
ClientCacheEntryRemovedEvent(key=3)
```



IMPORTANT

Filter instances must be marshallable when they are deployed in a cluster in order for filtering to occur where the event is generated, even if the event is generated in a different node to where the listener is registered. To make them marshallable, either make them extend Serializable, Externalizable, or provide a custom Externalizer.

10.6.4.3. Enhanced Filter Factories

When adding client listeners, users can provide parameters to the filter factory in order to generate different filter instances with different behaviors from a single filter factory based on client-side information.

The following configuration demonstrates how to enhance the filter factory so that it can filter dynamically based on the key provided when adding the listener, rather than filtering on a statically given key.

Configuring an Enhanced Filter Factory

-

```
package sample;

import java.io.Serializable;
import org.infinispan.notifications.cachelistener.filter.*;
import org.infinispan.metadata.*;

@NamedFactory(name = "basic-filter-factory")
public class BasicKeyValueFilterFactory implements CacheEventFilterFactory {
  @Override public CacheEventFilter<Integer, String> getFilter(final Object[] params) {
    return new BasicKeyValueFilter(params);
}

  static class BasicKeyValueFilter implements CacheEventFilter<Integer, String>, Serializable {
    private final Object[] params;
    public BasicKeyValueFilter(Object[] params) { this.params = params; }
    @Override public boolean accept(Integer key, String oldValue, Metadata oldMetadata, String
newValue, Metadata newMetadata, EventType eventType) {
      return !params[0].equals(key);
    }
  }
}
```

The filter can now filter by “3” instead of “2”:

Running an Enhanced Filter Factory

```
import org.infinispan.client.hotrod.*;

RemoteCacheManager rcm = new RemoteCacheManager();
RemoteCache<Integer, String> cache = rcm.getCache();
BasicFilteredEventLogListener listener = new BasicFilteredEventLogListener();
try {
  cache.addClientListener(listener, new Object[]{3}, null); // <- Filter parameter passed
  cache.putIfAbsent(1, "one");
  cache.replace(1, "new-one");
  cache.putIfAbsent(2, "two");
  cache.replace(2, "new-two");
  cache.putIfAbsent(3, "three");
  cache.replace(3, "new-three");
  cache.remove(1);
  cache.remove(2);
  cache.remove(3);
} finally {
  cache.removeClientListener(listener);
}
```

Result

The provided example results in the following output:

```
ClientCacheEntryCreatedEvent(key=1,dataVersion=1)
ClientCacheEntryModifiedEvent(key=1,dataVersion=2)
ClientCacheEntryCreatedEvent(key=2,dataVersion=3)
```

```
ClientCacheEntryModifiedEvent(key=2,dataVersion=4)
ClientCacheEntryRemovedEvent(key=1)
ClientCacheEntryRemovedEvent(key=2)
```

The amount of information sent to clients can be further reduced or increased by customizing remote events.

10.6.5. Customizing Remote Events

10.6.5.1. Customizing Remote Events

In Red Hat JBoss Data Grid, Hot Rod remote events can be customized to contain the information required to be sent to a client. By default, events contain only a basic set of information, such as a key and type of event, in order to avoid overloading the client, and to reduce the cost of sending them.

The information included in these events can be customized to contain more information, such as values, or contain even less information. Customization is done via **CacheEventConverter** instances, which are created by implementing a **CacheEventConverterFactory** class. Each factory must have a name associated to it via the **@NamedFactory** annotation.

To plug the Red Hat JBoss Data Grid Server with an event converter use the following procedure:

Using a Converter

1. Create a *JAR* file with the converter implementation within it. Each factory must have a name assigned to it via the **org.infinispan.filter.NamedFactory** annotation.
2. Create a *META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory* file within the *JAR* file and within it, write the fully qualified class name of the converter class implementation.
3. Deploy the *JAR* file in the Red Hat JBoss Data Grid Server by performing any of the following options:
   - Option 1: Deploy the JAR through the deployment scanner
     - Copy the *JAR* to the **$JDG_HOME/standalone/deployments/** directory. The deployment scanner actively monitors this directory and will deploy the newly placed file.
   - Option 2: Deploy the JAR through the CLI
     - Connect to the desired instance with the CLI:

       ```
       [$JDG_HOME] $ bin/cli.sh --connect=$IP:$PORT
       ```

     - Once connected execute the **deploy** command:

       ```
       deploy /path/to/artifact.jar
       ```

   - Option 3: Deploy the JAR as a custom module
     - Connect to the JDG server by running the below command:

```
[$JDG_HOME] $ bin/cli.sh --connect=$IP:$PORT
```

- The jar containing the Custom Converter must be defined as a module for the Server; to add this substitute the desired name of the module and the .jar name in the below command, adding additional dependencies as necessary for the Custom Converter:

```
module add --name=$MODULE-NAME --resources=$JAR-NAME.jar --
dependencies=org.infinispan
```

- In a different window add the newly added module as a dependency to the **org.infinispan** module by editing **$JDG_HOME/modules/system/layers/base/org/infinispan/main/module.xml**. In this file add the following entry:

```
<dependencies>
  [...]
  <module name="$MODULE-NAME">
</dependencies>
```

- Restart the JDG server.

Converters can also act on client provided information, allowing converter instances to customize events based on the information provided when the listener was added. The API allows converter parameters to be passed in when the listener is added.

10.6.5.2. Adding a Converter

When a listener is added, the name of a converter factory can be provided to use with the listener. When the listener is added, the server looks up the factory and invokes the **getConverter** method to get a **org.infinispan.filter.Converter** class instance to customize events server side.

The following example demonstrates sending custom events containing value information to remote clients for a cache of Integers and Strings. The converter generates a new custom event, which includes the value as well as the key in the event. The custom event has a bigger event payload compared with default events, however if combined with filtering, it can reduce bandwidth cost.

Sending Custom Events

```
import org.infinispan.notifications.cachelistener.filter.*;

@NamedFactory(name = "value-added-converter-factory")
class ValueAddedConverterFactory implements CacheEventConverterFactory {
  // The following types correspond to the Key, Value, and the returned Event, respectively.
  public CacheEventConverter<Integer, String, ValueAddedEvent> getConverter(final Object[]
params) {
    return new ValueAddedConverter();
  }

  static class ValueAddedConverter implements CacheEventConverter<Integer, String,
ValueAddedEvent> {
    public ValueAddedEvent convert(Integer key, String oldValue,
                                   Metadata oldMetadata, String newValue,
                                   Metadata newMetadata, EventType eventType) {
      return new ValueAddedEvent(key, newValue);
```

```
    }
  }
}

// Must be Serializable or Externalizable.
class ValueAddedEvent implements Serializable {
   final Integer key;
   final String value;
   ValueAddedEvent(Integer key, String value) {
     this.key = key;
     this.value = value;
   }
}
```

10.6.5.3. Lightweight Events

Other converter implementations are able to send back events that contain no key or event type information, resulting in extremely lightweight events at the expense of having rich information provided by the event.

In order to plug the server with this converter, deploy the converter factory and associated converter class within a *JAR* file including a service definition inside the *META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory* file as follows:

```
sample.ValueAddedConverterFactor
```

The client listener must then be linked with the converter factory by adding the factory name to the **@ClientListener** annotation.

```
@ClientListener(converterFactoryName = "value-added-converter-factory")
public class CustomEventLogListener { ... }
```

10.6.5.4. Dynamic Converter Instances

Dynamic converter instances convert based on parameters provided when the listener is registered. Converters use the parameters received by the converter factories to enable this option. For example:

Dynamic Converter

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;

class DynamicCacheEventConverterFactory implements CacheEventConverterFactory {
  // The following types correspond to the Key, Value, and the returned Event, respectively.
  public CacheEventConverter<Integer, String, CustomEvent> getConverter(final Object[] params) {
    return new DynamicCacheEventConverter(params);
  }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers needed when running in a
cluster
class DynamicCacheEventConverter implements CacheEventConverter<Integer, String,
CustomEvent>, Serializable {
  final Object[] params;
```

```
  DynamicCacheEventConverter(Object[] params) {
    this.params = params;
  }

  public CustomEvent convert(Integer key, String oldValue, Metadata metadata, String newValue,
Metadata prevMetadata, EventType eventType) {
    // If the key matches a key given via parameter, only send the key information
    if (params[0].equals(key))
      return new ValueAddedEvent(key, null);

    return new ValueAddedEvent(key, newValue);
  }
}
```

The dynamic parameters required to do the conversion are provided when the listener is registered:

```
RemoteCache<Integer, String> cache = rcm.getCache();
cache.addClientListener(new EventLogListener(), null, new Object[]{1});
```

10.6.5.5. Adding a Remote Client Listener for Custom Events

Implementing a listener for custom events is slightly different to other remote events, as they involve non-default events. The same annotations are used as in other remote client listener implementations, but the callbacks receive instances of **ClientCacheEntryCustomEvent<T>**, where **T** is the type of custom event we are sending from the server. For example:

Custom Event Listener Implementation

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener(converterFactoryName = "value-added-converter-factory")
public class CustomEventLogListener {

   @ClientCacheEntryCreated
   @ClientCacheEntryModified
   @ClientCacheEntryRemoved
   public void handleRemoteEvent(ClientCacheEntryCustomEvent<ValueAddedEvent> event)
   {
      System.out.println(event);
   }
}
```

To use the remote event listener to execute operations against the remote cache, write a simple main Java class, which adds the remote event listener and executes some operations against the remote cache. For example:

Execute Operations against the Remote Cache

```
import org.infinispan.client.hotrod.*;

RemoteCacheManager rcm = new RemoteCacheManager();
RemoteCache<Integer, String> cache = rcm.getCache();
```

```
CustomEventLogListener listener = new CustomEventLogListener();
try {
  cache.addClientListener(listener);
  cache.put(1, "one");
  cache.put(1, "new-one");
  cache.remove(1);
} finally {
  cache.removeClientListener(listener);
}
```

Result

Once executed, the console output should appear similar to the following:

```
ClientCacheEntryCustomEvent(eventData=ValueAddedEvent{key=1, value='one'},
eventType=CLIENT_CACHE_ENTRY_CREATED)
ClientCacheEntryCustomEvent(eventData=ValueAddedEvent{key=1, value='new-one'},
eventType=CLIENT_CACHE_ENTRY_MODIFIED)
ClientCacheEntryCustomEvent(eventData=ValueAddedEvent{key=1, value='null'},
eventType=CLIENT_CACHE_ENTRY_REMOVED
```

IMPORTANT

Converter instances must be marshallable when they are deployed in a cluster in order for conversion to occur where the event is generated, even if the event is generated in a different node to where the listener is registered. To make them marshallable, either make them extend Serializable, Externalizable, or provide a custom Externalizer for them. Both client and server need to be aware of any custom event type and be able to marshall it in order to facilitate both server and client writing against type safe APIs. On the client side, this is done by an optional marshaller configurable via the RemoteCacheManager. On the server side, this is done by a marshaller added to the Hot Rod server configuration.

10.6.6. Event Marshalling

When filtering or customizing events, the **KeyValueFilter** and **Converter** instances must be marshallable. As the client listener is installed in a cluster, the filter and/or converter instances are sent to other nodes in the cluster in order for filtering and conversion to occur where the event originates, improving efficiency. These classes can be made marshallable by having them extend Serializable or by providing and registering a custom Externalizer.

To deploy a Marshaller instance server-side, use a similar method to that used for filtering and customized events.

Deploying a Marshaller

1. Create a *JAR* file with the converter implementation within it. Each factory must have a name assigned to it via the **org.infinispan.filter.NamedFactory** annotation.

2. Create a *META-INF/services/org.infinispan.commons.marshall.Marshaller* file within the *JAR* file and within it, write the fully qualified class name of the marshaller class implementation

3. Deploy the *JAR* file in the Red Hat JBoss Data Grid by performing any of the following options:

    - Option 1: Deploy the JAR through the deployment scanner

- Copy the *JAR* to the **$JDG_HOME/standalone/deployments/** directory. The deployment scanner actively monitors this directory and will deploy the newly placed file.

- Option 2: Deploy the JAR through the CLI
  - Connect to the desired instance with the CLI:

```
[$JDG_HOME] $ bin/cli.sh --connect=$IP:$PORT
```

  - Once connected execute the **deploy** command:

```
deploy /path/to/artifact.jar
```

- Option 3: Deploy the JAR as a custom module
  - Connect to the JDG server by running the below command:

```
[$JDG_HOME] $ bin/cli.sh --connect=$IP:$PORT
```

  - The jar containing the Custom Marshaller must be defined as a module for the Server; to add this substitute the desired name of the module and the .jar name in the below command, adding additional dependencies as necessary for the Custom Marshaller:

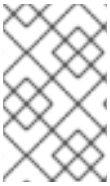
```
module add --name=$MODULE-NAME --resources=$JAR-NAME.jar --
dependencies=org.infinispan
```

  - In a different window add the newly added module as a dependency to the **org.infinispan** module by editing **$JDG_HOME/modules/system/layers/base/org/infinispan/main/module.xml**. In this file add the following entry:

```
<dependencies>
  [...]
  <module name="$MODULE-NAME">
</dependencies>
```

  - Restart the JDG server.

The Marshaller can be deployed either in a separate jar, or in the same jar as the CacheEventConverter, and/or CacheEventFilter instances.



NOTE

Only the deployment of a single Marshaller instance is supported. If multiple marshaller instances are deployed, warning messages will be displayed as a reminder indicating which marshaller instance will be used.

10.6.7. Remote Event Clustering and Failover

When a client adds a remote listener, it is installed in a single node in the cluster, which is in charge of sending events back to the client for all affected operations that occur cluster-wide.

In a clustered environment, when the node containing the listener goes down, the Hot Rod client implementation transparently fails over the client listener registration to a different node. This may result in a gap in event consumption, which can be solved using one of the following solutions.

State Delivery

The **@ClientListener** annotation has an optional **includeCurrentState** parameter, which when enabled, has the server send **CacheEntryCreatedEvent** event instances for all existing cache entries to the client. As this behavior is driven by the client it detects when the node where the listener is registered goes offline and automatically registers the listener on another node in the cluster. By enabling **includeCurrentState** clients may recompute their state or computation in the event the Hot Rod client transparently fails over registered listeners. The performance of the **includeCurrentState** parameter is impacted by the cache size, and therefore it is disabled by default.

@ClientCacheFailover

Rather than relying on receiving state, users can define a method with the **@ClientCacheFailover** annotation, which receives **ClientCacheFailoverEvent** parameter inside the client listener implementation. If the node where a Hot Rod client has registered a client listener fails, the Hot Rod client detects it transparently, and fails over all listeners registered in the node that failed to another node.

During this failover, the client may miss some events. To avoid this, the **includeCurrentState** parameter can be set to true. With this enabled a client is able to clear its data, receive all of the **CacheEntryCreatedEvent** instances, and cache these events with all keys. Alternatively, Hot Rod clients can be made aware of failover events by adding a callback handler. This callback method is an efficient solution to handling cluster topology changes affecting client listeners, and allows the client listener to determine how to behave on a failover. Near Caching takes this approach and clears the near cache upon receiving a **ClientCacheFailoverEvent**.

@ClientCacheFailover

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class EventLogListener {
// ...

   @ClientCacheFailover
   public void handleFailover(ClientCacheFailoverEvent e) {
     // Deal with client failover, e.g. clear a near cache.
   }
}
```

NOTE

The **ClientCacheFailoverEvent** is only thrown when the node that has the client listener installed fails.

CHAPTER 11. JSR-107 (JCACHE) API

11.1. JSR-107 (JCACHE) API

Starting with Red Hat JBoss Data Grid 7.2 an implementation of the JCache 1.1.0 API (JSR-107) is included. JCache specified a standard Java API for caching temporary Java objects in memory. Caching java objects can help get around bottlenecks arising from using data that is expensive to retrieve (i.e. DB or web service), or data that is hard to calculate. Caching these types of objects in memory can help speed up application performance by retrieving the data directly from memory instead of doing an expensive roundtrip or recalculation. This document specifies how to use JCache with Red Hat JBoss Data Grid's implementation of the new specification, and explains key aspects of the API.

11.2. DEPENDENCIES

The JCache dependencies may either be defined in Maven or added to the classpath; both methods are described below:

11.2.1. Option 1: Maven

In order to use the JCache implementation the following dependencies need to be added to the Maven *pom.xml* depending on how it is used:

- **embedded:**

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

<dependency>
   <groupId>javax.cache</groupId>
   <artifactId>cache-api</artifactId>
   <version>1.1.0.redhat-1</version>
</dependency>
```

- **remote:**

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

<dependency>
   <groupId>javax.cache</groupId>
   <artifactId>cache-api</artifactId>
   <version>1.1.0.redhat-1</version>
</dependency>
```

11.2.2. Option 2: Adding the necessary files to the classpath

When not using Maven the necessary jar files must be on the classpath at runtime. Having these available at runtime may either be accomplished by embedding the jar files directly, by specifying them at runtime, or by adding them into the container used to deploy the application.

Embedded Mode

1. Download the **Red Hat JBoss Data Grid 7.2.1 Library** from the Red Hat Customer Portal.
2. Extract the downloaded archive to a local directory.
3. Locate the following files:
   - *jboss-datagrid-7.2.1-library/infinispan-embedded-8.5.3.Final-redhat-00002.jar*
   - *jboss-datagrid-7.2.1-library/lib/cache-api-1.1.0.redhat-1.jar*
4. Ensure both of the above jar files are on the classpath at runtime.

Remote Mode

1. Download the **Red Hat JBoss Data Grid 7.2.1 Hot Rod Java Client** from the Red Hat Customer Portal.
2. Extract the downloaded archive to a local directory.
3. Locate the following files:
   - *jboss-datagrid-7.2.1-remote-java-client/infinispan-remote-8.5.3.Final-redhat-00002.jar*
   - *jboss-datagrid-7.2.1-remote-java-client/lib/cache-api-1.1.0.redhat-1.jar*
4. Ensure both of the above jar files are on the classpath at runtime.

11.3. CREATE A LOCAL CACHE

Creating a local cache, using default configuration options as defined by the JCache API specification, is as simple as doing the following:

```
import javax.cache.*;
import javax.cache.configuration.*;

// Retrieve the system wide cache manager
CacheManager cacheManager = Caching.getCachingProvider().getCacheManager();
// Define a named cache with default JCache configuration
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
```

WARNING

By default, the JCache API specifies that data should be stored as **storeByValue**, so that object state mutations outside of operations to the cache, won't have an impact in the objects stored in the cache. JBoss Data Grid has so far implemented this using serialization/marshalling to make copies to store in the cache, and that way adhere to the spec. Hence, if using default JCache configuration with Infinispan, data stored must be marshallable.

Alternatively, JCache can be configured to store data by reference. To do that simply call:

```
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>().setStoreByValue(false));
```

11.3.1. Library Mode

With Library mode a **CacheManager** may be configured by specifying the location of a configuration file via the **URL** parameter of **CachingProvider.getCacheManager**. This allows the opportunity to define clustered caches in a configuration file, and then obtain a reference to the preconfigured cache by passing the cache's name to the **CacheManager.getCache** method; otherwise local caches can only be used, created from the **CacheManager.createCache** .

11.3.2. Client-Server Mode

With Client-Server mode specific configurations of a remote **CacheManager** is performed by passing standard HotRod client properties via **properties** parameter of **CachingProvider.getCacheManager**. The remote servers referenced must be running and able to receive the request.

If not specified the default address and port will be used (127.0.0.1:11222). In addition, contrary to Library mode, the first time a cache reference is obtained **CacheManager.createCache** must be used so that the cache may be registered internally. Subsequent queries may be performed via **CacheManager.getCache**.

11.4. STORE AND RETRIEVE DATA

Even though the JCache API does not extend either java.util.Map or java.util.concurrent.ConcurrentMap it provides a key/value API to store and retrieve data:

```
import javax.cache.*;
import javax.cache.configuration.*;

CacheManager cacheManager = Caching.getCachingProvider().getCacheManager();
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
cache.put("hello", "world"); // Notice that javax.cache.Cache.put(K) returns void!
String value = cache.get("hello"); // Returns "world"
```

Contrary to standard java.util.Map, javax.cache.Cache comes with two basic put methods called **put** and **getAndPut**. The former returns **void** whereas the latter returns the previous value associated with the key. The equivalent of java.util.Map.put(K) in JCache is javax.cache.Cache.getAndPut(K).

TIP

Even though JCache API only covers standalone caching, it can be plugged with a persistence store, and has been designed with clustering or distribution in mind. The reason why javax.cache.Cache offers two put methods is because standard java.util.Map put call forces implementors to calculate the previous value. When a persistent store is in use, or the cache is distributed, returning the previous value could be an expensive operation, and often users call standard http://docs.oracle.com/javase/7/docs/api/java/util/Map.html#put(K, V)[java.util.Map.put(K)] without using the return value. Hence, JCache users need to think about whether the return value is relevant to them, in which case they need to call javax.cache.Cache.getAndPut(K) , otherwise they can call java.util.Map.put(K) which avoids returning the potentially expensive operation of returning the previous value.

11.5. COMPARING JAVA.UTIL.CONCURRENT.CONCURRENTMAP AND JAVAX.CACHE.CACHE APIS

Here is a brief comparison of the data manipulation APIs provided by java.util.concurrent.ConcurrentMap and javax.cache.Cache APIs:

Table 11.1. java.util.concurrent.ConcurrentMap and javax.cache.Cache Comparison

Operation	java.util.concurrent.ConcurrentMap<K,V>	javax.cache.Cache<K,V>
store and no return	N/A	`void put(K key)`
store and return previous value	`V put(K key)`	`V getAndPut(K key)`
store if not present	`V putIfAbsent(K key, V Value)`	`boolean putIfAbsent(K key, V value)`
retrieve	`V get(Object key)`	`V get(K key)`
delete if present	`V remove(Object key)`	`boolean remove(K key)`
delete and return previous value	`V remove(Object key)`	`V getAndRemove(K key)`

Operation	java.util.concurrent.ConcurrentMap<K,V>	javax.cache.Cache<K,V>
delete conditional	`boolean remove(Object key, Object value)`	`boolean remove(K key, V oldValue)`
replace if present	`V replace(K key, V value)`	`boolean replace(K key, V value)`
replace and return previous value	`V replace(K key, V value)`	`V getAndReplace(K key, V value)`
replace conditional	`boolean replace(K key, V oldValue, V newValue)`	`boolean replace(K key, V oldValue, V newValue)`

Comparing the two APIs it can be seen that, where possible, JCache avoids returning the previous value to avoid operations doing expensive network or IO operations. This is an overriding principle in the design of the JCache API. In fact, there is a set of operations that are present in java.util.concurrent.ConcurrentMap, but are not present in the javax.cache.Cache because they could be expensive to compute in a distributed cache. The only exception is iterating over the contents of the cache:

Table 11.2. javax.cache.Cache avoiding returns

Operation	java.util.concurrent.ConcurrentMap<K,V>	javax.cache.Cache<K,V>
calculate size of cache	`int size()`	N/A
return all keys in the cache	`Set<K> keySet()`	N/A
return all values in the cache	`Collection<V> values()`	N/A
return all entries in the cache	`Set<Map.Entry<K, V>> entrySet()`	N/A

Operation	java.util.concurrent.ConcurrentMap<K,V>	javax.cache.Cache<K,V>
iterate over the cache	use **iterator()** method on keySet, values, or entrySet	Iterator<Cache.Entry<K, V>> iterator()

11.6. CLUSTERING JCACHE INSTANCES

Red Hat JBoss Data Grid implementation goes beyond the specification in order to provide the possibility to cluster caches using the standard API. Given a configuration file to replicate caches such as:

```
<namedCache name="namedCache">
   <clustering mode="replication"/>
</namedCache>
```

It is possible to create a cluster of caches using this code:

```
import javax.cache.*;
import java.net.URI;

// For multiple cache managers to be constructed with the standard JCache API
// and live in the same JVM, either their names, or their classloaders, must
// be different.
// This example shows how to force their classloaders to be different.
// An alternative method would have been to duplicate the XML file and give
// it a different name, but this results in unnecessary file duplication.
ClassLoader tccl = Thread.currentThread().getContextClassLoader();
CacheManager cacheManager1 = Caching.getCachingProvider().getCacheManager(
     URI.create("infinispan-jcache-cluster.xml"), new TestClassLoader(tccl));
CacheManager cacheManager2 = Caching.getCachingProvider().getCacheManager(
     URI.create("infinispan-jcache-cluster.xml"), new TestClassLoader(tccl));

Cache<String, String> cache1 = cacheManager1.getCache("namedCache");
Cache<String, String> cache2 = cacheManager2.getCache("namedCache");

cache1.put("hello", "world");
String value = cache2.get("hello"); // Returns "world" if clustering is working

// --

public static class TestClassLoader extends ClassLoader {
  public TestClassLoader(ClassLoader parent) {
    super(parent);
  }
}
```

11.7. MULTIPLE CACHING PROVIDERS

Caching providers are obtained from **javax.cache.Caching** using the overloaded

getCachingProvider() method; by default this method will attempt to load any **META-INF/services/javax.cache.spi.CachingProvider** files found in the classpath. If one is found it will determine the caching provider in use.

With multiple caching providers available a specific provider may be selected using either of the following methods:

- **getCachingProvider(ClassLoader classLoader)**
- **getCachingProvider(String fullyQualifiedClassName)**

To switch between caching providers ensure that the appropriate provider is available in the default classpath, or select it using one of the above methods.

All **javax.cache.spi.CachingProviders** that are detected or have been loaded by the **Caching** class are maintained in an internal registry, and subsequent requests for the same caching provider will be returned from this registry instead of being reloaded or reinstantiating the caching provider implementation. To view the current caching providers either of the following methods may be used:

- **getCachingProviders()** - provides a list of caching providers in the default class loader.
- **getCachingProviders(ClassLoader classLoader)** - provides a list of caching providers in the specified class loader.

CHAPTER 12. THE HEALTH CHECK API

12.1. THE HEALTH CHECK API

The Health Check API allows users to monitor the health of the cluster, and the caches contained within. This information is particularly important when working in a cloud environment, as it provides a method of querying to report the status of the cluster or cache.

This API exposes the following information:

- The name of the cluster.
- The number of machines in the cluster.
- The overall status of the cluster or cache, represented in one of three values:
  - Healthy - The entity is healthy.
  - Unhealthy - The entity is unhealthy. This value indicates that one or more caches are in a degraded state.
  - Rebalancing - The entity is operational, but a rebalance is in progress. Cluster nodes should not be adjusted when this value is reported.
- The status of each cache.
- A tail of the server log.

For information on using the Health Check API through non-programmatic methods, refer to the JBoss Data Grid **Administration and Configuration Guide**.

12.2. ACCESSING THE HEALTH CHECK API PROGRAMMATICALLY

The Health Check API is only accessible programatically in Library mode, and may be accessed by calling the **embeddedCacheManager.getHealth()** method.

This method returns an **org.infinispan.health.Health** object, which has access to the following methods:

- **getClusterHealth()** - returns a **ClusterHealth** object with access to the following methods:
  - **getNumberOfNodes()** - returns an int representing the number of all nodes in the cluster
  - **getNodeNames()** - returns a **List<String>** containing the names of all nodes in the cluster
  - **getClusterName()** - returns a **String** containing the name of the cluster
  - **getHealthStatus()** - returns a **HealthStatus** that contains the cluster's health, being reported as **HEALTHY**, **UNHEALTHY**, or **REBALANCING**
- **getHostInfo()** - returns a **HostInfo** object with access to the following methods:
  - **getNumberOfCpus()** - returns an int containing the number of CPUs installed in the host
  - **getTotalMemoryKb()** - returns a long containing the total memory in KB
  - **getFreeMemoryInKb()** - returns a long containing the free memory in KB

- **getCacheHealth()** - returns a **List<CacheHealth>**. Each **CacheHealth** object has access to the following methods:
  - **getCacheName()** - returns a **String** containing the name of the cache
  - **getStatus()** - returns a **HealthStatus** that contains the cache's health, being reported as HEALTHY, UNHEALTHY, or REBALANCING

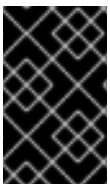
CHAPTER 13. THE REST API

13.1. THE REST INTERFACE

Red Hat JBoss Data Grid provides a REST interface, allowing for loose coupling between the client and server. Its primary benefit is interoperability with existing HTTP clients, along with providing a connection for php clients. In addition, the need for specific versions of client libraries and bindings is eliminated.

The REST API introduces an overhead, and requires a REST client or custom code to understand and create REST calls. It is recommended to use the Hot Rod client where performance is a concern.

To interact with Red Hat JBoss Data Grid's REST API only a HTTP client library is required. For Java, this may be the Apache HTTP Commons Client, or the java.net API.



IMPORTANT

The following examples assume that REST security is disabled on the REST connector. To disable REST security remove the **authentication** and **encryption** elements from the connector.

13.2. RUBY CLIENT CODE

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

Using the REST API with Ruby

```
require 'net/http'

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

#An example of how to create a new entry

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

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

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

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

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

#An example of Removing the remote copy of the key

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

#An example of creating binary data

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

13.3. USING JSON WITH RUBY EXAMPLE

Prerequisites

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

```
require 'json'
```

Using JSON with Ruby

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

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

13.4. PYTHON CLIENT CODE

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

Using the REST API with Python

```
import httplib

#How to insert data

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

#How to retrieve data

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

13.5. JAVA CLIENT CODE

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

Defining Imports

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

Adding a String Value to a Cache

```
// Using the imports in the previous example
public class RestExample {

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

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

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

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

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

    connection.disconnect();
  }
```

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

Get a String Value from a Cache

```
  // Continuation of RestExample defined in previous example
 /**
  * Method that gets an value by a key in url as param value.
  * @param urlServerAddress
  * @return String value
  * @throws IOException
  */
  public String getMethod(String urlServerAddress) throws IOException {
    String line = new String();
    StringBuilder stringBuilder = new StringBuilder();
```

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

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

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

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

        connection.connect();

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

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

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

        connection.disconnect();

        return stringBuilder.toString();
    }
```

Using a Java Main Method

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

13.6. USING THE REST INTERFACE

13.6.1. REST Interface Operations

In Remote Client-Server mode, Red Hat JBoss Data Grid provides a REST interface that allows clients to:

- Add data
- Retrieve data
- Remove data
- Query data

13.6.1.1. Data Formats

The REST API exposes caches that store data in a format defined by a configurable media type.

The following XML snippet shows an example configuration that defines the media type for keys and values:

```
<cache>
  <encoding>
    <key media-type="application/x-java-object; type=java.lang.Integer"/>
    <value media-type="application/xml; charset=UTF-8"/>
  </encoding>
</cache>
```

For more information, see Configuring Media Types.

13.6.1.2. Headers

Calls to the Red Hat JBoss Data Grid REST API can provide headers that describe:

- Content written to the cache.
- Required data format of the content when reading from the cache.

JBoss Data Grid supports the HTTP/1.1 **Content-Type** and **Accept** headers applied to values as well as the **Key-Content-Type** header for keys.

13.6.1.3. Accept Header

The Red Hat JBoss Data Grid REST server complies with the RFC-2616 specification for **Accept** headers and negotiates the correct media type based on the supported conversions.

For example, a client sends the following header in a call to read data from the cache:

```
Accept: text/plain;q=0.7, application/json;q=0.8, */*;q=0.6
```

In this case, JBoss Data Grid gives precedence to **JSON** format during negotiation because that media type has highest priority (0.8). If the server does not support JSON format, **text/plain** takes precedence because the media type has the next highest priority (0.7).

In the event that the server does not support either the **JSON** or **text/plain** media types, / takes precedence, which indicates any suitable media type based on the cache configuration.

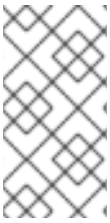
When the negotiation completes, the server continues using the chosen media type for the operation. If any errors occur during the operation, the server does not attempt to use any other media type.

13.6.1.4. Key-Content-Type Header

Most calls to the REST API include the key in the URL. When handling those calls, Red Hat JBoss Data Grid uses the **java.lang.String** as the content type for keys by default. However, you can use the **Key-Content-Type** header to specify different content types for keys.

Table 13.1. Key-Content-Type Header Examples

Use	API Call	Header
Specify a **byte[]** key as a Base64 string	**PUT /my-cache/AQIDBDM=**	**Key-Content-Type: application/octet-stream**
Specify a **byte[]`** key as a hexadecimal string	**GET /my-cache/0x01CA03042F**	**Key-Content-Type: application/octet-stream; encoding=hex**
Specify a double key	**POST /my-cache/3.141456**	**Key-Content-Type: application/x-java-object;type=java.lang.Double**



NOTE

The **type** parameter for **application/x-java-object** is restricted to primitive wrapper types and **java.lang.String**. This parameter is also restricted to bytes, with the result that **application/x-java-object;type=Bytes** is equivalent to **application/octet-stream;encoding=hex**.

13.6.2. Adding Data Through the REST API

13.6.2.1. Adding Data to the Cache

Add data to the cache with the following methods:

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

When you call the REST API with the **PUT** and **POST** methods, the body of the request contains the data.

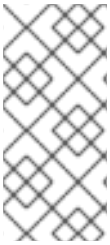
13.6.2.2. PUT /{cacheName}/{cacheKey}

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

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

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

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



NOTE

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

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

13.6.2.3. POST /{cacheName}/{cacheKey}

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

13.6.2.4. Headers for the PUT and POST Methods

Table 13.2. Headers for PUT and POST Methods

Header	Optional or Required	Description
Key-Content-Type	Optional	Specifies the content type for the key in the URL.
Content-Type	Optional	Specifies the media type of the value sent to the REST API.
performAsync	Optional	Specifies a boolean value. If the value is **true**, it returns immediately and then independently replicates data to the cluster, which is useful when inserting data in bulk or for large clusters.
timeToLiveSeconds	Optional	Specifies the number of seconds before the entry is automatically deleted. Negative values create entries that are never deleted.
maxIdleTimeSeconds	Optional	Specifies the number of seconds that the entry can remain idle before it is deleted. Negative values create entries that are never deleted.

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

- If both the **timeToLiveSeconds** and **maxIdleTimeSeconds** headers are assigned the value **0**, the cache uses the default **timeToLiveSeconds** and **maxIdleTimeSeconds** values configured either using ` XML ` or programatically.
- If only the **maxIdleTimeSeconds** header value is set to **0**, the **timeToLiveSeconds** value

should be passed as the parameter (or the default **-1**, if the parameter is not present). Additionally, the **maxIdleTimeSeconds** parameter value defaults to the values configured either using ` XML` or programatically.

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

13.6.3. Retrieving Data Through the REST API

13.6.3.1. Retrieving Data from the Cache

Retrieve data from the cache with the following methods:

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

13.6.3.2. GET /{cacheName}/{cacheKey}

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

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

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

Appending the **extended** parameter to the query returns additional information. For example,

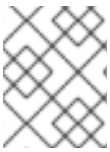
```
GET /{cacheName}/{cacheKey}?extended
```

Returns the following custom headers:

- **Cluster-Primary-Owner** which identifies the node that is the primary owner of the key.
- **Cluster-Node-Name** which specifies the JGroups node name of the server that handled the request.
- **Cluster-Physical-Address** which specifies the physical JGroups address of the server that handled the request.

13.6.3.3. HEAD /{cacheName}/{cacheKey}

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



NOTE

The **HEAD** method also supports the **extended** parameter to return additional information.

13.6.3.4. GET /{cacheName}

The **GET** method can return a list of keys that reside in the cache. The list of keys is returned in the body of the response.

The **Accept** header can format the response as follows:

- **application/xml** returns a list of keys in XML format.
- **application/json** returns a list of keys in JSON format.
- **text/plain** returns a list of keys in plain text with one key per line.

If the cache is distributed then only keys that are owned by the node that handles the request are returned. To return all keys, append the **global** parameter to the query as follows:

```
GET /{cacheName}?global
```

13.6.3.5. Headers for the GET and HEAD Methods

Table 13.3. Headers for GET and HEAD Methods

Header	Optional or Required	Description
Key-Content-Type	Optional	Specifies the content type for the key in the URL. Defaults to **application/x-java-object; type=java.lang.String** if not specified.
Accept	Optional	Specifies the format in which to return the content for calls with the **GET** method.

13.6.4. Removing Data Through the REST API

13.6.4.1. Removing Data from the Cache

Remove data from Red Hat JBoss Data Grid with the HTTP **DELETE** method.

The **DELETE** method can:

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

13.6.4.2. DELETE /{cacheName}/{cacheKey}

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

13.6.4.3. DELETE /{cacheName}

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

13.6.4.4. Background Delete Operations

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

13.6.5. ETag Based Headers

ETag Based Headers

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

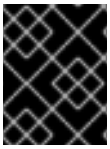
Table 13.4. Entity Tag Related Headers

Header	Algorithm	Example	Description
If-Match	If-Match = "If-Match" ":" ("*" | 1#entity-tag)	-	Used in conjunction with a list of associated entity tags to verify that a specified entity (that was previously obtained from a resource) remains current.
If-None-Match		-	Used in conjunction with a list of associated entity tags to verify that none of the specified entities (that was previously obtained from a resource) are current. This feature facilitates efficient updates of cached information when required and with minimal transaction overhead.

Header	Algorithm	Example	Description
If-Modified-Since	If-Modified-Since = "If-Modified-Since" ":" HTTP-date	If-Modified-Since: Sat, 29 Oct 1994 19:43:31 GMT	Compares the requested variant's last modification time and date with a supplied time and date value. If the requested variant has not been modified since the specified time and date, a **304** (not modified) response is returned without a message-body instead of an entity.
If-Unmodified-Since	If-Unmodified-Since = "If-Unmodified-Since" ":" HTTP-date	If-Unmodified-Since: Sat, 29 Oct 1994 19:43:31 GMT	Compares the requested variant's last modification time and date with a supplied time and date value. If the requested resources has not been modified since the supplied date and time, the specified operation is performed. If the requested resource has been modified since the supplied date and time, the operation is not performed and a **412** (Precondition Failed) response is returned.

13.6.6. Querying Data via the REST Interface

Red Hat JBoss Data Grid lets you query data via the REST interface using Ickle queries in JSON format.



IMPORTANT

Querying data via the REST interface is a Technology Preview feature in JBoss Data Grid 7.2.

13.6.6.1. JSON to Protostream Conversion

JBoss Data Grid uses protocol buffers to efficiently store data in binary format in the cache while exposing queries and enabling you to read and write content in JSON format.

To store Protobuf encoded entries, the cache must be configured with the **application/x-protostream** media type. JBoss Data Grid then automatically converts JSON to Protobuf.

If the cache is indexed, you do not need to perform any configuration. By default, an indexed cache stores entries with the **application/x-protostream** media type.

However, if the cache is not indexed, you must configure keys and values with the **application/x-protostream** media type, as in the following example:

```
<cache>
  <encoding>
    <key media-type="application/x-protostream"/>
    <value media-type="application/x-protostream"/>
  </encoding>
</cache>
```

13.6.6.2. Registering Protobuf Schemas

To register a Protobuf schema, you can use the HTTP **POST** method to insert the schema in the **___protobuf_metadata** cache, as in the following example:

```
curl -u user:password -X POST --data-binary @./schema.proto
http://127.0.0.1:8080/rest/___protobuf_metadata/schema.proto
```

For more information about Protobuf encoding and registering Protobuf schemas, see Protobuf Encoding.

13.6.6.3. Mapping JSON Documents to Protobuf Messages

The **_type** field must be included in JSON documents. This field identifies the Protobuf message to which the JSON document corresponds.

For example, the following is a Protobuf message defined as **Person**:

```
message Person {
  required string name = 1;
  required int32 age = 2;
}
```

The corresponding JSON document is as follows:

Person.json

```
{
  "_type": "Person",
  "name": "user1",
  "age": 32
}
```

13.6.6.4. Populating the Cache

You can write content to the cache in JSON format as follows:

```
curl -u user:user -XPOST --data-binary @./Person.json -H "Content-Type: application/json;
charset=UTF-8" http://127.0.0.1:8080/rest/{cacheName}/{key}
```

- **{cacheName}** specifies the name of the cache to query.
- **{key}** specifies the name of the key that stores the data in the cache.

After you write content to the cache, you can read it in JSON format as follows:

```
curl -u user:user http://127.0.0.1:8080/rest/{cacheName}/{key}
```

13.6.6.5. Querying REST Endpoints

Use the HTTP **GET** method or the **POST** method to query data via the REST interface.

Query requests with the **GET** method have the following structure:

```
{cacheName}?action=search&query={ickle query}
```

- **{cacheName}** specifies the name of the cache to query.
- **{ickle query}** specifies the Ickle query to perform.

The following are example queries:

- Return all data from the entity named **Person**: **http://localhost:8080/rest/mycache?action=search&query=from Person**
- Refine the query with a *select* clause: **http://localhost:8080/rest/mycache?action=search&query=Select name, age from Person**
- Group the results of the query: **http://localhost:8080/rest/mycache?action=search&query=from Person group by age**

Query requests with the **POST** method have the following structure:

```
{cacheName}?action=search
```

The body of the request specifies the query and any parameters in JSON format.

The following is an example query that returns data from the entity named **Entity** and filters results using a *where* clause:

```
{
 "query":"from Entity where name:\"user1\"",
 "max_results":20,
 "offset":10
}
```

13.6.6.5.1. Optional Request Parameters

The following optional parameters can apply to query requests:

Parameter	Description

Parameter	Description
max_results	Limits the results of the query to a maximum number. The default value is **10**.
offset	Specifies the index of the first result to return. The default value is **0**.
query_mode	Specifies how the server executes the query with the following values: **BROADCAST** broadcasts a query to each node in the cluster and then retrieves and combines the results of the query before returning them. This execution mode is suitable for non-shared indexes where each node contains a subset of data in its index. **FETCH** executes the query in the node that the query calls. This execution mode is suitable where all of the indexes for data across the cluster are available locally. This is the default value.

13.6.6.5.2. Query Results

Results of Ickle queries are returned in JSON format as in the following example:

```
{
  "total_results" : 150,
  "hits" : [ {
    "hit" : {
      "name" : "user1",
      "age" : 35
    }
  }, {
    "hit" : {
      "name" : "user2",
      "age" : 42
    }
  }, {
    "hit" : {
      "name" : "user3",
      "age" : 25
    }
  } ]
}
```

- **total_results** is the number of results that the query returned.
- **hits** lists all results that match the query.
- **hit** contains the fields for each result in the query.

For more information about Ickle queries, see Building a Query using the Ickle Query Language .

CHAPTER 14. CLUSTERED COUNTERS

Clustered counters are distributed and shared across nodes in a Red Hat JBoss Data Grid cluster. Clustered counters allow you to record the count of objects.

Clustered counters are identified by their names and are initialized with a value, which defaults to 0. Clustered counters can also be persisted so that the values are kept after cluster restarts.

There are two types of clustered counter:

- **Strong** stores the counter value in a single key for consistency. During updates to the counter, the value is known. Updates to the counter value are performed under the key lock. However, reads of the current value of the counter do not acquire any lock. Strong counters allow the counter value to be bounded and provide atomic operations such as **compareAndSet** or **compareAndSwap**.

- **Weak** stores the counter value in multiple keys. Each key stores a partial state of the counter value and can be updated concurrently. During updates to the counter, the value is not known. Retrieving the counter value does not always return the current, up to date value.

Both strong and weak clustered counters support updating the counter value, return the current value of a counter, and provide events when a counter value is updated.

14.1. THE COUNTER API

The **counter** API consists of the following:

- **EmbeddedCounterManagerFactory** initializes a counter manager from an embedded cache manager.

- **RemoteCounterManagerFactory** initializes a counter manager from a remote cache manager.

- **CounterManager** provides methods to create, define, and return counters.

- **StrongCounter** implements strong counters. This interface provides atomic updates for a counter. All operations are performed asynchronously and use the **CompletableFuture** class for completion logic.

- **SyncStrongCounter** implements synchronous strong counters.

- **WeakCounter** implements weak counters. All operations are performed asynchronously and use the **CompletableFuture** class for completion logic.

- **SyncWeakCounter** implements synchronous weak counters.

- **CounterListener** listens for changes to strong counters.

- **CounterEvent** returns events when changes to strong counters occur.

- **Handle** extends the **CounterListener** interface.

14.2. ADDING MAVEN DEPENDENCIES

To start using clustered counters, add the following dependency to **pom.xml**:

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-counter</artifactId>
  <version>...</version> <!-- 7.2.0 or later -->
</dependency>
```

14.3. RETRIEVING THE COUNTERMANAGER INTERFACE

To use clustered counters in Red Hat JBoss Data Grid embedded mode, do the following:

```
// Create or obtain an EmbeddedCacheManager.
EmbeddedCacheManager manager = ...;

// Retrieve the CounterManager interface.
CounterManager counterManager =
EmbeddedCounterManagerFactory.asCounterManager(manager);
```

To use clustered counters with a Hot Rod client that interacts with a Red Hat JBoss Data Grid remote server, do the following:

```
// Create or obtain a RemoteCacheManager.
RemoteCacheManager manager = ...;

// Retrieve the CounterManager interface.
CounterManager counterManager = RemoteCounterManagerFactory.asCounterManager(manager);
```

14.4. USING CLUSTERED COUNTERS

You can define and configure clustered counters in the **cache-container** XML configuration or programmatically.

14.4.1. XML Configuration for Clustered Counters

The following XML snippet provides an example of a clustered counters configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan>
   <cache-container>
      <!-- cache container configuration goes here -->
      <!-- cache configuration goes here -->
      <counters>
         <strong-counter name="counter-1" initial-value="1">
            <upper-bound value="10"/>
         </strong-counter>
         <strong-counter name="counter-2" initial-value="2"/>
         <weak-counter name="counter-3" initial-value="3"/>
       </counters>
   </cache-container>
</infinispan>
```

14.4.1.1. XML Definition

The **counters** element configures counters for a cluster and has the following attributes:

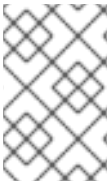
- **num-owners** sets the number of copies of each counter to store across the cluster. A smaller number results in faster update operations but supports a lower number of server crashes. The value must be a positive number. The default value is **2**.
- **reliability** sets the counter update behavior in a network partition and takes the following values:
  - **AVAILABLE** all partitions can read and update the value of the counter. This is the default value.
  - **CONSISTENT** the primary partition can read and update the value of the counter. The remaining partitions can only read the value of the counter.

The **strong-counter** element creates and defines a strong clustered counter. The **weak-counter** element creates and defines a weak clustered counter. The following attributes are common to both elements:

- **initial-value** sets the initial value of the counter. The default value is **0**.
- **storage** configures how counter values are stored. This attribute determines if the counter values are saved after the cluster shuts down and restarts. This attribute takes the following values:
  - **VOLATILE** stores the value of the counter in memory. The value of the counter is discarded when the cluster shuts down. This is the default value.
  - **PERSISTENT** stores the value of the counter in a private, local persistence store. The value of the counter is saved when the cluster shuts down and restarts.

Attributes specific to the **strong-counter** element are as follows:

- **lower-bound** sets the lower bound of a strong counter. The default value is **Long.MIN_VALUE**.
- **upper-bound** sets the upper bound of a strong counter. The default value is **Long.MAX_VALUE**.



NOTE

The value of the **initial-value** attribute must be between the **lower-bound** value and the **upper-bound** value. If you do not specify a lower and upper bound for a strong counter, the counter is not bounded.

Attributes specific to the **weak-counter** element are as follows:

- **concurrency-level** sets the maximum number of concurrent updates to the value of a counter. The value must be a positive number. The default value is **16**.

14.4.2. Run-time Configuration of Clustered Counters

You can configure clustered counters on-demand at run-time after the **EmbeddedCacheManager** is initialized, as in the following example:

```
CounterManager manager = ...;
```

```
// Create three counters.
// The first counter is a strong counter bounded to 10.
manager.defineCounter("counter-1",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(1).upperBound(10).buil
d());

// The second counter is an unbounded strong counter.
manager.defineCounter("counter-2",
CounterConfiguration.builder(CounterType.UNBOUNDED_STRONG).initialValue(2).build());

// The third counter is a weak counter.
manager.defineCounter("counter-3",
CounterConfiguration.builder(CounterType.WEAK).initialValue(3).build());
```

The **defineCounter()** method returns **true** if the counter is defined successfully or **false** if not. If the counter configuration is not valid, a **CounterConfigurationException** exception is thrown.

TIP

Use the **isDefined()** method to determine if a counter is already defined, as in the following example:

```
CounterManager manager = ...
if (!manager.isDefined("someCounter")) {
   manager.define("someCounter", ...);
}
```

14.4.3. Programmatic Configuration of Clustered Counters

The following code sample illustrates how to configure clustered counters programmatically with the **GlobalConfigurationBuilder**:

```
// Set up a clustered cache manager.
GlobalConfigurationBuilder global = GlobalConfigurationBuilder.defaultClusteredBuilder();

// Create a counter configuration builder.
CounterManagerConfigurationBuilder builder =
global.addModule(CounterManagerConfigurationBuilder.class);

// Create three counters.
// The first counter is a strong counter bounded to 10.
builder.addStrongCounter().name("counter-1").upperBound(10).initialValue(1);

// The second counter is an unbounded strong counter.
builder.addStrongCounter().name("counter-2").initialValue(2);

// The third counter is a weak counter.
builder.addWeakCounter().name("counter-3").initialValue(3);

// Initialize a new default cache manager.
DefaultCacheManager cacheManager = new DefaultCacheManager(global.build());
```

14.4.3.1. Using Clustered Counters

The following code example illustrates how you can use clustered counters that you create and define programmatically:

```
// Retrieve the CounterManager interface from the cache manager.
CounterManager counterManager =
EmbeddedCounterManagerFactory.asCounterManager(cacheManager);

// Strong counters provide greater consistency than weak counters.
// The value of a strong counter is known during an increment or decrement operation.
// The value of a strong counter can also be bounded in cases where a limit is required.
StrongCounter counter1 = counterManager.getStrongCounter("counter-1");

// All methods are asynchronous and return CompletableFuture objects so that you can perform other
operations while the counter value is computed.
counter1.getValue().thenAccept(value -> System.out.println("Counter-1 initial value is " +
value)).get();

// Attempt to add a value that exceeds the upper-bound value.
counter1.addAndGet(10).handle((value, throwable) -> {
   // Value is null since the counter is bounded to a maximum of 10.
   System.out.println("Counter-1 Exception is " + throwable.getMessage());
   return 0;
}).get();

// Check the counter value. The value should be 10.
counter1.getValue().thenAccept(value -> System.out.println("Counter-1 value is " + value)).get();

//Decrement the counter value. The new value should be 9.
counter1.decrementAndGet().handle((value, throwable) -> {
   // No exception is thrown.
   System.out.println("Counter-1 new value is " + value);
   return value;
}).get();

// The second counter, counter2, is a strong counter that is unbounded. It never throws the
CounterOutOfBoundsException.
StrongCounter counter2 = counterManager.getStrongCounter("counter-2");

// All counters allow a listener to be registered.
// The handle interface can remove the listener.
counter2.addListener(event -> System.out
   .println("Counter-2 event: oldValue=" + event.getOldValue() + " newValue=" +
event.getNewValue()));

// Adding MAX_VALUE does not throw an exception.
// No increments take effect if the value exceeds the MAX_VALUE.
counter2.addAndGet(Long.MAX_VALUE).thenAccept(aLong -> System.out.println("Counter-2 value
is " + aLong)).get();

// Conditional operations are allowed in strong counters.
counter2.compareAndSet(Long.MAX_VALUE, 0)
   .thenAccept(aBoolean -> System.out.println("Counter-2 CAS result is " + aBoolean)).get();
counter2.getValue().thenAccept(value -> System.out.println("Counter-2 value is " + value)).get();

// Reset the value of the second counter to its initial value.
counter2.reset().get();
```

```
counter2.getValue().thenAccept(value -> System.out.println("Counter-2 initial value is " +
value)).get();

// Retrieve the third counter, counter3.
WeakCounter counter3 = counterManager.getWeakCounter("counter-3");

// The value of weak counters is not available during update operations. As a result these counters
can increment faster than strong counters.
// The counter value is computed lazily and stored locally.
counter3.add(5).thenAccept(aVoid -> System.out.println("Adding 5 to counter-3 completed!")).get();

// Check the counter value.
System.out.println("Counter-3 value is " + counter3.getValue());

// Stop the cache manager and release all resources.
cacheManager.stop();
```

CHAPTER 15. CLUSTERED LOCKS

Clustered locks are data structures that are distributed and shared across nodes in a Red Hat JBoss Data Grid cluster. Clustered locks allow you to run code that is synchronized between the nodes in a cluster.

15.1. THE LOCK API

The **lock** API consists of the following:

- **EmbeddedClusteredLockManagerFactory** initializes a clustered lock manager from an embedded cache manager.
- **ClusteredLockManager** provides methods to define, configure, retrieve, and remove clustered locks.
- **ClusteredLock** provides methods to implement clustered locks.

NOTE

Clustered locks are available in Red Hat JBoss Data Grid embedded mode only.

15.2. SUPPORTED CONFIGURATION

As of this release, Red Hat JBoss Data Grid supports **NODE** ownership and non-reentrant clustered locks.

NODE ownership allows all nodes in the Red Hat JBoss Data Grid cluster to use a lock.

Reentrant locks allow the node that owns the lock to acquire it again while the node has ownership of the lock. Non-reentrant locks allow any node to acquire the lock. As a result, if two consecutive lock calls are sent for the same owner, the first call acquires the lock if it is available and the second call is blocked.

15.3. ADDING MAVEN DEPENDENCIES

To start using clustered locks, add the following dependency to **pom.xml**:

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-lock</artifactId>
  <version>...</version> <!-- 7.2.0 or later -->
</dependency>
```

15.4. USING CLUSTERED LOCKS

The following code sample illustrates how to use clustered locks:

```
// Set up a clustered cache manager.
GlobalConfigurationBuilder global = GlobalConfigurationBuilder.defaultClusteredBuilder();

// Configure the cache mode as distributed and synchronous.
```

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clustering().cacheMode(CacheMode.DIST_SYNC);

// Initialize a new default cache manager.
DefaultCacheManager cm = new DefaultCacheManager(global.build(), builder.build());

// Initialize a clustered lock manager from the cache manager.
ClusteredLockManager clm1 = EmbeddedClusteredLockManagerFactory.from(cm);

// Define a clustered lock named 'lock' with the default configuration.
clm1.defineLock("lock");

// Get a lock from each node in the cluster.
ClusteredLock lock = clm1.get("lock");

AtomicInteger counter = new AtomicInteger(0);

// Acquire the lock as follows.
// Each 'lock.tryLock(1, TimeUnit.SECONDS)" method attempts to acquire the lock.
// If the lock is not available, the method waits for the timeout period to elapse. When the lock is
acquired, other calls to acquire the lock are blocked until the lock is released.
CompletableFuture<Boolean> call1 = lock.tryLock(1, TimeUnit.SECONDS).whenComplete((r, ex) -> {
   if (r) {
      System.out.println("lock is acquired by the call 1");
      lock.unlock().whenComplete((nil, ex2) -> {
         System.out.println("lock is released by the call 1");
         counter.incrementAndGet();
      });
   }
});

CompletableFuture<Boolean> call2 = lock.tryLock(1, TimeUnit.SECONDS).whenComplete((r, ex) -> {
   if (r) {
      System.out.println("lock is acquired by the call 2");
      lock.unlock().whenComplete((nil, ex2) -> {
         System.out.println("lock is released by the call 2");
         counter.incrementAndGet();
      });
   }
});

CompletableFuture<Boolean> call3 = lock.tryLock(1, TimeUnit.SECONDS).whenComplete((r, ex) -> {
   if (r) {
      System.out.println("lock is acquired by the call 3");
      lock.unlock().whenComplete((nil, ex2) -> {
         System.out.println("lock is released by the call 3");
         counter.incrementAndGet();
      });
   }
});

CompletableFuture.allOf(call1, call2, call3).whenComplete((r, ex) -> {
   // Print the value of the counter.
   System.out.println("Value of the counter is " + counter.get());
```

```
    // Stop the cache manager.
    cm.stop();
});
```

CHAPTER 16. THE HOT ROD INTERFACE

16.1. ABOUT HOT ROD

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

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

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

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

16.2. HOT ROD HEADERS

16.2.1. Hot Rod Header Data Types

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

Table 16.1. Header Data Types

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

16.2.2. Request Header

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

Table 16.2. Request Header Fields

Field Name	Data Type/Size	Details

Field Name	Data Type/Size	Details
Magic	1 byte	Indicates whether the header is a request header or response header.
Message ID	vLong	Contains the message ID. Responses use this unique ID when responding to a request. This allows Hot Rod clients to implement the protocol in an asynchronous manner.
Version	1 byte	Contains the Hot Rod server version.
Opcode	1 byte	Contains the relevant operation code. In a request header, opcode can only contain the request operation codes.
Cache Name Length	vInt	Stores the length of the cache name. If Cache Name Length is set to **0** and no value is supplied for Cache Name, the operation interacts with the default cache.
Cache Name	string	Stores the name of the target cache for the specified operation. This name must match the name of a predefined cache in the cache configuration file.
Flags	vInt	Contains a numeric value of variable length that represents flags passed to the system. Each bit represents a flag, except the most significant bit, which is used to determine whether more bytes must be read. Using a bit to represent each flag facilitates the representation of flag combinations in a condensed manner.
Client Intelligence	1 byte	Contains a value that indicates the client capabilities to the server.

Field Name	Data Type/Size	Details
Topology ID	vInt	Contains the last known view ID in the client. Basic clients supply the value **0** for this field. Clients that support topology or hash information supply the value **0** until the server responds with the current view ID, which is subsequently used until a new view ID is returned by the server to replace the current view ID.

16.2.3. Response Header

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

Table 16.3. Response Header Fields

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

16.2.4. Topology Change Headers

16.2.4.1. Topology Change Headers

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

16.2.4.2. Topology Change Marker Values

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

Table 16.4. Topology Change Marker Field Values

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

16.2.4.3. Topology Change Headers for Topology-Aware Clients

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

Table 16.5. Topology Change Header Fields

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

Response Header Fields	Data Type/Size	Details
mX: Port	Unsigned Short. 2 bytes	Contains the port used by Hot Rod clients to communicate with the cluster member.

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

16.2.4.4. Topology Change Headers for Hash Distribution-Aware Clients

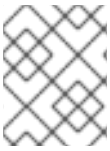
The response header sent to clients when a topology change is returned by the server includes the following elements:

Table 16.6. Topology Change Header Fields

Field	Data Type/Size	Details
Response Header with Topology Change Marker	variable	Refer to Response Header.
Topology ID	vInt	Topology ID.
Number Key Owners	Unsigned short. 2 bytes.	Contains the number of globally configured copies for each distributed key. Contains the value **0** if distribution is not configured on the cache.
Hash Function Version	1 byte	Contains a pointer to the hash function in use. Contains the value **0** if distribution is not configured on the cache.
Hash Space Size	vInt	Contains the modulus used by JBoss Data Grid for all module arithmetic related to hash code generation. Clients use this information to apply the correct hash calculations to the keys. Contains the value **0** if distribution is not configured on the cache.

Field	Data Type/Size	Details
Number servers in topology	vInt	Contains the number of [path]_ Hot Rod_ servers running in the cluster. This value can be a subset of the entire cluster if only some nodes are running [path]_ Hot Rod_ servers. This value also represents the number of host to port pairings included in the header.
Number Virtual Nodes Owners	vInt	Contains the number of configured virtual nodes. Contains the value **0** if no virtual nodes are configured or if distribution is not configured on the cache.
mX: Host/IP Length	vInt	Contains the length of the hostname or [path]_ IP_ address of an individual cluster member. Variable length allows this element to include hostnames, [path]_ IPv4_ and [path]_ IPv6_ addresses.
mX: Host/IP Address	string	Contains the hostname or [path]_ IP_ address of an individual cluster member. The [path]_ Hot Rod_ client uses this information to access the individual cluster member.
mX: Port	Unsigned short. 2 bytes.	Contains the port used by [path]_ Hot Rod_ clients to communicate with the cluster member.
Hash Function Version	1 byte	0x03
Number of Segments in Topology	vInt	Total number of segments in the topology.
Number of Owners in Segment	1 byte	This can be either 0, 1, or 2 owners.
First Wwner's Index	vInt	Given the list of all nodes, the position of this owner in this list. This is only present if number of owners for this segment is 1 or 2.

Field	Data Type/Size	Details
Second Owner's Index	vInt	Given the list of all nodes, the position of this owner in this list. This is only present if number of owners for this segment is 2.



NOTE

Even though it is possible to have more than 2 owners per segment, the Hot Rod protocol limits the number of owners to send for efficiency reasons.

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

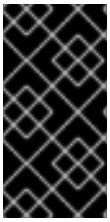
16.3. HOT ROD OPERATIONS

16.3.1. Hot Rod Operations

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

- Authenticate
- AuthMechList
- BulkGet
- BulkKeysGet
- Clear
- ContainsKey
- Exec
- Get
- GetAll
- GetWithMetadata
- GetWithVersion
- IterationEnd
- IterationNext
- IterationStart
- Ping

- Put
- PutAll
- PutIfAbsent
- Query
- Remove
- RemoveIfUnmodified
- Replace
- ReplaceIfUnmodified
- Stats
- Size



IMPORTANT

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

16.3.2. Hot Rod Authenticate Operation

The purpose of this operation is to authenticate a client against a server using SASL. The authentication process, depending on the chosen mech, might be a multi-step operation. Once complete the connection becomes authenticated.

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

Table 16.7. Authenticate Operation Request Format

Field	Data Type	Details
Header	variable	Request header.
Mech	String	String containing the name of the mech chosen by the client for authentication. Empty on the successive invocations.
Response length	vInt	Length of the SASL client response.
Response data	byte array	The SASL client response.

The response header for this operation contains the following:

Table 16.8. Authenticate Operation Response Format

Field	Data Type	Details
Header	variable	Response header.
Completed	byte	0 if further processing is needed, or 1 if authentication is complete.
Challenge length	vInt	Length of the SASL server challenge.
Challenge data	byte array	The SASL server challenge.

16.3.3. Hot Rod AuthMechList Operation

The purpose of this operation is to obtain the list of valid SASL authentication mechs supported by the server. The client will then need to issue an **Authenticate** request with the preferred mech.

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

Table 16.9. AuthMechList Operation Request Format

Field	Data Type	Details
Header	Variable	Request header

The response header for this operation contains the following:

Table 16.10. AuthMechList Operation Response Format

Field	Data Type	Details
Header	Variable	Response header
Mech count	vInt	The number of mechs.
Mech	String	String containing the name of the SASL mech in its IANA-registered form (e.g. GSSAPI, CRAM-MD5, etc)

The **Mech** value recurs for each supported mech.

16.3.4. Hot Rod BulkGet Operation

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

Table 16.11. BulkGet Operation Request Format

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

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

Table 16.12. BulkGet Operation Response Format

Field	Data Type	Details
Header	variable	Response Header
More	vInt	Represents if more entries must be read from the stream. While **More** is set to **1**, additional entries follow until the value of More is set to **0**, which indicates the end of the stream.
Key Length	vInt	Contains the length of the key.
Key	byte array	Contains the key value.
Value Length	vInt	Contains the length of the value.
Value	byte array	Contains the value.

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

16.3.5. Hot Rod BulkKeysGet Operation

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

Table 16.13. BulkKeysGet Operation Request Format

Field	Data Type	Details
Header	variable	Request header.

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

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

Table 16.14. BulkKeysGet Operation Response Format

Field	Data Type	Details
Header	variable	Response header.
Response Status	1 byte	**0x00** = success, data follows.

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

The values **Key Length** and **Key** recur for each key.

16.3.6. Hot Rod Clear Operation

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

Valid response statuses for this operation are as follows:

Table 16.15. Clear Operation Response

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

16.3.7. Hot Rod ContainsKey Operation

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

Table 16.16. ContainsKey Operation Request Format

Field	Data Type	Details
Header	-	-

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

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

Table 16.17. ContainsKey Operation Response Format

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

The response for this operation is empty.

16.3.8. Hot Rod Exec Operation

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

Table 16.18. Exec Operation Request Format

Field	Data Type	Details
Header	variable	Request header.
Script	String	Name of the script to execute.
Parameter Count	vInt	The number of parameters.
Parameter Name (per parameter)	String	The name of the parameter.
Parameter Length (per parameter)	vInt	The length of the parameter.
Parameter Value (per parameter)	byte array	The value of the parameter.

Field	Data Type	Details

The response header for this operation contains the following:

Table 16.19. Exec Operation Response Format

Field	Data Type	Details
Header	variable	Response header.
Response status	1 byte	0x00 if the execution completed successfully. 0x85 if the server resulted in an error.
Value Length	vInt	If success, length of return value.
Value	byte array	If success, the result of the execution.

16.3.9. Hot Rod Get Operation

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

Table 16.20. Get Operation Request Format

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

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

Table 16.21. Get Operation Response Format

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

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

Table 16.22. Get Operation Response Format

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

16.3.10. Hot Rod GetAll Operation

Bulk operation to get all entries that map to a given set of keys.

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

Table 16.23. GetAll Operation Request Format

Field	Data Type	Details
Header	variable	Request header
Key Count	vInt	How many keys to find entities for.
Key Length	vInt	Length of key.
Key	byte array	Retrieved key.

The **Key Length** and **Key** values recur for each key.

The response header for this operation contains the following:

Table 16.24. GetAll Operation Response Format

Field	Data Type	Details
Header	variable	Response header

Field	Data Type	Details
Entry count	vInt	How many entries are being returned.
Key Length	vInt	Length of key.
Key	byte array	Retrieved key.
Value Length	vInt	Length of value.
Value	byte array	Retrieved value.

The **Key Length**, **Key**, **Value Length**, and **Value** entries recur per key and value.

16.3.11. Hot Rod GetWithMetadata Operation

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

Table 16.25. GetWithMetadata Operation Request Format

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

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

Table 16.26. GetWithMetadata Operation Response Format

Field	Data Type	Details
Header	variable	Response header.

Field	Data Type	Details
Response status	1 byte	**0x00** = success, if key retrieved. **0x02** = if key does not exist.
Flag	1 byte	A flag indicating whether the response contains expiration information. The value of the flag is obtained as a bitwise OR operation between **INFINITE_LIFESPAN (0x01)** and **INFINITE_MAXIDLE (0x02)**.
Created	Long	(optional) a Long representing the timestamp when the entry was created on the server. This value is returned only if the flag's **INFINITE_LIFESPAN** bit is not set.
Lifespan	vInt	(optional) a vInt representing the lifespan of the entry in seconds. This value is returned only if the flag's **INFINITE_LIFESPAN** bit is not set.
LastUsed	Long	(optional) a Long representing the timestamp when the entry was last accessed on the server. This value is returned only if the flag's **INFINITE_MAXIDLE** bit is not set.
MaxIdle	vInt	(optional) a vInt representing the maxIdle of the entry in seconds. This value is returned only if the flag's **INFINITE_MAXIDLE** bit is not set.
Entry Version	8 bytes	Unique value of an existing entry modification. The protocol does not mandate that entry_version values are sequential, however they need to be unique per update at the key level.
Value Length	vInt	If success, length of value.
Value	byte array	If success, the requested value.

16.3.12. Hot Rod GetWithVersion Operation

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

Table 16.27. GetWithVersion Operation Request Format

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

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

Table 16.28. GetWithVersion Operation Response Format

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

The format of the **GetWithVersion** operation's response when the key is found is as follows:

Table 16.29. GetWithVersion Operation Response Format

Field	Data Type	Details
Header	variable	Response header
Entry Version	8 bytes	Unique value of an existing entry's modification. The protocol does not mandate that entry_version values are sequential. They just need to be unique per update at the key level.

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

16.3.13. Hot Rod IterationEnd Operation

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

Table 16.30. IterationEnd Operation Request Format

Field	Data Type	Details
iterationId	String	The unique id of the iteration.

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

Table 16.31. IterationEnd Operation Response Format

Response Status	Details
0x00	Successful operation.
0x05	Error for non existent iterationId.

16.3.14. Hot Rod IterationNext Operation

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

Table 16.32. IterationNext Operation Request Format

Field	Data Type	Details
IterationId	String	The unique id of the iteration.

The response header for this operation contains the following:

Table 16.33. IterationNext Operation Response Format

Field	Data Type	Details
Finished segments size	vInt	Size of the bitset representing segments that were finished iterating.

Field	Data Type	Details
Finished segments	byte array	Bitset encoding of the segments that were finished iterating.
Entry count	vInt	How many entries are being returned.
Number of value projections	vInt	Number of projections for the values.
Metadata	1 byte	If set, entry has metadata associated.
Expiration	1 byte	A flag indicating whether the response contains expiration information. The value of the flag is obtained as a bitwise OR operation between **INFINITE_LIFESPAN (0x01)** and **INFINITE_MAXIDLE (0x02)**. Only present if the metadata flag above is set.
Created	Long	(optional) a Long representing the timestamp when the entry was created on the server. This value is returned only if the flag's **INFINITE_LIFESPAN** bit is not set.
Lifespan	vInt	(optional) a vInt representing the lifespan of the entry in seconds. This value is returned only if the flag's **INFINITE_LIFESPAN** bit is not set.
LastUsed	Long	(optional) a Long representing the timestamp when the entry was last accessed on the server. This value is returned only if the flag's **INFINITE_MAXIDLE** bit is not set.
MaxIdle	vInt	(optional) a vInt representing the maxIdle of the entry in seconds. This value is returned only if the flag's **INFINITE_MAXIDLE** bit is not set.

Field	Data Type	Details
Entry Version	8 bytes	Unique value of an existing entry's modification. Only present if Metadata flag is set.
Key Length	vInt	Length of key.
Key	byte array	Retrieved key.
Value Length	vInt	Length of value.
Value	byte array	Retrieved value.

For each entry the **Metadata**, **Expiration**, **Created**, **Lifespan**, **LastUsed**, **MaxIdle**, **Entry Version**, **Key Length**, **Key**, **Value Length**, and **Value** fields recur.

16.3.15. Hot Rod IterationStart Operation

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

Table 16.34. IterationStart Operation Request Format

Field	Data Type	Details
Segments size	signed vInt	Size of the bitset encoding of the segments ids to iterate on. The size is the maximum segment id rounded to nearest multiple of 8. A value -1 indicates no segment filtering is to be done

Field	Data Type	Details
Segments	byte array	(Optional) Contains the segments ids bitset encoded, where each bit with value 1 represents a segment in the set. Byte order is little-endian. Example: segments [1,3,12,13] would result in the following encoding: 00001010 00110000 size: 16 bits first byte: represents segments from 0 to 7, from which 1 and 3 are set second byte: represents segments from 8 to 15, from which 12 and 13 are set More details in the **java.util.BitSet** implementation. Segments will be sent if the previous field is not negative
FilterConverter size	signed vInt	The size of the String representing a **KeyValueFilterConverter** factory name deployed on the server, or -1 if no filter will be used.
FilterConverter	UTF-8 byte array	(Optional) **KeyValueFilterConverter** factory name deployed on the server. Present if previous field is not negative.
Parameters size	byte	The number of parameters of the filter. Only present when **FilterConverter** is provided.
Parameters	byte[][]	An array of parameters. Each parameter is a byte array. Only present if **Parameters** size is greater than 0.
BatchSize	vInt	Number of entries to transfers from the server at one go.

Field	Data Type	Details
Metadata	1 byte	1 if metadata is to be returned for each entry, 0 otherwise.

The response header for this operation contains the following:

Table 16.35. IterationEnd Operation Response Format

Field	Data Type	Details
IterationId	String	The unique id of the iteration.

16.3.16. Hot Rod Ping Operation

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

Valid response statuses for this operation are as follows:

Table 16.36. Ping Operation Response

Response Status	Details
0x00	Successful ping without any errors.

16.3.17. Hot Rod Put Operation

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

.

Field	Data Type	Details
Header	variable	Request header.
Key Length	-	Contains the length of the key.
Key	Byte array	Contains the key value.

Field	Data Type	Details
TimeUnits	Byte	Time units of lifespan (first 4 bits) and maxIdle (last 4 bits). Special units **DEFAULT** and **INFINITE** can be used for default server expiration and no expiration respectively. Possible values: 0x00 = SECONDS 0x01 = MILLISECONDS 0x02 = NANOSECONDS 0x03 = MICROSECONDS 0x04 = MINUTES 0x05 = HOURS 0x06 = DAYS 0x07 = DEFAULT 0x08 = INFINITE
Lifespan	vInt	Duration which the entry is allowed to life. Only sent when time unit is not **DEFAULT** or **INFINITE**
Max Idle	vInt	Duration that each entry can be idle before it's evicted from the cache. Only sent when time unit is not **DEFAULT** or **INFINITE**.
Value Length	vInt	Contains the length of the value.
Value	Byte array	The requested value.

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

.

Response Status	Details
0x00	The value was successfully stored.
0x03	The value was successfully stored, and the previous value follows.

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

16.3.18. Hot Rod PutAll Operation

Bulk operation to put all key value entries into the cache at the same time.

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

Table 16.37. PutAll Operation Request Format

Field	Data Type	Details
Header	variable	Request header.
TimeUnits	Byte	Time units of lifespan (first 4 bits) and maxIdle (last 4 bits). Special units **DEFAULT** and **INFINITE** can be used for default server expiration and no expiration respectively. Possible values: 0x00 = SECONDS 0x01 = MILLISECONDS 0x02 = NANOSECONDS 0x03 = MICROSECONDS 0x04 = MINUTES 0x05 = HOURS 0x06 = DAYS 0x07 = DEFAULT 0x08 = INFINITE
Lifespan	vInt	Duration which the entry is allowed to life. Only sent when time unit is not **DEFAULT** or **INFINITE**
Max Idle	vInt	Duration that each entry can be idle before it's evicted from the cache. Only sent when time unit is not **DEFAULT** or **INFINITE**.
Entry count	vInt	How many entries are being inserted.
Key Length	vInt	Length of key.
Key	byte array	Retrieved key.
Value Length	vInt	Length of value.
Value	byte array	Retrieved value.

The **Key Length**, **Key**, **Value Length**, and **Value** fields repeat for each entry that will be placed.

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

Table 16.38. PutAll Operation Response Format

Response Status	Details
0x00	Successful operation, indicating all keys were successfully put.

16.3.19. Hot Rod PutIfAbsent Operation

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

Table 16.39. PutIfAbsent Operation Request Fields

Field	Data Type	Details
Header	variable	Request header.
Key Length	vInt	Contains the length of the key.
Key	Byte array	Contains the key value.
TimeUnits	Byte	Time units of lifespan (first 4 bits) and maxIdle (last 4 bits). Special units **DEFAULT** and **INFINITE** can be used for default server expiration and no expiration respectively. Possible values: 0x00 = SECONDS 0x01 = MILLISECONDS 0x02 = NANOSECONDS 0x03 = MICROSECONDS 0x04 = MINUTES 0x05 = HOURS 0x06 = DAYS 0x07 = DEFAULT 0x08 = INFINITE
Lifespan	vInt	Duration which the entry is allowed to life. Only sent when time unit is not **DEFAULT** or **INFINITE**
Max Idle	vInt	Duration that each entry can be idle before it's evicted from the cache. Only sent when time unit is not **DEFAULT** or **INFINITE**.
Value Length	vInt	Contains the length of the value.

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

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

.

Response Status	Details
0x00	The value was successfully stored.
0x01	The key was present, therefore the value was not stored. The current value of the key is returned.
0x04	The operation failed because the key was present and its value follows in the response.

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

16.3.20. Hot Rod Query Operation

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

Table 16.40. Query Operation Request Fields

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

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

Table 16.41. Query Operation Response

Response Status	Data	Details
Header	variable	Response header.

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

The Hot Rod **Query** operation request and response types are defined in the **org/infinispan/query/remote/client/query.proto** resource filed, found inside **infinispan-remote-query-client.jar**.

16.3.21. Hot Rod Remove Operation

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

Table 16.42. Remove Operation Request Format

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

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

Table 16.43. Remove Operation Response Format

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

Response Status	Details
0x03	The key was removed, and the previous or removed value follows in the response.

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

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

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

16.3.22. Hot Rod RemoveIfUnmodified Operation

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

Table 16.44. RemoveIfUnmodified Operation Request Fields

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

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

Table 16.45. RemoveIfUnmodified Operation Response

Response Status	Details
0x00	The entry was replaced or removed.
0x01	The entry replace or remove was unsuccessful because the key was modified.
0x02	The key does not exist.
0x03	The key was removed, and the previous or replaced value follows in the response.

Response Status	Details
0x04	The entry remove was unsuccessful because the key was modified, and the modified value follows in the response.

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

16.3.23. Hot Rod Replace Operation

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

Table 16.46. Replace Operation Request Fields

Field	Data Type	Details
Header	variable	Request header.
Key Length	vInt	Contains the length of the key.
Key	Byte array	Contains the key value.
TimeUnits	Byte	Time units of lifespan (first 4 bits) and maxIdle (last 4 bits). Special units **DEFAULT** and **INFINITE** can be used for default server expiration and no expiration respectively. Possible values: 0x00 = SECONDS 0x01 = MILLISECONDS 0x02 = NANOSECONDS 0x03 = MICROSECONDS 0x04 = MINUTES 0x05 = HOURS 0x06 = DAYS 0x07 = DEFAULT 0x08 = INFINITE
Lifespan	vInt	Duration which the entry is allowed to life. Only sent when time unit is not **DEFAULT** or **INFINITE**

Field	Data Type	Details
Max Idle	vInt	Duration that each entry can be idle before it's evicted from the cache. Only sent when time unit is not **DEFAULT** or **INFINITE**.
Value Length	vInt	Contains the length of the value.
Value	Byte array	Contains the requested value.

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

Table 16.47. Replace Operation Response

Response Status	Details
0x00	The value was successfully stored.
0x01	The value was not stored because the key does not exist.
0x03	The value was successfully replaced, and the previous or replaced value follows in the response.

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

16.3.24. Hot Rod ReplaceIfUnmodified Operation

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

Table 16.48. ReplaceIfUnmodified Operation Request Format

Field	Data Type	Details
Header	variable	Request header.
Key Length	vInt	Length of key. Note that the size of a vint can be up to 5 bytes which in theory can produce bigger numbers than **Integer.MAX_VALUE**. However, Java cannot create a single array that's bigger than **Integer.MAX_VALUE**, hence the protocol is limiting vint array lengths to **Integer.MAX_VALUE**.

Field	Data Type	Details
Key	byte array	Byte array containing the key whose value is being requested.
TimeUnits	Byte	Time units of lifespan (first 4 bits) and maxIdle (last 4 bits). Special units **DEFAULT** and **INFINITE** can be used for default server expiration and no expiration respectively. Possible values: 0x00 = SECONDS 0x01 = MILLISECONDS 0x02 = NANOSECONDS 0x03 = MICROSECONDS 0x04 = MINUTES 0x05 = HOURS 0x06 = DAYS 0x07 = DEFAULT 0x08 = INFINITE
Lifespan	vInt	Duration which the entry is allowed to life. Only sent when time unit is not **DEFAULT** or **INFINITE**
Max Idle	vInt	Duration that each entry can be idle before it's evicted from the cache. Only sent when time unit is not **DEFAULT** or **INFINITE**.
Entry Version	8 bytes	Use the value returned by **GetWithVersion** operation.
Value Length	vInt	Length of value.
Value	byte array	Value to be stored.

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

Table 16.49. ReplaceIfUnmodified Operation Response Status

Response Status	Details
0x00	The value was successfully stored.
0x01	Replace did not happen because key had been modified.
0x02	Replace did not happen because key does not exist.
0x03	The key was replaced, and the previous or replaced value follows in the response.
0x04	The entry replace was unsuccessful because the key was modified, and the modified value follows in the response.

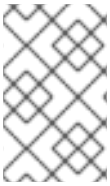
The following are the valid response values returned from this operation:

Table 16.50. ReplaceIfUnmodified Operation Response Format

Field	Data Type	Details
Header	variable	Response header.
Previous value length	vInt	If force return previous value flag was sent in the request, the length of the previous value will be returned. If the key does not exist, value length would be 0. If no flag was sent, no value length would be present.
Previous value	byte array	If force return previous value flag was sent in the request and the key was replaced, previous value.

16.3.25. Hot Rod ReplaceWithVersion Operation

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



NOTE

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

Table 16.51. ReplaceWithVersion Operation Request Fields

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

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

Table 16.52. ReplaceWithVersion Operation Response

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

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

16.3.26. Hot Rod Stats Operation

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

The following are supported statistics for this operation:

Table 16.53. Stats Operation Request Fields

Name	Details
timeSinceStart	Contains the number of seconds since Hot Rod started.
currentNumberOfEntries	Contains the number of entries that currently exist in the Hot Rod server.
totalNumberOfEntries	Contains the total number of entries stored in the Hot Rod server.
stores	Contains the number of put operations attempted.
retrievals	Contains the number of get operations attempted.
hits	Contains the number of get hits.
misses	Contains the number of get misses.
removeHits	Contains the number of remove hits.
removeMisses	Contains the number of removal misses.
globalCurrentNumberOfEntries	Number of entries currently across the Hot Rod cluster.
globalStores	Total number of put operations across the Hot Rod cluster.
globalRetrievals	Total number of get operations across the Hot Rod cluster.
globalHits	Total number of get hits across the Hot Rod cluster.
globalMisses	Total number of get misses across the Hot Rod cluster.
globalRemoveHits	Total number of removal hits across the Hot Rod cluster.

Name	Details
globalRemoveMisses	Total number of removal misses across the Hot Rod cluster.

NOTE

Any of the statistics beginning with **global** are not available if Hot Rod is running in local mode.

The response header for this operation contains the following:

Table 16.54. Stats Operation Response

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

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

16.3.27. Hot Rod Size Operation

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

Table 16.55. Size Operation Request Format

Field	Data Type	Details
Header	variable	Request header

The response header for this operation contains the following:

Table 16.56. Size Operation Response Format

Field	Data Type	Details
Header	variable	Response header.
Size	vInt	Size of the remote cache, which is calculated globally in the clustered set ups, and if present, takes cache store contents into account as well.

16.4. HOT ROD OPERATION VALUES

16.4.1. Hot Rod Operation Values

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

Table 16.57. Opcode Request and Response Header Values

Operation	Request Operation Code	Response Operation Code
put	0x01	0x02
get	0x03	0x04
putIfAbsent	0x05	0x06
replace	0x07	0x08
replaceIfUnmodified	0x09	0x0A
remove	0x0B	0x0C
removeIfUnmodified	0x0D	0x0E
containsKey	0x0F	0x10
clear	0x13	0x14
stats	0x15	0x16
ping	0x17	0x18
bulkGet	0x19	0x1A
getWithMetadata	0x1B	0x1C

Operation	Request Operation Code	Response Operation Code
bulkKeysGet	0x1D	0x1E
query	0x1F	0x20
authMechList	0x21	0x22
auth	0x23	0x24
addClientListener	0x25	0x26
removeClientListener	0x27	0x28
size	0x29	0x2A
exec	0x2B	0x2C
putAll	0x2D	0x2E
getAll	0x2F	0x30
iterationStart	0x31	0x32
iterationNext	0x33	0x34
iterationEnd	0x35	0x36

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

16.4.2. Magic Values

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

Table 16.58. Magic Field Values

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

16.4.3. Status Values

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

Table 16.59. Status Values

Value	Details
0x00	No error.
0x01	Not put/removed/replaced.
0x02	Key does not exist.
0x06	Success status and compatibility mode is enabled.
0x07	Success status and return previous value, with compatibility mode is enabled.
0x08	Not executed and return previous value, with compatibility mode is enabled.
0x81	Invalid Magic value or Message ID.
0x82	Unknown command.
0x83	Unknown version.
0x84	Request parsing error.
0x85	Server error.
0x86	Command timed out.

16.4.4. Client Intelligence Values

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

Table 16.60. Client Intelligence Field Values

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

16.4.5. Flag Values

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

Table 16.61. Flag Field Values

Value	Details
0x0001	ForceReturnPreviousValue

16.4.6. Hot Rod Error Handling

Table 16.62. Hot Rod Error Handling using Response Header Fields

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

16.5. HOT ROD REMOTE EVENTS

16.5.1. Hot Rod Remote Events

Clients may register Remote Event Listeners, allowing them to receive updates on events happening in the server. As soon as a client listener has been added events are generated and sent, allowing the client to receive all events that have occurred after adding the listener.

16.5.2. Hot Rod Add Client Listener for Remote Events

Adding client listeners for remote events uses the following request format:

Table 16.63. Add Client Listener Operation Request Format

Field	Data Type	Details
Header	variable	Request Header.

Field	Data Type	Details
Listener ID	byte array	Listener identifier.
Include state	byte	When this byte is set to 1, cached state is sent back to remote clients when either adding a cache listener for the first time, or when the node where a remote listener is registered changes in a clustered environment. When enabled, state is sent back as cache entry created events to the clients. If set to 0, no state is sent back to the client when adding a listener, nor it gets state when the node where the listener is registered changes.
Key/value filter factory name	String	Optional name of the key/value filter factory to be used with this listener. The factory is used to create key/value filter instances which allow events to be filtered directly in the Hot Rod server, avoiding sending events that the client is not interested in. If no factory is to be used, the length of the string is 0.
Key/value filter factory parameter count	byte	The key/value filter factory, when creating a filter instance, can take an arbitrary number of parameters, enabling the factory to be used to create different filter instances dynamically. This count field indicates how many parameters will be passed to the factory. If no factory name was provided, this field is not present in the request.
Key/value filter factory parameter (per parameter)	byte array	Key/value filter factory parameter.

Field	Data Type	Details
Converter factory name	String	Optional name of the converter factory to be used with this listener. The factory is used to transform the contents of the events sent to clients. By default, when no converter is in use, events are well defined, according to the type of event generated. However, there might be situations where users want to add extra information to the event, or they want to reduce the size of the events. In these cases, a converter can be used to transform the event contents. The given converter factory name produces converter instances to do this job. If no factory is to be used, the length of the string is 0.
Converter factory parameter count	byte	The converter factory, when creating a converter instance, can take an arbitrary number of parameters, enabling the factory to be used to create different converter instances dynamically. This count field indicates how many parameters will be passed to the factory. If no factory name was provided, this field is not present in the request.
Converter factory parameter (per parameter)	byte array	Converter factory parameter.
Use raw data	byte	If filter/converter parameters should be raw binary, then 1, otherwise 0.

The format of the operation's response is as follows:

Table 16.64. Add Client Listener Response Format

Field	Data Type	Details
Header	Variable	Response Header.

16.5.3. Hot Rod Remote Client Listener for Remote Events

Removing a previously added client listener uses the following request format:

Table 16.65. Remove Client Listener Operation Request Format

Field	Data Type	Details
Header	variable	Request Header.
Listener ID	byte array	Listener Identifier

The format of the operation's response is as follows:

Table 16.66. Add Client Listener Response Format

Field	Data Type	Details
Header	Variable	Response Header.

16.5.4. Hot Rod Event Header

Each remote event uses a header that adheres to the following format:

Table 16.67. Remote Event Header

Field Name	Size	Value
Magic	1 byte	0xA1 = response
Message ID	vLong	ID of event
Opcode	1 byte	A code responding to the Event type: 0x60 = cache entry created event 0x61 = cache entry modified event 0x62 = cache entry removed event 0x50 = error
Status	1 byte	Status of the response, with the following possible values: 0x00 = No error

Field Name	Size	Value
Topology Change Marker	1 byte	Since events are not associated with a particular incoming topology ID to be able to decide whether a new topology is required to be sent or not, new topologies will never be sent with events. Hence, this marker will always have 0 value for events.

16.5.5. Hot Rod Cache Entry Created Event

The **CacheEntryCreated** event includes the following:

Table 16.68. Cache Entry Created Event

Field Name	Size	Value
Header	variable	Event header with **0x60** operation code.
Listener ID	byte array	Listener for which this event is directed
Custom Marker	byte	Custom event marker. For created events, this is 0.
Command Retried	byte	Marker for events that are result of retried commands. If command is retried, it returns 1, otherwise 0.
Key	byte array	Created key.
Version	long	Version of the created entry. This version information can be used to make conditional operations on this cache entry.

16.5.6. Hot Rod Cache Entry Modified Event

The **CacheEntryModified** event includes the following:

Table 16.69. Cache Entry Modified Event

Field Name	Size	Value
Header	variable	Event header with **0x61** operation code.

Field Name	Size	Value
Listener ID	byte array	Listener for which this event is directed
Custom Marker	byte	Custom event marker. For created events, this is 0.
Command Retried	byte	Marker for events that are result of retried commands. If command is retried, it returns 1, otherwise 0.
Key	byte array	Modified key.
Version	long	Version of the modified entry. This version information can be used to make conditional operations on this cache entry.

16.5.7. Hot Rod Cache Entry Removed Event

The **CacheEntryRemoved** event includes the following:

Table 16.70. Cache Entry Removed Event

Field Name	Size	Value
Header	variable	Event header with **0x62** operation code.
Listener ID	byte array	Listener for which this event is directed
Custom Marker	byte	Custom event marker. For created events, this is 0.
Command Retried	byte	Marker for events that are result of retried commands. If command is retried, it returns 1, otherwise 0.
Key	byte array	Removed key.

16.5.8. Hot Rod Custom Event

The **Custom** event includes the following:

Table 16.71. Custom Event

Field Name	Size	Value
Header	variable	Event header with event specific operation code
Listener ID	byte array	Listener for which this event is directed
Custom Marker	byte	Custom event marker. For custom events whose event data needs to be unmarshalled before returning to user the value is 1. For custom events that need to return the event data as-is to the user, the value is 2.
Event Data	byte array	Custom event data. If the custom marker is 1, the bytes represent the marshalled version of the instance returned by the converter. If custom marker is 2, it represents the byte array, as returned by the converter.

16.6. PUT REQUEST EXAMPLE

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

Table 16.72. Put Request Example

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

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

Table 16.73. Example Request Field Names and Values

Field Name	Byte	Value
Magic	0	0xA0
Version	2	0x41
Cache Name Length	4	0x07
Flag	12	0x00
Topology ID	14	0x00
Transaction ID	16	0x00
Key	18-22	'Hello'
Max Idle	24	0x00
Value	26-30	'World'
Message ID	1	0x09
Opcode	3	0x01
Cache Name	5-11	'MyCache'
Client Intelligence	13	0x03
Transaction Type	15	0x00
Key Field Length	17	0x05
Lifespan	23	0x00
Value Field Length	25	0x05

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

Table 16.74. Coded Response for the Sample Put Request

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

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

Table 16.75. Example Response Field Names and Values

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

16.7. HOT ROD JAVA CLIENT

16.7.1. Hot Rod Java Client

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

16.7.2. Hot Rod Java Client Download

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

Procedure: Download Hot Rod Java Client

1. Log into the Customer Portal at https://access.redhat.com.
2. Click the **Downloads** button near the top of the page.
3. In the **Product Downloads** page, click **Red Hat JBoss Data Grid**.
4. Select the appropriate JBoss Data Grid version from the **Version:** drop down menu.
5. Locate the **Red Hat JBoss Data Grid 7.2 Hot Rod Java Client** entry and click the corresponding **Download** link.

16.7.3. Hot Rod Java Client Configuration

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

Client Instance Creation

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
= new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.tcpNoDelay(true)
  .connectionPool()
     .numTestsPerEvictionRun(3)
     .testOnBorrow(false)
     .testOnReturn(false)
```

```
      .testWhileIdle(true)
   .addServer()
      .host("localhost")
      .port(11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

Configuring the Hot Rod Java client using a properties file

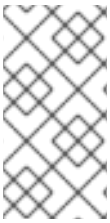
To configure the Hot Rod Java client, edit the *hotrod-client.properties* file on the classpath.

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

Configuration

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

infinispan.client.hotrod.server_list = 127.0.0.1:11222

infinispan.client.hotrod.marshaller = org.infinispan.commons.marshall.jboss.GenericJBossMarshaller

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

infinispan.client.hotrod.default_executor_factory.pool_size = 1

infinispan.client.hotrod.default_executor_factory.queue_size = 10000

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

infinispan.client.hotrod.tcp_no_delay = true

infinispan.client.hotrod.ping_on_startup = true

infinispan.client.hotrod.request_balancing_strategy =
org.infinispan.client.hotrod.impl.transport.tcp.RoundRobinBalancingStrategy

infinispan.client.hotrod.key_size_estimate = 64

infinispan.client.hotrod.value_size_estimate = 512

infinispan.client.hotrod.force_return_values = false

infinispan.client.hotrod.tcp_keep_alive = true

## below is connection pooling config

maxActive=-1

maxTotal = -1

maxIdle = -1

whenExhaustedAction = 1
```

```
timeBetweenEvictionRunsMillis=120000

minEvictableIdleTimeMillis=300000

testWhileIdle = true

minIdle = 1
```



NOTE

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

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

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

16.7.4. Hot Rod Java Client Basic API

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

Basic API

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

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

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

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

Use this method with caution if there are a large number of keys.

```
Set keys = remoteCache.keySet();
```

16.7.5. Hot Rod Java Client Versioned API

To ensure data consistency, Hot Rod stores a version number that uniquely identifies each modification. Using **getVersioned**, clients can retrieve the value associated with the key as well as the current version.

When using the Hot Rod Java client, a **RemoteCacheManager** provides instances of the **RemoteCache** interface that accesses the named or default cache on the remote cluster. This extends the **Cache** interface to which it adds new methods, including the versioned API.

Using Versioned Methods

```
// To use the versioned API, remote classes are specifically needed
RemoteCacheManager remoteCacheManager = new RemoteCacheManager();
RemoteCache<String, String> remoteCache = remoteCacheManager.getCache();
remoteCache.put("car", "ferrari");
VersionedValue valueBinary = remoteCache.getWithMetadata("car");
// removal only takes place only if the version has not been changed
// in between. (a new version is associated with 'car' key on each change)
assert remoteCache.removeWithVersion("car", valueBinary.getVersion());
assert !remoteCache.containsKey("car");
```

Using Replace

```
remoteCache.put("car", "ferrari");
VersionedValue valueBinary = remoteCache.getWithMetadata("car");
assert remoteCache.replaceWithVersion("car", "lamborghini", valueBinary.getVersion());
```

16.7.6. Cluster-Wide Dynamic Cache Creation with Hot Rod Java Client

If a cache needs to be created dynamically from a client, use the **createCache()** method as follows:

```
BasicCache<String, String> cache =
remoteCacheManager.administration().createCache("newCacheName", "newTemplate");
```

While a cache created this way will be available on all nodes in the cluster, it will also be ephemeral: shutting down the entire cluster and restarting it will not automatically recreate the caches. To make the caches persistent, use the **PERMANENT** flag as follows:

```
BasicCache<String, String> cache =
remoteCacheManager.administration().withFlags(AdminFlag.PERMANENT).createCache("newCache
Name", "newTemplate");
```

In order for the above to work, global state must be enabled and a suitable configuration storage selected. The available configuration stores are:

- **VOLATILE**: as the name implies, this configuration storage does not support **PERMANENT** caches.
- **OVERLAY**: this stores configurations in the global shared state persistent path in a file named *caches.xml*.

- **MANAGED**: this is only supported in server deployments, and will store **PERMANENT** caches in the server model.
- **CUSTOM**: a custom configuration store.

16.8. HOT ROD C++ CLIENT

16.8.1. Hot Rod C++ Client

The Hot Rod C++ client enables C++ runtime applications to connect and interact with Red Hat JBoss Data Grid remote servers, and to read or write data to remote caches. The Hot Rod C++ client supports all three levels of client intelligence and is supported on the following platforms:

- Red Hat Enterprise Linux 6, 64-bit
- Red Hat Enterprise Linux 7, 64-bit

> The Hot Rod C++ client is available as a Technology Preview on 64-bit Windows with Visual Studio 2015.

16.8.2. Hot Rod C++ Client Formats

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

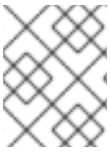
- Static library
- Shared/Dynamic library

Static Library

The static library is statically linked to an application. This increases the size of the final executable. The application is self-contained and it does not need to ship a separate library.

Shared/Dynamic Library

Shared/Dynamic libraries are dynamically linked to an application at runtime. The library is stored in a separate file and can be upgraded separately from the application, without recompiling the application.



NOTE

This can only happen if the library's major version is equal to the one against which the application was linked at compile time, indicating that it is binary compatible.

16.8.3. Hot Rod C++ Client Prerequisites

The following table details requirements needed to use the Hot Rod C++ Client depending on the underlying OS:

Table 16.76. Hot Rod C++ Client Prerequisites by OS

Operating System	Hot Rod C++ Client Prerequisites
RHEL 6, 64-bit	C++ 03 compiler with support for shared_ptr TR1 (GCC 4.0+)

Operating System	Hot Rod C++ Client Prerequisites
RHEL 7, 64-bit	C++ 11 compiler (GCC 4.8.1)
Windows 7 x64	C 11 compiler (Visual Studio 2015, Microsoft Visual C 2013 Redistributable Package for the x64 platform)

16.8.4. Installing the Hot Rod C++ Client

16.8.4.1. Hot Rod C++ Client Download and Installation

The Hot Rod C++ client is distributed in two file types, based on the Operating System where the client will be used:

- RHEL servers install via an RPM distribution.
- Windows servers install via a zip distribution.

16.8.4.2. Hot Rod C++ Client RHEL Download and Installation

To install the client perform the following steps:

1. Ensure your Red Hat Enterprise Linux (RHEL) system is registered to your account using Red Hat Subscription Manager. For more information, refer to the Red Hat Subscription Management documentation.
2. Using Red Hat Subscription Manager, enable the appropriate repository based on your version of RHEL:

   Table 16.77. RHSM Repositories

RHEL Version	Repo Name
RHEL 6	jb-datagrid-7.2-for-rhel-6-server-rpms
RHEL 7	jb-datagrid-7.2-for-rhel-7-server-rpms

   For instance, to enable the RHEL 7 repo the following command would be used:

   ```
   subscription-manager repos --enable=jb-datagrid-7.2-for-rhel-7-server-rpms
   ```

   For RHEL 7 you also need to enable the **rhel-7-server-optional-rpms** repo which provides the required **protobuf-devel** and **protobuf-static** RPMs:

   ```
   subscription-manager repos --enable=rhel-7-server-optional-rpms
   ```

3. Once the appropriate repos have been added the C++ client RPM may be installed with:

```
yum install jdg-cpp-client
```

16.8.4.3. Hot Rod C++ Client Windows Download and Installation

The Hot Rod C++ Client for Windows is included in a separate zip file *jboss-datagrid-<version>-hotrod-cpp-WIN-x86_64.zip* under Red Hat JBoss Data Grid binaries on the Red Hat Customer Portal at https://access.redhat.com.

Once downloaded the C++ Client may be installed by extracing the zip file to the desired location on the system.

16.8.5. Utilizing the Protobuf Compiler with the Hot Rod C++ Client

16.8.5.1. Using the Protobuf Compiler in RHEL 7

The C++ Hot Rod client channel in RHEL 7 includes the Protobuf compiler. The following instructions detail using this compiler:

1. Ensure that the C++ channel has been added to the RHEL system, as outlined in Hot Rod C++ Client RHEL Download and Installation.:
2. Install the **protobuf** rpm:

```
yum install protobuf
```

3. Add the included protobuf libraries to the library path. These libraries are included in **/opt/lib64** by default:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/lib64
```

4. Compile the desired protobuf files into C++ header and source files:

```
/bin/protoc --cpp_out dllexport_decl=HR_PROTO_EXPORT:/path/to/output/ $FILE
```

NOTE

HR_PROTO_EXOPRT is a macro defined within the Hot Rod client code, and will be expanded when the files are subsequently compiled.

5. The resulting header and source files will be generated in the designated output directory, allowing them to be referenced and compiled as normal with the specific application code.

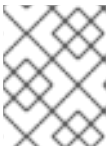
For additional information on Protobuf refer to Protobuf Encoding.

16.8.5.2. Using the Protobuf Compiler in Windows

The C++ Hot Rod client for Windows ships with the precompiled Hot Rod components along with the Protobuf compiler included. For many users the included components may be used without the need for additional compilation; however, should any .proto files require compiling the following instructions document this process:

1. Extract the *jboss-datagrid-<version>-hotrod-cpp-client-WIN-x86_64.zip* locally to the filesystem.
2. Open a command prompt and navigate to the newly extracted directory.
3. Compile the desired protobuf files into C++ header and source files:

```
bin\protoc --cpp_out dllexport_decl=HR_PROTO_EXPORT:path\to\output\ $FILE
```



NOTE

HR_PROTO_EXOPRT is a macro defined within the Hot Rod client code, and will be expanded when the files are subsequently compiled.

4. The resulting header and source files will be generated in the designated output directory, allowing them to be referenced and compiled as normal with the specific application code.

For additional information on Protobuf refer to Protobuf Encoding.

16.8.6. Hot Rod C++ Client Configuration

The Hot Rod C++ client interacts with a remote Hot Rod server using the RemoteCache API. To initiate communication with a particular Hot Rod server, configure RemoteCache and choose the specific cache on the Hot Rod server.

Use the ConfigurationBuilder API to configure:

- The initial set of servers to connect to.
- Connection pooling attributes.
- Connection/Socket timeouts and TCP nodelay.
- Hot Rod protocol version.

Sample C++ main executable file configuration

The following example shows how to use the ConfigurationBuilder to configure a **RemoteCacheManager** and how to obtain the default remote cache:

SimpleMain.cpp

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include <stdlib.h>
using namespace infinispan::hotrod;
int main(int argc, char** argv) {
    ConfigurationBuilder b;
    b.addServer().host("127.0.0.1").port(11222);
    RemoteCacheManager cm(builder.build());
    RemoteCache<std::string, std::string> cache = cm.getCache<std::string, std::string>();
    return 0;
}
```

16.8.7. Hot Rod C++ Client API

The RemoteCacheManager is a starting point to obtain a reference to a RemoteCache. The RemoteCache API can interact with a remote Hot Rod server and the specific cache on that server.

Using the RemoteCache reference obtained in the previous example, it is possible to put, get, replace and remove values in a remote cache. It is also possible to perform bulk operations, such as retrieving all of the keys, and clearing the cache.

When a RemoteCacheManager is stopped, all resources in use are released.

SimpleMain.cpp

```
RemoteCache<std::string, std::string> rc = cm.getCache<std::string, std::string>();
    std::string k1("key13");
    std::string v1("boron");
    // put
    rc.put(k1, v1);
    std::auto_ptr<std::string> rv(rc.get(k1));
    rc.putIfAbsent(k1, v1);
    std::auto_ptr<std::string> rv2(rc.get(k1));
    std::map<HR_SHARED_PTR<std::string>,HR_SHARED_PTR<std::string> > map = rc.getBulk(0);
    std::cout << "getBulk size" << map.size() << std::endl;
    ..
    .
    cm.stop();
```

16.8.8. Hot Rod C++ Client Asynchronous API

The Hot Rod C++ client offers asynchronous versions of many of the synchronous methods, allowing non-blocking methods for interacting with remote caches.

These methods follow the same naming convention as the synchronous methods, except that **Async** is appended to the end of each method. Asynchronous methods return a **std::future** containing the result of the operation. If a method were to return a **std::string**, instead it will return a **std::future < std::string* >**

A list of asynchronous methods are below:

- **clearAsync**
- **getAsync**
- **putAsync**
- **putAllAsync**
- **putIfAbsentAsync**
- **removeAsync**
- **removeWithVersionAsync**
- **replaceAsync**
- **replaceWithVersionAsync**

Hot Rod C++ Asynchronous API Example

The following example demonstrates using these methods:

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"

#include "infinispan/hotrod/JBasicMarshaller.h"
#include <iostream>
#include <thread>
#include <future>

using namespace infinispan::hotrod;

int main(int argc, char** argv) {
    ConfigurationBuilder builder;
    builder.addServer().host(argc > 1 ? argv[1] : "127.0.0.1").port(argc > 2 ? atoi(argv[2]) :
11222).protocolVersion(Configuration::PROTOCOL_VERSION_24);
    RemoteCacheManager cacheManager(builder.build(), false);
    auto *km = new BasicMarshaller<std::string>();
    auto *vm = new BasicMarshaller<std::string>();
    auto cache = cacheManager.getCache<std::string, std::string>(km,
&Marshaller<std::string>::destroy, vm, &Marshaller<std::string>::destroy );
    cacheManager.start();
    std::string ak1("asyncK1");
    std::string av1("asyncV1");
    std::string ak2("asyncK2");
    std::string av2("asyncV2");
    cache.clear();

    // Put ak1,av1 in async thread
    std::future<std::string*> future_put= cache.putAsync(ak1,av1);
    // Get the value in this thread
    std::string* arv1= cache.get(ak1);

    // Now wait for put completion
    future_put.wait();

    // All is synch now
    std::string* arv11= cache.get(ak1);
    if (!arv11 || arv11->compare(av1))
    {
        std::cout << "fail: expected " << av1 << "got " << (arv11 ? *arv11 : "null") << std::endl;
        return 1;
    }

    // Read ak1 again, but in async way and test that the result is the same
    std::future<std::string*> future_ga= cache.getAsync(ak1);
    std::string* arv2= future_ga.get();
    if (!arv2 || arv2->compare(av1))
    {
        std::cerr << "fail: expected " << av1 << " got " << (arv2 ? *arv2 : "null") << std::endl;
        return 1;
    }
```

```
    // Now user pass a simple lambda func that set a flag to true when the put completes
    bool flag=false;
    std::future<std::string*> future_put1= cache.putAsync(ak2,av2,0,0,[&] (std::string *v){flag=true;
return v;});
    // The put is not completed here so flag must be false
    if (flag)
    {
        std::cerr << "fail: expected false got true" << std::endl;
        return 1;
    }
    // Now wait for put completion
    future_put1.wait();
    // The user lambda must be executed so flag must be true
    if (!flag)
    {
        std::cerr << "fail: expected true got false" << std::endl;
        return 1;
    }

    // Same test for get
    flag=false;
    // Now user pass a simple lambda func that set a flag to true when the put completes
    std::future<std::string*> future_get1= cache.getAsync(ak2,[&] (std::string *v){flag=true; return v;});
    // The get is not completed here so flag must be false
    if (flag)
    {
        std::cerr << "fail: expected false got true" << std::endl;
        return 1;
    }
    // Now wait for get completion
    future_get1.wait();
    if (!flag)
    {
        std::cerr << "fail: expected true got false" << std::endl;
        return 1;
    }
    std::string* arv3= future_get1.get();
    if (!arv3 || arv3->compare(av2))
    {
        std::cerr << "fail: expected " << av2 << " got " << (arv3 ? *arv3 : "null") << std::endl;
        return 1;
    }
    cacheManager.stop();
}
```

16.8.9. Hot Rod C++ Client Remote Event Listeners

The Hot Rod C++ client supports remote cache listeners, and these may be added using the **add_listener** function on the **ClientCacheListener**.

IMPORTANT

Remote Event Listeners are a Technology Preview feature of the Hot Rod C++ client in Red Hat JBoss Data Grid 7.2.

This function takes a listener for each event type(**create**, **modify**, **remove**, **expire**, or **custom**). For more information on Remote Event Listeners refer to Remote Event Listeners (Hot Rod). An example of this is provided below:

```
ConfigurationBuilder builder;
   builder.balancingStrategyProducer(nullptr);
builder.addServer().host("127.0.0.1").port(11222);
builder.protocolVersion(Configuration::PROTOCOL_VERSION_24);
RemoteCacheManager cacheManager(builder.build(), false);
cacheManager.start();
JBasicMarshaller<int> *km = new JBasicMarshaller<int>();
JBasicMarshaller<std::string> *vm = new JBasicMarshaller<std::string>();
RemoteCache<int, std::string> cache = cacheManager.getCache<int, std::string>(km,
   &Marshaller<int>::destroy,
   vm,
   &Marshaller<std::string>::destroy);
cache.clear();
std::vector<std::vector<char> > filterFactoryParams;
std::vector<std::vector<char> > converterFactoryParams;
CacheClientListener<int, std::string> cl(cache);
int createdCount=0, modifiedCount=0, removedCount=0, expiredCount=0;

// We're using future and promise to have a basic listeners/main thread synch
int setFutureEventKey=0;
std::promise<void> promise;
std::function<void(ClientCacheEntryCreatedEvent<int>)> listenerCreated = [&createdCount,
&setFutureEventKey, &promise](ClientCacheEntryCreatedEvent<int> e) { createdCount++; if
(setFutureEventKey==e.getKey()) promise.set_value(); };
std::function<void(ClientCacheEntryModifiedEvent<int>)> listenerModified = [&modifiedCount,
&setFutureEventKey, &promise](ClientCacheEntryModifiedEvent <int> e) { modifiedCount++; if
(setFutureEventKey==e.getKey()) promise.set_value(); };
std::function<void(ClientCacheEntryRemovedEvent<int>)> listenerRemoved = [&removedCount,
&setFutureEventKey, &promise](ClientCacheEntryRemovedEvent <int> e) { removedCount++; if
(setFutureEventKey==e.getKey()) promise.set_value(); };
std::function<void(ClientCacheEntryExpiredEvent<int>)> listenerExpired = [&expiredCount,
&setFutureEventKey, &promise](ClientCacheEntryExpiredEvent <int> e) { expiredCount++; if
(setFutureEventKey==e.getKey()) promise.set_value(); };

cl.add_listener(listenerCreated);
cl.add_listener(listenerModified);
cl.add_listener(listenerRemoved);
cl.add_listener(listenerExpired);

cache.addClientListener(cl, filterFactoryParams, converterFactoryParams);
```

16.8.10. Hot Rod C++ Client Working with Sites

Multiple Red Hat JBoss Data Grid Server clusters may be deployed so that each cluster belongs to a different site. Such deployments are done to enable data to be backed up from one cluster to another, potentially in a different geographical location. C++ client implementation can failover between nodes within a cluster, along with failing over to a different cluster entirely, should the original cluster become nonresponsive. To be able to failover between clusters all Red Hat JBoss Data Grid Servers must be configured with Cross-Datacenter replication. Instructions for this procedure are found in the Red Hat JBoss Data Grid Administration and Configuration Guide.

Once failed over the client will remain connected to the alternative cluster until this new cluster becomes unavailable, in which case it will throw an exception. If the original cluster becomes operational, the client will not switch over automatically. To switch back to the original cluster use the **switchToDefaultCluster()** method mentioned below.

Once Cross-Datacenter replication has been configured on the servers, the client has to provide the alternative clusters' configuration with at least one host/port pair details for each of the clusters configured. For example:

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include <stdlib.h>
using namespace infinispan::hotrod;
int main(int argc, char** argv) {
    ConfigurationBuilder b;
    b.addServer().host("127.0.0.1").port(11222);
    b.addCluster("nyc").addClusterNode("127.0.0.1", 11322);

    RemoteCacheManager cm(builder.build());
    RemoteCache<std::string, std::string> cache = cm.getCache<std::string, std::string>();
    return 0;
}
```

16.8.10.1. Manual Cluster Switch

In addition to automatic site failover, C++ clients may switch between clusters by calling either of the following methods:

- **switchToCluster(clusterName)** - Forces the client to switch to the pre-defined cluster name passed in.
- **switchToDefaultCluster** - Forces the client to switch to the initial servers defined in the client configuration.

16.8.11. Performing Remote Queries via the Hot Rod C++ Client

The Hot Rod C++ client allows remote querying, using Google's Protocol Buffers, once the **RemoteCacheManager** has been configured with the Protobuf marshaller.

IMPORTANT

Performing Remote Queries is a Technology Preview feature of the Hot Rod C++ client in Red Hat JBoss Data Grid 7.2.

Enable Remote Querying on the Hot Rod C++ Client

1. Obtain a connection to the remote Red Hat JBoss Data Grid server:

```
#include "addressbook.pb.h"
#include "bank.pb.h"
#include <infinispan/hotrod/BasicTypesProtoStreamMarshaller.h>
#include <infinispan/hotrod/ProtoStreamMarshaller.h>
#include "infinispan/hotrod/ConfigurationBuilder.h"
```

```
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"
#include "infinispan/hotrod/query.pb.h"
#include "infinispan/hotrod/QueryUtils.h"
#include <vector>
#include <tuple>

#define PROTOBUF_METADATA_CACHE_NAME "___protobuf_metadata"
#define ERRORS_KEY_SUFFIX  ".errors"

using namespace infinispan::hotrod;
using namespace org::infinispan::query::remote::client;

std::string read(std::string file)
{
  std::ifstream t(file);
  std::stringstream buffer;
  buffer << t.rdbuf();
  return buffer.str();
}

int main(int argc, char** argv) {
  std::cout << "Tests for Query" << std::endl;
    ConfigurationBuilder builder;
    builder.addServer().host(argc > 1 ? argv[1] : "127.0.0.1").port(argc > 2 ? atoi(argv[2]) :
11222).protocolVersion(Configuration::PROTOCOL_VERSION_24);
    RemoteCacheManager cacheManager(builder.build(), false);
    cacheManager.start();
```

2. Create the Protobuf metadata cache with the Protobuf Marshaller:

```
    // This example continues the previous codeblock
    // Create the Protobuf Metadata cache peer with a Protobuf marshaller
    auto *km = new BasicTypesProtoStreamMarshaller<std::string>();
    auto *vm = new BasicTypesProtoStreamMarshaller<std::string>();
    auto metadataCache = cacheManager.getCache<std::string, std::string>(
        km, &Marshaller<std::string>::destroy,
        vm, &Marshaller<std::string>::destroy,PROTOBUF_METADATA_CACHE_NAME,
false);
```

3. Install the data model in the Protobuf metadata cache:

```
    // This example continues the previous codeblock
    // Install the data model into the Protobuf metadata cache
    metadataCache.put("sample_bank_account/bank.proto", read("proto/bank.proto"));
    if (metadataCache.containsKey(ERRORS_KEY_SUFFIX))
    {
        std::cerr << "fail: error in registering .proto model" << std::endl;
        return -1;
    }
```

4. This step adds data to the cache for the purposes of this demonstration, and may be ignored when simply querying a remote cache:

```
// This example continues the previous codeblock
// Fill the cache with the application data: two users Tom and Jerry
testCache.clear();
sample_bank_account::User_Address a;
sample_bank_account::User user1;
user1.set_id(3);
user1.set_name("Tom");
user1.set_surname("Cat");
user1.set_gender(sample_bank_account::User_Gender_MALE);
sample_bank_account::User_Address * addr= user1.add_addresses();
addr->set_street("Via Roma");
addr->set_number(3);
addr->set_postcode("202020");
testCache.put(3, user1);
user1.set_id(4);
user1.set_name("Jerry");
user1.set_surname("Mouse");
addr->set_street("Via Milano");
user1.set_gender(sample_bank_account::User_Gender_MALE);
testCache.put(4, user1);
```

5. Query the remote cache:

```
// This example continues the previous codeblock
// Simple query to get User objects
{
    QueryRequest qr;
    std::cout << "Query: from sample_bank_account.User" << std::endl;
    qr.set_jpqlstring("from sample_bank_account.User");
    QueryResponse resp = testCache.query(qr);
    std::vector<sample_bank_account::User> res;
    unwrapResults(resp, res);
    for (auto i : res) {
        std::cout << "User(id=" << i.id() << ",name=" << i.name()
        << ",surname=" << i.surname() << ")" << std::endl;
    }
}
cacheManager.stop();
return 0;
}
```

Additional Query Examples

The following examples are included to demonstrate more complicated queries, and may be used on the same dataset found in the above procedure.

Using a query with a conditional

```
// Simple query to get User objects with where condition
{
    QueryRequest qr;
    std::cout << "from sample_bank_account.User u where u.addresses.street=\"Via Milano\"" <<
std::endl;
    qr.set_jpqlstring("from sample_bank_account.User u where u.addresses.street=\"Via Milano\"");
    QueryResponse resp = testCache.query(qr);
```

```
    std::vector<sample_bank_account::User> res;
    unwrapResults(resp, res);
    for (auto i : res) {
        std::cout << "User(id=" << i.id() << ",name=" << i.name()
        << ",surname=" << i.surname() << ")" << std::endl;
    }
}
```

Using a query with a projection

```
// Simple query to get projection (name, surname)
{
    QueryRequest qr;
    std::cout << "Query: select u.name, u.surname from sample_bank_account.User u" << std::endl;
    qr.set_jpqlstring(
        "select u.name, u.surname from sample_bank_account.User u");
    QueryResponse resp = testCache.query(qr);

    //Typed resultset
    std::vector<std::tuple<std::string, std::string> > prjRes;
    unwrapProjection(resp, prjRes);
    for (auto i : prjRes) {
        std::cout << "Name: " << std::get<0> (i)
        << " Surname: " << std::get<1> (i) << std::endl;
    }
}
```

16.8.12. Using the Near Cache with the Hot Rod C++ Client

Near caches are optional caches for the Hot Rod C++ client that keep recently accessed data close to the user, providing faster access to data that is accessed frequently. This cache acts as a local Hot Rod client cache that are synchronized with the remote server in the background.

Near caches are enabled programmatically on the **ConfigurationBuilder** by using the **nearCache()** method, as seen in the following example:

```
int main(int argc, char** argv) {
    ConfigurationBuilder confBuilder;
    confBuilder.addServer().host("127.0.0.1").port(11222);
    confBuilder.protocolVersion(Configuration::PROTOCOL_VERSION_24);
    confBuilder.balancingStrategyProducer(nullptr);

    // Enable the near cache support
    confBuilder.nearCache().mode(NearCacheMode::INVALIDATED).maxEntries(4);
```

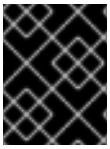
The following methods are used to configure the near cache's behavior:

- **nearCache()** - defines a **NearCacheConfigurationBuilder** which may be modified further.
- **mode(NearCacheMode mode)** - requires a **NearCacheMode** be passed in. Defaults to **DISABLED**, indicating no near cache is enabled.
- **maxEntries(int maxEntries)** - indicates the maximum number of entries for the near cache to contain. Once the near cache is full, the oldest entry will be evicted. Setting this value to **0** defines an unbounded near cache.

Entries in the near cache are kept aligned with the remote cache via events. If a change occurs in the server then an appropriate event is sent to the client, which will update the near cache accordingly.

16.8.13. Script Execution Using the Hot Rod C++ Client

The Hot Rod C++ client allows tasks to be executed directly on JBoss Data Grid servers via Remote Execution. This feature executes logic close to the data, utilizing the resources of all nodes in the cluster. Tasks may be deployed to the server instances, and may then be executed programmatically.



IMPORTANT

Remote Execution is a Technology Preview feature of the Hot Rod C++ client in Red Hat JBoss Data Grid 7.2.

Installing a Task

Tasks may be installed on the server by being using the **put(std::string name, std::string script)** method of the **___script_cache**. The extension of the script name determines the engine used to execute the script; however, this may be overridden by metadata in the script itself.

The following example demonstrates installing scripts:

Installing a Task with the C++ Client

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"
#include "infinispan/hotrod/JBasicMarshaller.h"
using namespace infinispan::hotrod;
int main(int argc, char** argv) {
   // Configure the client
   ConfigurationBuilder builder;
   builder.addServer().host("127.0.0.1").port(11222).protocolVersion(
      Configuration::PROTOCOL_VERSION_24);
   RemoteCacheManager cacheManager(builder.build(), false);
   try {
   // Create the cache with the given marshallers
   auto *km = new JBasicMarshaller<std::string>();
   auto *vm = new JBasicMarshaller<std::string>();
   RemoteCache<std::string, std::string> cache = cacheManager.getCache<
      std::string, std::string>(km, &Marshaller<std::string>::destroy,
      vm, &Marshaller<std::string>::destroy,
      std::string("namedCache"));
   cacheManager.start();

   // Obtain a reference to the ___script_cache
   RemoteCache<std::string, std::string> scriptCache =
      cacheManager.getCache<std::string, std::string>(
      "___script_cache", false);
   // Install on the server the getValue script
   std::string getValueScript(
      "// mode=local,language=javascript\n "
      "var cache = cacheManager.getCache(\"namedCache\");\n "
      "var ct = cache.get(\"accessCounter\");\n "
```

```
      "var c = ct==null ? 0 : parseInt(ct);\n "
      "cache.put(\"accessCounter\",(++c).toString());\n "
      "cache.get(\"privateValue\") ");
   std::string getValueScriptName("getValue.js");
   std::string pGetValueScriptName =
      JBasicMarshaller<std::string>::addPreamble(getValueScriptName);
   std::string pGetValueScript =
      JBasicMarshaller<std::string>::addPreamble(getValueScript);
   scriptCache.put(pGetValueScriptName, pGetValueScript);
   // Install on the server the get access counter script
   std::string getAccessScript(
      "// mode=local,language=javascript\n "
      "var cache = cacheManager.getCache(\"namedCache\");\n "
      "cache.get(\"accessCounter\")");
   std::string getAccessScriptName("getAccessCounter.js");
   std::string pGetAccessScriptName =
      JBasicMarshaller<std::string>::addPreamble(getAccessScriptName);
   std::string pGetAccessScript =
      JBasicMarshaller<std::string>::addPreamble(getAccessScript);
   scriptCache.put(pGetAccessScriptName, pGetAccessScript);
```

Executing a Task

Once installed, a task may be executed by using the **execute(std::string name, std::map<std::string, std::string> args)** method, passing in the name of the script to execute, along with any arguments that are required for execution.

The following example demonstrates executing a script:

Executing a Script with the C++ Client

```
   // The following is a continuation of the above example
   cache.put("privateValue", "Counted Access Value");
   std::map<std::string, std::string> s;
   // Execute the getValue script
   std::vector<unsigned char> execValueResult = cache.execute(
      getValueScriptName, s);
   // Execute the getAccess script
   std::vector<unsigned char> execAccessResult = cache.execute(
      getAccessScriptName, s);

   std::string value(
      JBasicMarshallerHelper::unmarshall<std::string>(
         (char*) execValueResult.data()));
   std::string access(
      JBasicMarshallerHelper::unmarshall<std::string>(
         (char*) execAccessResult.data()));

   std::cout << "Returned value is '" << value
      << "' and has been accessed: " << access << " times."
      << std::endl;

} catch (const Exception& e) {
std::cout << "is: " << typeid(e).name() << '\n';
std::cerr << "fail unexpected exception: " << e.what() << std::endl;
return 1;
```

```
    }

    cacheManager.stop();
    return 0;
  }
```

16.9. HOT ROD C# CLIENT

16.9.1. Hot Rod C# Client

The Hot Rod C# client allows .NET runtime applications to connect and interact with Red Hat JBoss Data Grid servers. This client is aware of the cluster topology and hashing scheme, and can access an entry on the server in a single hop similar to the Hot Rod Java and Hot Rod C++ clients.

The Hot Rod C# client is compatible with 64-bit operating systems on which the .NET Framework is supported by Microsoft. Visual Studio 2015 and .NET 4.6.2 are prerequisites for the Hot Rod C# client.

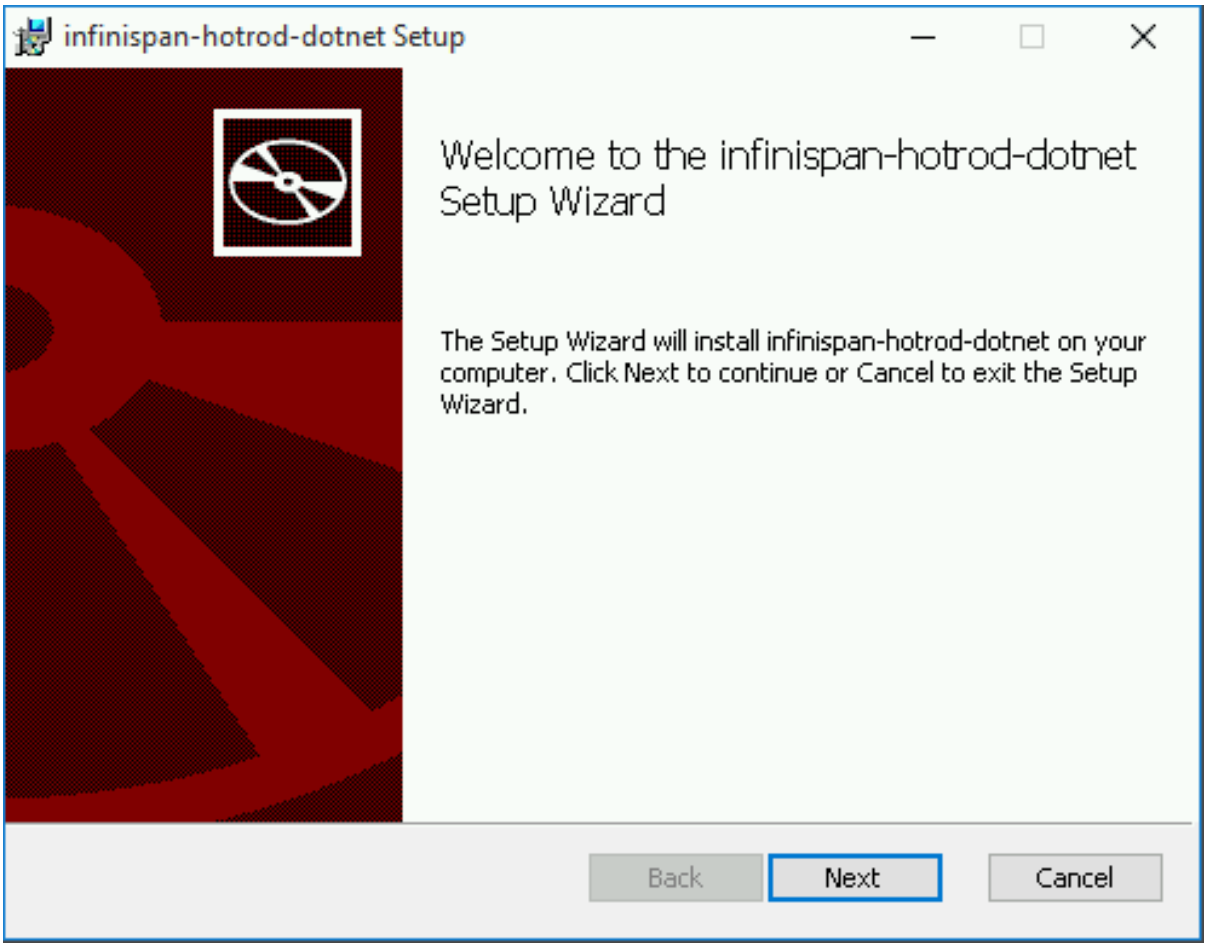
16.9.2. Hot Rod C# Client Download and Installation

The Hot Rod C# client is included in a .msi file *jboss-datagrid-<version>-hotrod-dotnet-client.msi* packed for download with Red Hat JBoss Data Grid. To install the Hot Rod C# client, execute the following instructions.

Installing the Hot Rod C# Client

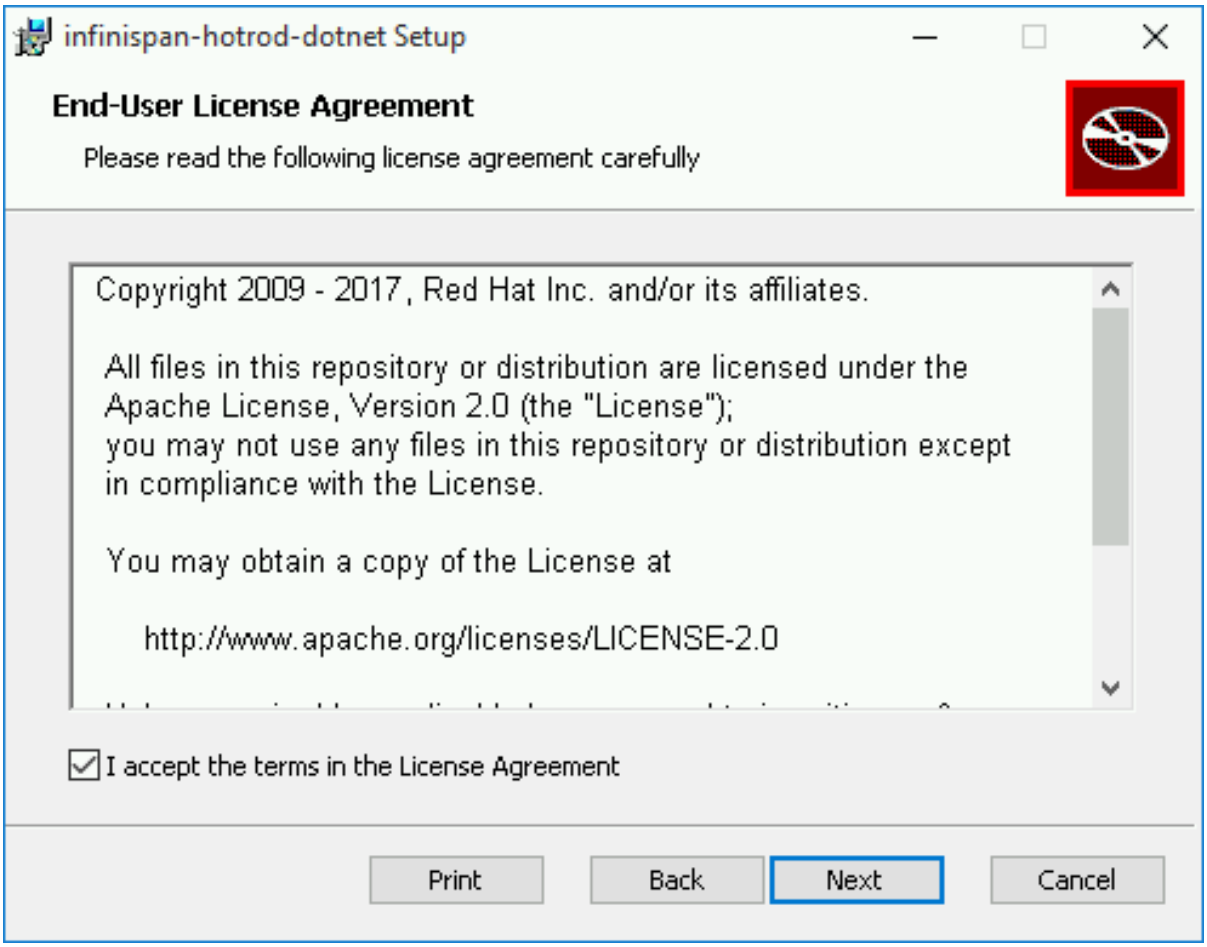
1. As an administrator, navigate to the location where the Hot Rod C# .msi file is downloaded. Run the .msi file to launch the windows installer and then click **Next**.

Figure 16.1. Hot Rod C# Client Setup Welcome



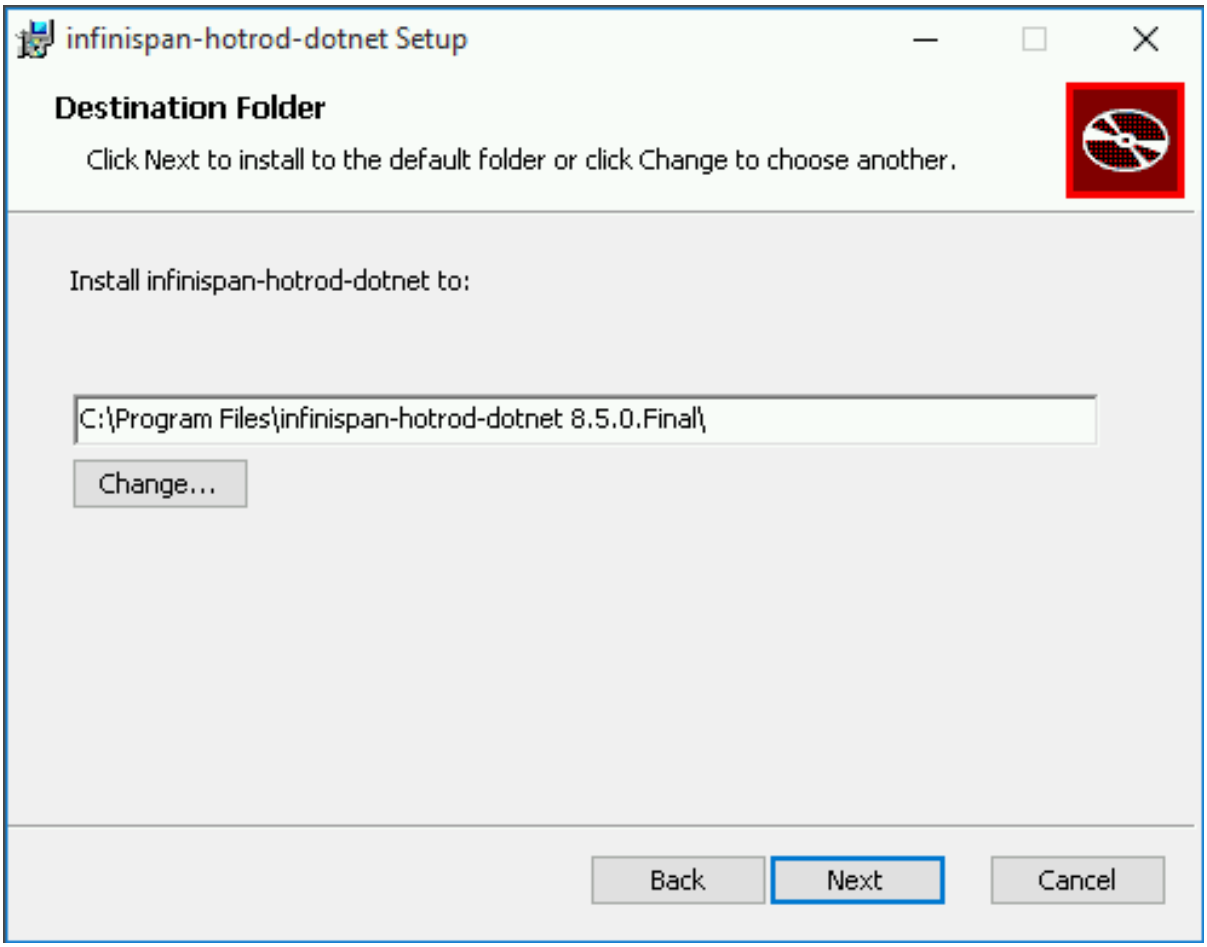
2. Review the end-user license agreement. Select the **I accept the terms in the License Agreement** check box and then click **Next**.

Figure 16.2. Hot Rod C# Client End-User License Agreement



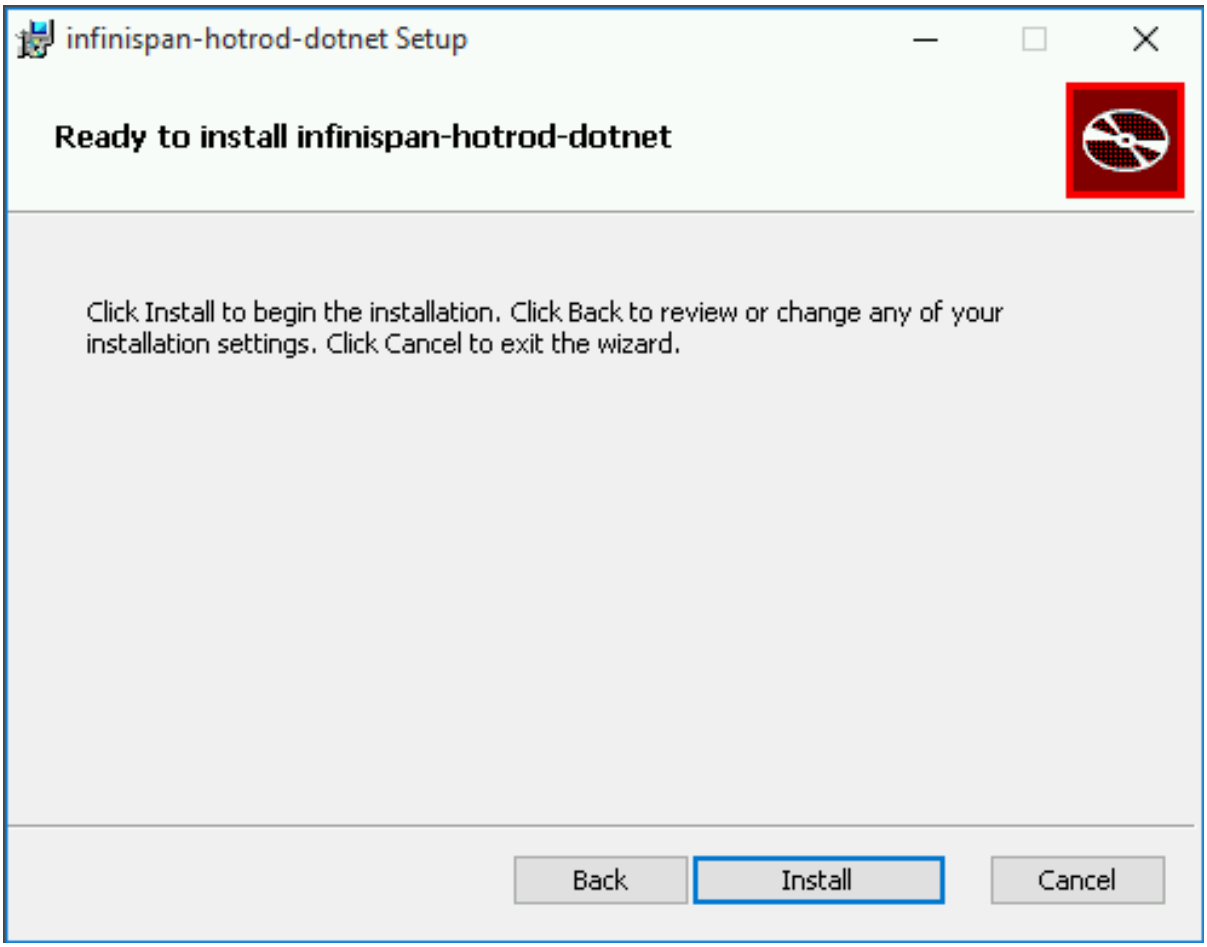
3. To change the default directory, click **Change...** or click **Next** to install in the default directory.

Figure 16.3. Hot Rod C# Client Destination Folder



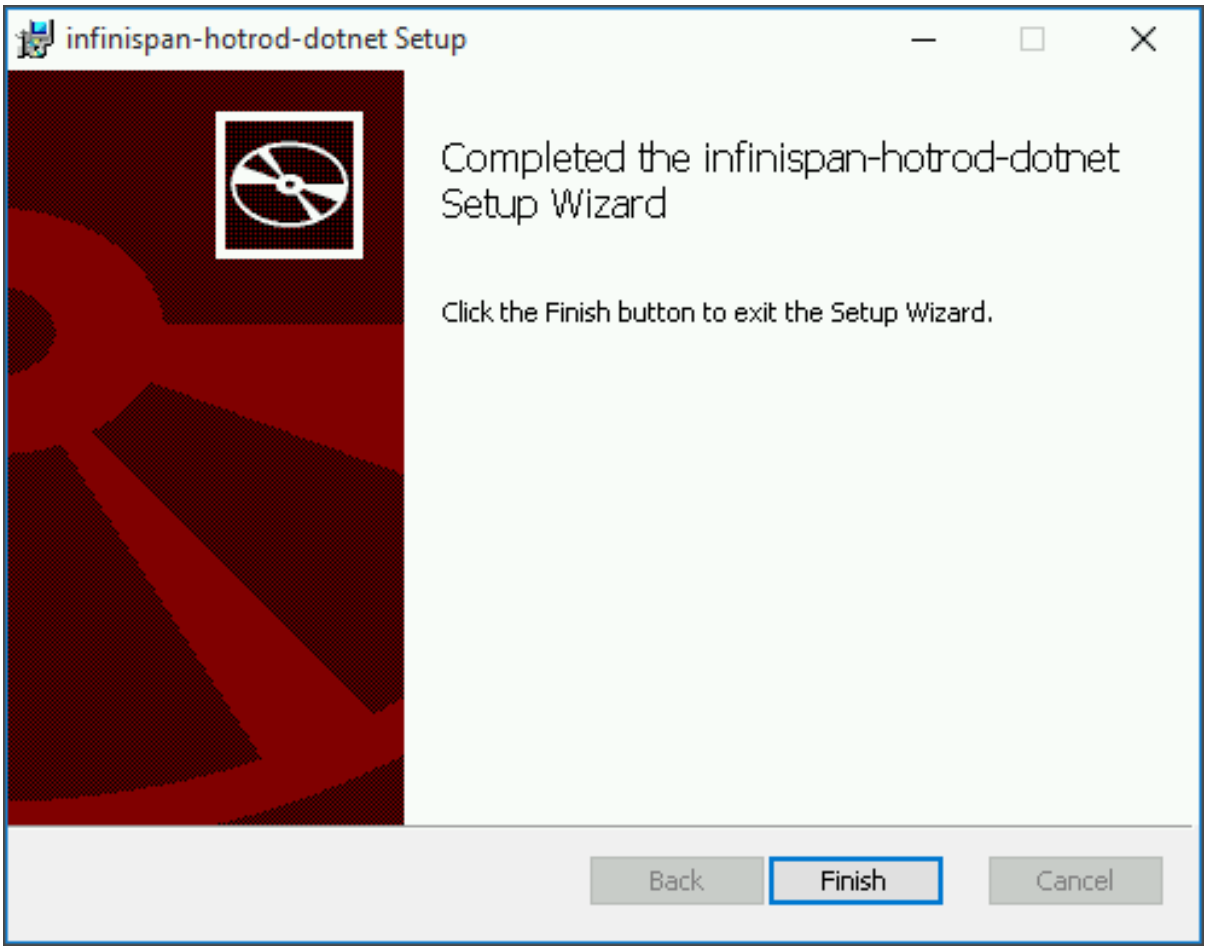
4. Click **Install** to begin the Hot Rod C# client installation.

Figure 16.4. Hot Rod C# Client Begin Installation



5. Click **Finish** to complete the Hot Rod C# client installation.

Figure 16.5. Hot Rod C# Client Setup Completion



16.9.3. Creating a Hot Rod C# .NET Project

To use the Hot Rod C# client in a .NET project the following steps must be performed:

Configure the Hot Rod C# Project

1. Add the Path Environment Variables
   The **PATH** environment variable must have the following folders added:

```
C:\path\to\infinispan-hotrod-dotnet 8.5.0.Final\bin
C:\path\to\infinispan-hotrod-dotnet 8.5.0.Final\lib
```

2. Remove **Prefer 32 bit**
   On the Project properties, under the **Build** tab, ensure that **Prefer 32 bit** is unchecked.

3. Add the Hot Rod C# dlls

   a. On the **Solution Explorer** view select **Project**.

   b. Select **References**.

   c. Right-click on references and select **Add Reference**.

   d. In the window presented, click **Browse** and navigate to the **C:\path\to\infinispan-hotrod-dotnet 8.5.0.Final\lib\hotrodcs.dll** file.

   e. Click **OK**.

The Hot Rod C# API may now be used in the .NET project.

16.9.4. Hot Rod C# Client Configuration

The Hot Rod C# client is configured programmatically using the ConfigurationBuilder. Configure the host and the port to which the client should connect.

Sample C# file configuration

The following example shows how to use the ConfigurationBuilder to configure a **RemoteCacheManager**.

C# configuration

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;
namespace simpleapp
{
   class Program
   {
      static void Main(string[] args)
      {
         ConfigurationBuilder builder = new ConfigurationBuilder();
         builder.AddServer()
            .Host(args.Length > 1 ? args[0] : "127.0.0.1")
            .Port(args.Length > 2 ? int.Parse(args[1]) : 11222);
         Configuration config = builder.Build();
         RemoteCacheManager cacheManager = new RemoteCacheManager(config);
         [...]
      }
   }
}
```

16.9.5. Hot Rod C# Client API

The **RemoteCacheManager** is a starting point to obtain a reference to a RemoteCache.

The following example shows retrieval of a default cache from the server and a few basic operations.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;
namespace simpleapp
{
   class Program
   {
```

```
        static void Main(string[] args)
        {
            ConfigurationBuilder builder = new ConfigurationBuilder();
            builder.AddServer()
                .Host(args.Length > 1 ? args[0] : "127.0.0.1")
                .Port(args.Length > 2 ? int.Parse(args[1]) : 11222);
            Configuration config = builder.Build();
            RemoteCacheManager cacheManager = new RemoteCacheManager(config);
            cacheManager.Start();
            // Retrieve a reference to the default cache.
            IRemoteCache<String, String> cache = cacheManager.GetCache<String, String>();
            // Add entries.
            cache.Put("key1", "value1");
            cache.PutIfAbsent("key1", "anotherValue1");
            cache.PutIfAbsent("key2", "value2");
            cache.PutIfAbsent("key3", "value3");
            // Retrive entries.
            Console.WriteLine("key1 -> " + cache.Get("key1"));
            // Bulk retrieve key/value pairs.
            int limit = 10;
            IDictionary<String, String> result = cache.GetBulk(limit);
            foreach (KeyValuePair<String, String> kv in result)
            {
                Console.WriteLine(kv.Key + " -> " + kv.Value);
            }
            // Remove entries.
            cache.Remove("key2");
            Console.WriteLine("key2 -> " + cache.Get("key2"));
            cacheManager.Stop();
        }
    }
}
```

16.9.6. Hot Rod C# Client Asynchronous API

The Hot Rod C# client offers asynchronous versions of many of the synchronous methods, allowing non-blocking methods for interacting with remote caches.

These methods follow the same naming convention as the synchronous methods, except that Async is appended to the end of each method. Asynchronous methods return a **Task** containing the result of the operation. If a method were to return a **String**, instead it will return a **Task<String>**

A list of asynchronous methods are below:

- **ClearAsync**
- **GetAsync**
- **PutAsync**
- **PutAllAsync**
- **PutIfAbsentAsync**
- **RemoveAsync**

- **RemoveWithVersionAsync**
- **ReplaceAsync**
- **ReplaceWithVersionAsync**

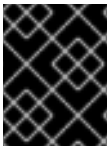
Hot Rod C# Asynchronous API Example

The following example demonstrates using these methods:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;
namespace simpleapp
{
   class Program
   {
      static void Main(string[] args)
      {
         ConfigurationBuilder builder = new ConfigurationBuilder();
         builder.AddServer()
            .Host(args.Length > 1 ? args[0] : "127.0.0.1")
            .Port(args.Length > 2 ? int.Parse(args[1]) : 11222);
         Configuration config = builder.Build();
         RemoteCacheManager cacheManager = new RemoteCacheManager(config);
         IRemoteCache<String,String> cache = cacheManager.GetCache<String,String>();

         // Add Entries Async
         cache.PutAsync("key1","value1");
         cache.PutAsync("key2","value2");

         // Retrieve Entries Async
         Task<string> futureExec = cache.GetAsync("key1");

         string result = futureExec.Result;
      }
   }
}
```

16.9.7. Hot Rod C# Client Remote Event Listeners

The Hot Rod C# client supports remote cache listeners, and these may be added using the **addListener** method on the **ClientListener**.



IMPORTANT

Remote Event Listeners is a Technology Preview feature of the Hot Rod C# client in Red Hat JBoss Data Grid 7.2.

This method takes a listener for each event type(**create**, **modify**, **remove**, **expire**, or **custom**). For more information on Remote Event Listeners refer to Remote Event Listeners (Hot Rod). An example of a modifiedEvent is provided below:

```
[...]
private static void modifiedEventAction(Event.ClientCacheEntryModifiedEvent<string> e)
{
    ++modifiedEventCounter;
    modifiedSemaphore.Release();
}
[...]
public void ModifiedEventTest()
{
    IRemoteCache<string, string> cache = remoteManager.GetCache<string, string>();
    cache.Clear();
    Event.ClientListener<string, string> cl = new Event.ClientListener<string, string>();
    cl.filterFactoryName = "";
    cl.converterFactoryName = "";
    cl.addListener(modifiedEventAction);
    cache.addClientListener(cl, new string[] { }, new string[] { }, null);
}
```

16.9.8. Hot Rod C# Client Working with Sites

Multiple Red Hat JBoss Data Grid Server clusters may be deployed so that each cluster belongs to a different site. Such deployments are done to enable data to be backed up from one cluster to another, potentially in a different geographical location. The C# client implementation can failover between nodes within a cluster, along with failing over to a different cluster entirely, should the original cluster become nonresponsive. To be able to failover between clusters all Red Hat JBoss Data Grid Servers must be configured with Cross-Datacenter replication. Instructions for this procedure are found in the Red Hat JBoss Data Grid Administration and Configuration Guide.

Once failed over the client will remain connected to the alternative cluster until this new cluster becomes unavailable, in which case it will throw an exception. If the original cluster becomes operational, the client will not switch over automatically. To switch back to the original cluster use the **SwitchToDefaultCluster()** method mentioned below.

Once Cross-Datacenter replication has been configured on the servers, the client has to provide the alternative clusters' configuration with at least one host/port pair details for each of the clusters configured. For example:

```
ConfigurationBuilder conf1 = new ConfigurationBuilder();
conf1.AddServer().Host("127.0.0.1").Port(11222);
conf1.AddCluster("nyc").AddClusterNode("127.0.0.1", 11322);
RemoteCacheManager manager1 = new RemoteCacheManager(conf1.Build(), true);

ConfigurationBuilder conf2 = new ConfigurationBuilder();
conf2.AddServer().Host("127.0.0.1").Port(11322);
conf2.AddCluster("lon").AddClusterNode("127.0.0.1", 11222);
RemoteCacheManager remoteManager = new RemoteCacheManager(conf2.Build(), true);
```

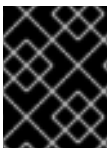
16.9.8.1. Manual Cluster Switch

In addition to automatic site failover, C++ clients may switch between clusters by calling either of the following methods:

- **SwitchToCluster(clusterName)** - Forces the client to switch to the pre-defined cluster name passed in.
- **SwitchToDefaultCluster()** - Forces the client to switch to the initial servers defined in the client configuration.

16.9.9. Performing Remote Queries via the Hot Rod C# Client

The Hot Rod C# client allows remote querying, using Google's Protocol Buffers, once the **RemoteCacheManager** has been configured with the Protobuf marshaller.



IMPORTANT

Performing Remote Queries is a Technology Preview feature of the Hot Rod C# Client in Red Hat JBoss Data Grid 7.2.

Enable Remote Querying on the Hot Rod C# Client

1. Obtain a connection to the remote JBoss Data Grid server, passing the Protobuf marshaller into the configuration:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;
using Google.Protobuf;
using Org.Infinispan.Protostream;
using Org.Infinispan.Query.Remote.Client;
using QueryExampleBankAccount;
using System.IO;

namespace Query
{
   /// <summary>
   /// This sample code shows how to perform Infinispan queries using the C# client
   /// </summary>
   class Query
   {
      static void Main(string[] args)
      {
         // Cache manager setup
         RemoteCacheManager remoteManager;
         const string ERRORS_KEY_SUFFIX = ".errors";
         const string PROTOBUF_METADATA_CACHE_NAME = "___protobuf_metadata";
         ConfigurationBuilder conf = new ConfigurationBuilder();

conf.AddServer().Host("127.0.0.1").Port(11222).ConnectionTimeout(90000).SocketTimeout(
6000);
         conf.Marshaller(new BasicTypesProtoStreamMarshaller());
```

```
            remoteManager = new RemoteCacheManager(conf.Build(), true);
            IRemoteCache<String, String> metadataCache = remoteManager.GetCache<String,
String>(PROTOBUF_METADATA_CACHE_NAME);
            IRemoteCache<int, User> testCache = remoteManager.GetCache<int, User>
("namedCache");
```

2. Install any protobuf entities model:

```
            // This example continues the previous codeblock
            // Installing the entities model into the Infinispan __protobuf_metadata cache
            metadataCache.Put("sample_bank_account/bank.proto",
File.ReadAllText("resources/proto2/bank.proto"));
            if (metadataCache.ContainsKey(ERRORS_KEY_SUFFIX))
            {
               Console.WriteLine("fail: error in registering .proto model");
               Environment.Exit(-1);
            }
```

3. This step adds data to the cache for the purposes of this demonstration, and may be ignored when simply querying a remote cache:

```
            // This example continues the previous codeblock
            // The application cache must contain entities only
            testCache.Clear();
            // Fill the application cache
            User user1 = new User();
            user1.Id = 4;
            user1.Name = "Jerry";
            user1.Surname = "Mouse";
            User ret = testCache.Put(4, user1);
```

4. Query the remote cache:

```
            // This example continues the previous codeblock
            // Run a query
            QueryRequest qr = new QueryRequest();
            qr.JpqlString = "from sample_bank_account.User";
            QueryResponse result = testCache.Query(qr);
            List<User> listOfUsers = new List<User>();
            unwrapResults(result, listOfUsers);

        }
```

5. To process the results convert the protobuf matter into C# objects. The following method demonstrates this conversion:

```
        // Convert Protobuf matter into C# objects
        private static bool unwrapResults<T>(QueryResponse resp, List<T> res) where T :
IMessage<T>
        {
            if (resp.ProjectionSize > 0)
            {  // Query has select
               return false;
            }
```

```
            for (int i = 0; i < resp.NumResults; i++)
            {
                WrappedMessage wm = resp.Results.ElementAt(i);

                if (wm.WrappedBytes != null)
                {
                    WrappedMessage wmr =
WrappedMessage.Parser.ParseFrom(wm.WrappedBytes);
                    if (wmr.WrappedMessageBytes != null)
                    {
                        System.Reflection.PropertyInfo pi = typeof(T).GetProperty("Parser");

                        MessageParser<T> p = (MessageParser<T>)pi.GetValue(null);
                        T u = p.ParseFrom(wmr.WrappedMessageBytes);
                        res.Add(u);
                    }
                }
            }
            return true;
        }
    }
}
```

16.9.10. Using the Near Cache with the Hot Rod C# Client

Near caches are optional caches for the Hot Rod C# client that keep recently accessed data close to the user, providing faster access to data that is accessed frequently. This cache acts as a local Hot Rod client cache that is synchronized with the remote server in the background.

Near caches are enabled programmatically on the **ConfigurationBuilder** by using the **NearCache()** method, as seen in the following example:

```
ConfigurationBuilder conf = new ConfigurationBuilder();
conf.AddServer().Host("127.0.0.1").Port(11222)

// Define a Near Cache that contains up to 10 entries
.NearCache().Mode(NearCacheMode.INVALIDATED).MaxEntries(10);
```

The following methods are used to configure the near cache's behavior:

- **NearCache()** - defines a **NearCacheConfigurationBuilder** which may be modified further.
- **Mode(NearCacheMode mode)** - requires a **NearCacheMode** be passed in. Defaults to **DISABLED**, indicating no near cache is enabled.
- **MaxEntries(int maxEntries)** - indicates the maximum number of entries for the near cache to contain. Once the near cache is full, the oldest entry will be evicted. Setting this value to **0** defines an unbounded near cache.

Entries in the near cache are kept aligned with the remote cache via events. If a change occurs in the server then an appropriate event is sent to the client, which will update the near cache accordingly.

16.9.11. Script Execution Using the Hot Rod C# Client

The Hot Rod C# client allows tasks to be executed directly on Red Hat JBoss Data Grid servers via Remote Execution. This feature executes logic close to the data, utilizing the resources of all nodes in the cluster. Tasks may be deployed to the server instances, and may then be executed programmatically.

Installing a Task

Tasks may be installed on the server by being using the **Put(string name, string script)** method of the **___script_cache**. The extension of the script name determines the engine used to execute the script; however, this may be overridden by metadata in the script itself.

The following example demonstrates installing scripts:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;

namespace RemoteExec
{
   /// <summary>
   /// This sample code shows how to perform a server remote execution using the C# client
   /// </summary>
   class RemoteExec
   {
      static void Main(string[] args)
      {
         // Cache manager setup
         RemoteCacheManager remoteManager;
         IMarshaller marshaller;
         ConfigurationBuilder conf = new ConfigurationBuilder();

conf.AddServer().Host("127.0.0.1").Port(11222).ConnectionTimeout(90000).SocketTimeout(6000);
         marshaller = new JBasicMarshaller();
         conf.Marshaller(marshaller);
         remoteManager = new RemoteCacheManager(conf.Build(), true);

         // Install the .js code into the Infinispan __script_cache
         const string SCRIPT_CACHE_NAME = "___script_cache";
         string valueScriptName = "getValue.js";
         string valueScript = "// mode=local,language=javascript\n "
             + "var cache = cacheManager.getCache(\"namedCache\");\n "
             + "var ct = cache.get(\"accessCounter\");\n "
             + "var c = ct==null ? 0 : parseInt(ct);\n "
             + "cache.put(\"accessCounter\",(++c).toString());\n "
             + "cache.get(\"privateValue\") ";
         string accessScriptName = "getAccess.js";
         string accessScript = "// mode=local,language=javascript\n "
            + "var cache = cacheManager.getCache(\"namedCache\");\n "
            + "cache.get(\"accessCounter\")";
         IRemoteCache<string, string> scriptCache = remoteManager.GetCache<string, string>
(SCRIPT_CACHE_NAME);
         IRemoteCache<string, string> testCache = remoteManager.GetCache<string, string>
```

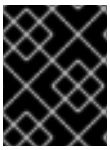
```
("namedCache");
            scriptCache.Put(valueScriptName, valueScript);
            scriptCache.Put(accessScriptName, accessScript);
```

Executing a Task

Once installed, a task may be executed by using the **Execute(string name, Dictionary<string, string> scriptArgs)** method, passing in the name of the script to execute, along with any arguments that are required for execution.

The following example demonstrates running the scripts:

```
            // This example continues the previous codeblock
            testCache.Put("privateValue", "Counted Access Value");
            Dictionary<string, string> scriptArgs = new Dictionary<string, string>();
            byte[] ret1 = testCache.Execute(valueScriptName, scriptArgs);
            string value = (string)marshaller.ObjectFromByteBuffer(ret1);
            byte[] ret2 = testCache.Execute(accessScriptName, scriptArgs);
            string accessCount = (string)marshaller.ObjectFromByteBuffer(ret2);
            Console.Write("Return value is '" + value + "' and has been accessed '" + accessCount + "'
times.");

        }
    }
}
```



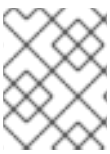
IMPORTANT

Script execution using the Hot Rod C# Client is a Technology Preview Feature in JBoss Data Grid 7.2.

16.9.12. String Marshaller for Interoperability

To use the string compatibility marshaller, pass an instance of **CompatibilityMarshaller** to the **Marshaller()** method of the **ConfigurationBuilder** object similar to this:

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.Marshaller(new CompatibilityMarshaller());
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build(), true);
IRemoteCache<String, String> cache = cacheManager.GetCache<String, String>();
[....]
cache.Put("key", "value");
[...]
cache.Get("key");
[...]
```



NOTE

Attempts to store or retrieve non-string key/values will result in a **HotRodClientException** being thrown.

16.10. HOT ROD NODE.JS CLIENT

16.10.1. Hot Rod Node.js Client

The Hot Rod Node.js client is an asynchronous event-driven client allowing Node.js users to communicate to Red Hat JBoss Data Grid servers. This client supports many of the features in the Java client, including the ability to execute and store scripts, utilize cache listeners, and receive the full cluster topology.

The asynchronous operation results are represented with **Promise** instances, allowing the client to easily chain multiple invocations together and centralizing error handling.

16.10.2. Installing the Hot Rod Node.js Client

The Hot Rod Node.js client is included in a standalone distribution that you download separately to Red Hat JBoss Data Grid.

Procedure: Installing the Hot Rod Node.js Client

1. Download the **jboss-datagrid-7.2.x-nodejs-client.zip** from the Red Hat Customer Portal.
2. Extract the downloaded archive.
3. Use **npm** to install the provided tarball, as seen in the following command:

```
npm install /path/to/jboss-datagrid-7.2.x-nodejs-client/infinispan-7.2.3-Final-redhat-00002.tgz
```

16.10.3. Hot Rod Node.js Requirements

The Hot Rod Node.js client has the following requirements:

- Node.js version 0.10 or higher.
- Red Hat JBoss Data Grid server instance 7.0.0 or higher.

16.10.4. Hot Rod Node.js Basic Functionality

The following example shows how to connect to a Red Hat JBoss Data Grid server and perform basic operations, such as putting and retrieving data. The following example assumes that a Red Hat JBoss Data Grid server is available at the default location of **localhost:11222**:

```
var infinispan = require('infinispan');

// Obtain a connection to the JBoss Data Grid server
// As no cache is specified all operations will occur on the 'default' cache
var connected = infinispan.client({port: 11222, host: '127.0.0.1'});

connected.then(function (client) {

  // Attempt to put a value in the cache.
  var clientPut = client.put('key', 'value');

  // Retrieve the value just placed
  var clientGet = clientPut.then(
     function() { return client.get('key'); });
```

```
  // Print out the value that was retrieved
  var showGet = clientGet.then(
      function(value) { console.log('get(key)=' + value); });

  // Disconnect from the server
  return showGet.finally(
      function() { return client.disconnect(); });
}).catch(function(error) {

  // Log any errors received
  console.log("Got error: " + error.message);

});
```

Connecting to a Named Cache

To connect to a specific cache the **cacheName** attribute may be defined when specifying the location of the Red Hat JBoss Data Grid server instance, as seen in the following example:

```
var infinispan = require('infinispan');

// Obtain a connection to the JBoss Data Grid server
// and connect to namedCache
var connected = infinispan.client(
  {port: 11222, host: '127.0.0.1'}, {cacheName: 'namedCache'});

connected.then(function (client) {

  // Log the result of the connection
  console.log('Connected to `namedCache`');

  // Disconnect from the server
  return client.disconnect();

}).catch(function(error) {

  // Log any errors received
  console.log("Got error: " + error.message);

});
```

Using Data Sets

In addition to placing single entries the **putAll** and **getAll** methods may be used to place or retrieve a set of data. The following example walks through these operations:

```
var infinispan = require('infinispan');

// Obtain a connection to the JBoss Data Grid server
// As no cache is specified all operations will occur on the 'default' cache
var connected = infinispan.client({port: 11222, host: '127.0.0.1'});

connected.then(function (client) {
  var data = [
    {key: 'multi1', value: 'v1'},
    {key: 'multi2', value: 'v2'},
```

```
    {key: 'multi3', value: 'v3'}];

  // Place all of the key/value pairs in the cache
  var clientPutAll = client.putAll(data);

  // Obtain the values for two of the keys
  var clientGetAll = clientPutAll.then(
    function() { return client.getAll(['multi2', 'multi3']); });

  // Print out the values obtained.
  var showGetAll = clientGetAll.then(
    function(entries) {
      console.log('getAll(multi2, multi3)=%s', JSON.stringify(entries));
    }
  );

  // Obtain an iterator for the cache
  var clientIterator = showGetAll.then(
    function() { return client.iterator(1); });

  // Iterate over the entries in the cache, printing the values
  var showIterated = clientIterator.then(
    function(it) {
      function loop(promise, fn) {
        // Simple recursive loop over iterator's next() call
        return promise.then(fn).then(function (entry) {
          return !entry.done ? loop(it.next(), fn) : entry.value;
        });
      }

      return loop(it.next(), function (entry) {
        console.log('iterator.next()=' + JSON.stringify(entry));
        return entry;
      });
    }
  );

  // Clear the cache of all values
  var clientClear = showIterated.then(
    function() { return client.clear(); });

  // Disconnect from the server
  return clientClear.finally(
    function() { return client.disconnect(); });

}).catch(function(error) {

  // Log any errors received
  console.log("Got error: " + error.message);

});
```

16.10.5. Hot Rod Node.js Conditional Operations

The Hot Rod protocol stores metadata in addition to each value associated with the keys.

The **getWithMetadata** retrieves the value and metadata for the key.

The following example demonstrates utilizing this metadata:

```
var infinispan = require('infinispan');

// Obtain a connection to the JBoss Data Grid server
// As no cache is specified all operations will occur on the 'default' cache
var connected = infinispan.client({port: 11222, host: '127.0.0.1'});

connected.then(function (client) {

  // Attempt to put a value in the cache if it does not exist
  var clientPut = client.putIfAbsent('cond', 'v0');

  // Print out the result of the put operation
  var showPut = clientPut.then(
     function(success) { console.log(':putIfAbsent(cond)=' + success); });

  // Replace the value in the cache
  var clientReplace = showPut.then(
     function() { return client.replace('cond', 'v1'); } );

  // Print out the result of the replace
  var showReplace = clientReplace.then(
     function(success) { console.log('replace(cond)=' + success); });

  // Obtain the value and metadata
  var clientGetMetaForReplace = showReplace.then(
     function() { return client.getWithMetadata('cond'); });

  // Replace the value only if the version matches
  var clientReplaceWithVersion = clientGetMetaForReplace.then(
     function(entry) {
       console.log('getWithMetadata(cond)=' + JSON.stringify(entry));
       return client.replaceWithVersion('cond', 'v2', entry.version);
     }
  );

  // Print out the result of the previous replace
  var showReplaceWithVersion = clientReplaceWithVersion.then(
     function(success) { console.log('replaceWithVersion(cond)=' + success); });

  // Obtain the value and metadata
  var clientGetMetaForRemove = showReplaceWithVersion.then(
     function() { return client.getWithMetadata('cond'); });

  // Remove the value only if the version matches
  var clientRemoveWithVersion = clientGetMetaForRemove.then(
     function(entry) {
       console.log('getWithMetadata(cond)=' + JSON.stringify(entry));
       return client.removeWithVersion('cond', entry.version);
     }
  );

  // Print out the result of the previous remove
```

```
  var showRemoveWithVersion = clientRemoveWithVersion.then(
    function(success) { console.log('removeWithVersion(cond)=' + success)});

  // Disconnect from the server
  return showRemoveWithVersion.finally(
    function() { return client.disconnect(); });

}).catch(function(error) {

  // Log any errors received
  console.log("Got error: " + error.message);

});
```

16.10.6. Hot Rod Node.js Data Sets

The client may specify multiple server addresses when a connection is defined. When multiple servers are defined it will loop through each one until a successful connection to a node is obtained. An example of this configuration is below:

```
var infinispan = require('infinispan');

// Accepts multiple addresses and fails over if connection not possible
var connected = infinispan.client(
  [{port: 99999, host: '127.0.0.1'}, {port: 11222, host: '127.0.0.1'}]);

connected.then(function (client) {

  // Obtain a list of all members in the cluster
  var members = client.getTopologyInfo().getMembers();

  // Print out the list of members
  console.log('Connected to: ' + JSON.stringify(members));

  // Disconnect from the server
  return client.disconnect();

}).catch(function(error) {

  // Log any errors received
  console.log("Got error: " + error.message);

});
```

16.10.7. Hot Rod Node.js Remote Events

The Hot Rod Node.js client supports remote cache listeners, and these may be added using the **addListener** method. This method takes the event type (**create**, **modify**, **remove**, or **expiry**) and the function callback as parameter. For more information on Remote Event Listeners refer to Remote Event Listeners (Hot Rod). An example of this is shown below:

```
var infinispan = require('infinispan');
var Promise = require('promise');
```

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'});

connected.then(function (client) {

  var clientAddListenerCreate = client.addListener(
    'create', function(key) { console.log('[Event] Created key: ' + key); });

  var clientAddListeners = clientAddListenerCreate.then(
    function(listenerId) {
      // Multiple callbacks can be associated with a single client-side listener.
      // This is achieved by registering listeners with the same listener id
      // as shown in the example below.
      var clientAddListenerModify = client.addListener(
        'modify', function(key) { console.log('[Event] Modified key: ' + key); },
        {listenerId: listenerId});

      var clientAddListenerRemove = client.addListener(
        'remove', function(key) { console.log('[Event] Removed key: ' + key); },
        {listenerId: listenerId});

      return Promise.all([clientAddListenerModify, clientAddListenerRemove]);
    });

  var clientCreate = clientAddListeners.then(
    function() { return client.putIfAbsent('eventful', 'v0'); });

  var clientModify = clientCreate.then(
    function() { return client.replace('eventful', 'v1'); });

  var clientRemove = clientModify.then(
    function() { return client.remove('eventful'); });

  var clientRemoveListener =
        Promise.all([clientAddListenerCreate, clientRemove]).then(
          function(values) {
            var listenerId = values[0];
            return client.removeListener(listenerId);
          });

  return clientRemoveListener.finally(
    function() { return client.disconnect(); });

}).catch(function(error) {

  console.log("Got error: " + error.message);

});
```

16.10.8. Hot Rod Node.js Working with Clusters

Red Hat JBoss Data Grid server instances may be clustered together to provide failover and capabilities for scaling up. While working with a cluster is very similar to using a single instance there are a few considerations:

- The client only needs to know about a single server's address to receive information about the entire server cluster, regardless of the cluster size.

- For distributed caches, key-based operations are routed in the cluster using the same consistent hash algorithms used by the server. This means that the client can locate where any particular key resides without the need for extra network hops.
- For distributed caches, multi-key or key-less operations are routed in round-robin fashion.
- For replicated and invalidated caches, all operations are routed in round-robin fashion, regardless of whether they are key-based or multi-key/key-less.

All routing and failover is transparent to the client, so operations executed against a cluster look identical to the code examples performed above.

The cluster topology can be obtained using the following example:

```
var infinispan = require('infinispan');

var connected = infinispan.client({port: 11322, host: '127.0.0.1'});

connected.then(function (client) {

  var members = client.getTopologyInfo().getMembers();

  // Should show all expected cluster members
  console.log('Connected to: ' + JSON.stringify(members));

  // Add your own operations here...

  return client.disconnect();

}).catch(function(error) {

  // Log any errors received
  console.log("Got error: " + error.message);

});
```

16.10.9. Hot Rod Node.js Working with Sites

Multiple Red Hat JBoss Data Grid Server clusters may be deployed so that each cluster belongs to a different site. Such deployments are done to enable data to be backed up from one cluster to another, potentially in a different geographical location. The Node.js client implementation can failover between nodes within a cluster, along with failing over to a different cluster entirely, should the original cluster become nonresponsive. To be able to failover between clusters all Red Hat JBoss Data Grid Servers must be configured with Cross-Datacenter replication. Instructions for this procedure are found in the Red Hat JBoss Data Grid Administration and Configuration Guide .

Once failed over the client will remain connected to the alternative cluster until this new cluster becomes unavailable, in which case it will try any other clusters defined, including the original server settings.

Once Cross-Datacenter replication has been configured on the servers, the client has to provide the alternative clusters' configuration with at least one host/port pair details for each of the clusters configured. For example:

```
var connected = infinispan.client({port: 11322, host: '127.0.0.1'},
```

```
{
  clusters: [
    {
      name: 'LON',
      servers: [{port: 1234, host: 'LONA1'}]
    },
    {
      name: 'NYC',
      servers: [{port: 2345, host: 'NYCB1'}, {port: 3456, host: 'NYCB2'}]
    }
  ]
});
```

16.10.9.1. Manual Cluster Switch

In addition to automatic site failover, Node.js clients may switch between site clusters manually by calling either of the following methods:

- **switchToCluster(clusterName)** - Forces the client to switch to the pre-defined cluster name passed in.
- **switchToDefaultCluster()** - Forces the client to switch to the initial servers defined in the client configuration.

For example, to manually switch to the NYC cluster the following could be used:

```
var connected = infinispan.client({port: 11322, host: '127.0.0.1'},
 {
   clusters: [
     {
       name: 'LON',
       servers: [{port: 1234, host: 'LONA1'}]
     },
     {
       name: 'NYC',
       servers: [{port: 2345, host: 'NYCB1'}, {port: 3456, host: 'NYCB2'}]
     }
   ]
 });

connected.then(function (client) {

  var switchToB = client.getTopologyInfo().switchToCluster('NYC');
  [...]
  });
```

16.10.10. Memory Profiling

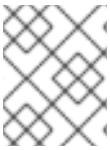
You can profile how much memory Hot Rod Node.js client consumes with the following programs:

- **infinispan_memory_many_get.js** profiles memory usage using multiple **GET** requests.
- **infinispan_memory_one_get.js** profiles memory usage using one **GET** request.

These programs are located in the **memory-profiling** directory of the client package.

To run the memory profiling programs, do the following:

```
node --expose-gc memory-profiling/infinispan_memory_many_get.js
```



NOTE

You must pass the **--expose-gc** parameter so that the programs can access the global garbage collector.

Tip: Use Google Chrome Developer Tools to visualize heap dumps. Load heap dumps from the **Memory** tab. This tab lets you compare multiple snapshots, which is useful for finding objects that have been kept in memory between points in time.

16.10.10.1. Avoiding Memory Issues with Promises

If the Node.js client creates many **Promise** instances the client can consume too much memory, which degrades performance.

The following program is an example where too many **Promise** instances are created. In this example, a user stores data and then generates multiple retrievals. The results are printed when all of the retrievals are complete, which results in increased memory consumption.

```
var _ = require('underscore');
var infinispan = require('infinispan');
var Promise = require('promise');

var heapdump = require('heapdump');

var connected = infinispan.client({port: 11222, host: '127.0.0.1'},{cacheName: 'namedCache'});
console.log("Connected to JDG server");
connected.then(function (client) {
  var sessionA = "Key";
  var clientPut = client.put(sessionA, "test");
  var clientTemp = clientPut;
  return clientTemp.then(function() {

    var initialHeapUsed = process.memoryUsage().heapUsed;
    console.log("process.memoryUsage().heapUsed: " + initialHeapUsed);
    heapdump.writeSnapshot('/tmp/' + Date.now() + '.heapsnapshot');
    var temp = [];

    var numOps = 10000; // 500000

    _.map(_.range(numOps), function(i) {
      temp.push(client.get(sessionA).then(function(value) {
        console.log("value " + value);
      }));
    });

    var promesas = Promise.all(temp);
    var completed = promesas.then(function() {
      console.log("Promises completed");
    });
```

```
      temp = null;
      promesas = null;

      return completed.then(function() {
        global.gc();
        console.log("process.memoryUsage().heapUsed (begin): " + initialHeapUsed);
        console.log("process.memoryUsage().heapUsed: "+process.memoryUsage().heapUsed);

        global.gc();
        console.log("process.memoryUsage().heapUsed: "+process.memoryUsage().heapUsed);

        heapdump.writeSnapshot('/tmp/' + Date.now() + '.heapsnapshot');

        return client.disconnect();
      });

    });
}).catch(function(err) {
  console.log("connect error", err);
});
```

The following output shows the increased memory consumption that resulted from having too many **Promise** instances created for the data retrieval:

```
node --expose-gc test.js
...
process.memoryUsage().heapUsed (begin): 5620856
process.memoryUsage().heapUsed: 14368456
process.memoryUsage().heapUsed: 14274008
```

To avoid memory issues with multiple **Promise** instances, you can either use **Promise** instances in the platform or generate a new **Promise** instance, depending on your version of Node.js.

Using Platform Promises

Recent Node.js versions include promise objects so that you do not need to load the promise library with the following line:

```
var Promise = require('promise')
```

If you remove that line from the preceding example and then run it with a Node.js version such as 8.11, the memory profiling results are as follows:

```
$ node --version
v8.11.1
$ node --expose-gc test.js
...
process.memoryUsage().heapUsed (begin): 6379448
process.memoryUsage().heapUsed: 6749056
process.memoryUsage().heapUsed: 6614560
```

Generating an Extra Promise

Older Node.js versions can generate a new **Promise** after the collection of promise objects has been handled, as in the following example:

```
var _ = require('underscore');
var infinispan = require('infinispan');
var Promise = require('promise');

var heapdump = require('heapdump');

var connected = infinispan.client({port: 11222, host: '127.0.0.1'},{cacheName: 'namedCache'});
console.log("Connected to JDG server");
connected.then(function (client) {
  var sessionA = "Key";
  var clientPut=client.put(sessionA, "test");
  var clientTemp = clientPut;
  return clientTemp.then(function() {

    var initialHeapUsed = process.memoryUsage().heapUsed;
    console.log("process.memoryUsage().heapUsed: " + initialHeapUsed);
    heapdump.writeSnapshot('/tmp/' + Date.now() + '.heapsnapshot');
    var temp = [];

    var numOps = 10000; // 500000

    _.map(_.range(numOps), function(i) {
      temp.push(client.get(sessionA).then(function(value) {
        console.log("value " + value);
      }));
    });

    var promesas = Promise.all(temp);
    var completed = promesas.then(function() {
      console.log("Promises completed");
    });

    temp = null;
    promesas = null;

    var getAfterAll = completed.then(function() {
      return client.get(sessionA);
    });

    var logGet = getAfterAll.then(function(value) {
      console.log("[get after all] value: " + value);
    })

    return logGet.then(function() {
      global.gc();
      console.log("process.memoryUsage().heapUsed (begin): " + initialHeapUsed);
      console.log("process.memoryUsage().heapUsed: "+process.memoryUsage().heapUsed);

      global.gc();
      console.log("process.memoryUsage().heapUsed: "+process.memoryUsage().heapUsed);

      heapdump.writeSnapshot('/tmp/' + Date.now() + '.heapsnapshot');

      return client.disconnect();
    });
  });
```

```
}).catch(function(err) {
  console.log("connect error", err);
});
```

The preceding example has the following the memory profiling results:

```
$ node --version
v0.10.48
$ node --expose-gc test.js
...
process.memoryUsage().heapUsed (begin): 5735864
process.memoryUsage().heapUsed: 4054352
process.memoryUsage().heapUsed: 4050064
```

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

Red Hat JBoss Data Grid provides interoperability between Hot Rod Java and Hot Rod C++ clients to access structured data. This is made possible by structuring and serializing data using Google's Protobuf format.

For example, using interoperability between languages would allow a Hot Rod C++ client to write the following **Person** object structured and serialized using Protobuf, and the Hot Rod Java client can read the same **Person** object structured as Protobuf.

Using Interoperability Between Languages

```
package sample;
message Person {
    required int32 age = 1;
    required string name = 2;
}
```

Interoperability between C++ and Hot Rod Java Client is fully supported for primitive data types, strings, and byte arrays, as Protobuf and Protostream are not required for these types of interoperability.

16.12. COMPATIBILITY BETWEEN SERVER AND HOT ROD CLIENT VERSIONS

Hot Rod clients, such as the Hot Rod Java, Hot Rod C++, and Hot Rod C#, are compatible with different versions of Red Hat JBoss Data Grid server. The server should be of the latest version in order to run with different Hot Rod clients.

NOTE

It is recommended to use the same version of the Hot Rod client and the Red Hat JBoss Data Grid server, except in a case of migration or upgrade, to prevent any known problems.

Consider the following scenarios.

Scenario 1: Server running on a newer version than the Hot Rod client.

The following will be the impact on the client side:

- client will not have advantage of the latest protocol improvements.
- client might run into known issues which are fixed for the server-side version.
- client can only use the functionalities available in its current version and the previous versions.

Scenario 2: Hot Rod client running on a newer version than the server.

In this case, when a Hot Rod client connects to a Red Hat JBoss Data Grid server, the connection will be rejected with an exception error. The client can be downgraded to a known protocol version by either setting the client side property **infinispan.client.hotrod.protocol_version**, or by using the **ConfigurationBuilder**'s **protocolVersion(String version)** method. When downgraded the client version using either of these methods a **String** containing the desired version should be passed in. In this case the client is able to connect to the server, but will be restricted to the functionality of that version. Any command which is not supported by this protocol version will not work and throw an exception; in addition, the topology information might be inefficient in this case.

Downgrading Client Hot Rod Protocol Version

The following code snippet demonstrates how to downgrade this version using the **protocolVersion(String version)** method:

```
Configuration config = new ConfigurationBuilder()
   [...]
   .protocolVersion("2.2")
   .build();
```

NOTE

It is not recommended to use this approach without guidance from Red Hat support.

The following table details the compatibility between different Hot Rod client and server versions.

Table 16.78. Hot Rod protocol and server compatibility

Red Hat JBoss Data Grid Server Version	Hot Rod Protocol Version
Red Hat JBoss Data Grid 7.2.0	Hot Rod 2.5 and later
Red Hat JBoss Data Grid 7.1.0	Hot Rod 2.5 and later
Red Hat JBoss Data Grid 7.0.0	Hot Rod 2.5 and later

PART II. CREATING AND USING INFINISPAN QUERIES IN RED HAT JBOSS DATA GRID

CHAPTER 17. GETTING STARTED WITH INFINISPAN QUERY

17.1. INTRODUCTION

The Red Hat JBoss Data Grid Library mode Querying API enables you to search for entries in the grid using properties of the values instead of keys. It provides features such as:

- Keyword, Range, Fuzzy, Wildcard, and Phrase queries
- Combining queries
- Sorting, filtering, and pagination of query results

This API, which is based on Apache Lucene and Hibernate Search, is supported in Red Hat JBoss Data Grid. Additionally, Red Hat JBoss Data Grid provides an alternate mechanism that allows both indexless and indexed searching. See The Infinispan Query DSL for details.

Enabling Querying

The Querying API is enabled by default in Remote Client-Server Mode. Instructions for enabling Querying in Library Mode are found in the Red Hat JBoss Data Grid Administration and Configuration Guide. .

17.2. INSTALLING QUERYING FOR RED HAT JBOSS DATA GRID

In Red Hat JBoss Data Grid, the JAR files required to perform queries are packaged within the Red Hat JBoss Data Grid Library and Remote Client-Server mode downloads.

For details about downloading and installing Red Hat JBoss Data Grid, see the Download and Install JBoss Data Grid chapter in the *Getting Started Guide*.

In addition, the following Maven dependency must be defined:

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

WARNING

The Infinispan query API directly exposes the Hibernate Search and the Lucene APIs and cannot be embedded within the *infinispan-embedded-query.jar* file. Do not include other versions of Hibernate Search and Lucene in the same deployment as *infinispan-embedded-query* . This action will cause classpath conflicts and result in unexpected behavior.

17.3. ABOUT QUERYING IN RED HAT JBOSS DATA GRID

17.3.1. Hibernate Search and the Query Module

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

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

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

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

WARNING

The query capability is currently intended for rich domain objects, and primitive values are not currently supported for querying.

17.3.2. Apache Lucene and the Query Module

In order to perform querying on the entire data set stored in the distributed grid, Red Hat JBoss Data Grid utilizes the capabilities of the Apache Lucene indexing tool, as well as Hibernate Search.

- Apache Lucene is a document indexing tool and search engine. JBoss Data Grid uses Apache Lucene 5.5.1.
- JBoss Data Grid's Query Module is a toolkit based on Hibernate Search that reduces Java objects into a format similar to a document, which is able to be indexed and queried by Apache Lucene.

In JBoss Data Grid, the Query Module indexes values annotated with Hibernate Search indexing annotations, then updates the index based in Apache Lucene accordingly.

Hibernate Search intercepts changes to entries stored in the data grid to generate corresponding indexing operations

17.4. INDEXING

17.4.1. Indexing

When indexing is set up, the Query module transparently indexes every added, updated, or removed cache entry. Indices improve performance of queries, though induce additional overhead during updates. For index-less querying see The Infinispan Query DSL.

For data that already exists in the grid, create an initial Lucene index. After relevant properties and annotations are added, trigger an initial batch index as shown in Rebuilding the Index.

17.4.2. Indexing with Transactional and Non-transactional Caches

In Red Hat JBoss Data Grid, the relationship between transactions and indexing is as follows:

- If the cache is transactional, index updates are applied using a listener after the commit process (after-commit listener). Index update failure does not cause the write to fail.
- If the cache is not transactional, index updates are applied using a listener that works after the event completes (post-event listener). Index update failure does not cause the write to fail.

17.4.3. Configure Indexing Programmatically

Indexing can be configured programmatically, avoiding **XML** configuration files.

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

Configure Indexing Programmatically

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

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

Configuration infinispanConfiguration = new ConfigurationBuilder()
      .indexing()
      .index(Index.LOCAL)
      .withProperties(properties)
      .build();

DefaultCacheManager cacheManager = new DefaultCacheManager(infinispanConfiguration);

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

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

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

17.4.4. Rebuilding the Index

You can manually rebuild the Lucene index if required. However, you do not usually need to rebuild the index manually because JBoss Data Grid maintains the index during normal operation.

Rebuilding the index actually reconstructs the entire index from the data store, which requires JBoss Data Grid to process all data in the grid and can take a very long time to complete. You should only need to rebuild the Lucene index if:

- The definition of what is indexed in the types has changed.

- A parameter affecting how the index is defined, such as the **Analyser** changes.
- The index is destroyed or corrupted, possibly due to a system administration error.

Server Mode

To rebuild the index in remote JBoss Data Grid servers, call the **reindexCache()** method in the **RemoteCacheManagerAdmin** HotRod client interface, for example:

```
remoteCacheManager.administration().reindexCache("MyCache");
```

Library Mode

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

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

17.5. SEARCHING

To execute a search, create a Lucene query (see Building a Lucene Query Using the Lucene-based Query API). Wrap the query in a **org.infinispan.query.CacheQuery** to get the required functionality from the Lucene-based API. The following code prepares a query against the indexed fields. Executing the code returns a list of **Book**s.

Using Infinispan Query to Create and Execute a Search

```
QueryBuilder qb = Search.getSearchManager(cache).buildQueryBuilderForClass(Book.class).get();

org.apache.lucene.search.Query query = qb
   .keyword()
   .onFields("title", "author")
   .matching("Java rocks!")
   .createQuery();

// wrap Lucene query in a org.infinispan.query.CacheQuery
CacheQuery cacheQuery = Search.getSearchManager(cache).getQuery(query);

List list = cacheQuery.list();
```

CHAPTER 18. ANNOTATING OBJECTS AND QUERYING

18.1. ANNOTATING OBJECTS AND QUERYING

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

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

- **@Indexed**

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

Annotating Objects with @Field

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

For other annotations and options, see Mapping Domain Objects to the Index Structure .

IMPORTANT

When using JBoss EAP modules with JBoss Data Grid with the domain model as a module, add the org.infinispan.query dependency with slot **7.2** into the *module.xml* file. The custom annotations are not picked by the queries without the org.infinispan.query dependency and results in an error.

18.2. REGISTERING A TRANSFORMER VIA ANNOTATIONS

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

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

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

Annotating the Key Type

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

}

public class CustomTransformer implements Transformer {
   @Override
   public Object fromString(String s) {
```

```
        return new CustomKey(...);
    }

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

The two methods must implement a biunique correspondence.

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

Biunique Correspondence

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

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

18.3. QUERYING EXAMPLE

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

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

Annotating the Person Object

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

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

Creating the SearchManager and QueryBuilder

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

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

Running the Query

```
CacheQuery query = manager.getQuery(luceneQuery);
List<Object> results = query.list();
for (Object result : results) {
    System.out.println("Found " + result);
}
```

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

CHAPTER 19. MAPPING DOMAIN OBJECTS TO THE INDEX STRUCTURE

19.1. BASIC MAPPING

19.1.1. Basic Mapping

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

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

The Lucene-based Query API uses the following common annotations to map entities:

- @Indexed
- @Field
- @NumericField

19.1.2. @Indexed

The **@Indexed** annotation declares a cached entry indexable. All entries not annotated with **@Indexed** are ignored.

Making a class indexable with @Indexed

```
@Indexed
public class Essay {
}
```

Optionally, specify the **index** attribute of the @Indexed annotation to change the default name of the index.

19.1.3. @Field

Each property or attribute of an entity can be indexed. Properties and attributes are not annotated by default, and therefore are ignored by the indexing process. The **@Field** annotation declares a property as indexed and allows the configuration of several aspects of the indexing process by setting one or more of the following attributes:

name

The name under which the property will be stored in the Lucene Document. By default, this attribute is the same as the property name, following the JavaBeans convention.

store

Specifies if the property is stored in the Lucene index. When a property is stored it can be retrieved in its original value from the Lucene Document. This is regardless of whether or not the element is indexed. Valid options are:

- **Store.YES**: Consumes more index space but allows projection. See Projection.

- **Store.COMPRESS**: Stores the property as compressed. This attribute consumes more CPU.
- **Store.NO**: No storage. This is the default setting for the store attribute.

index

Describes if property is indexed or not. The following values are applicable:

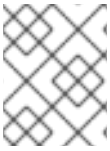
- **Index.NO**: No indexing is applied; cannot be found by querying. This setting is used for properties that are not required to be searchable, but are able to be projected.
- **Index.YES**: The element is indexed and is searchable. This is the default setting for the index attribute.

analyze

Determines if the property is analyzed. The analyze attribute allows a property to be searched by its contents. For example, it may be worthwhile to analyze a text field, whereas a date field does not need to be analyzed. Enable or disable the Analyze attribute using the following:

- **Analyze.YES**
- **Analyze.NO**

The analyze attribute is enabled by default. The **Analyze.YES** setting requires the property to be indexed via the **Index.YES** attribute.



NOTE

It is not possible to use relational operators if properties are analyzed with the **@Field(analyze=Analyze.YES)** annotation.

The following attributes are used for sorting, and must not be analyzed.

norms

Determines whether or not to store index time boosting information. Valid settings are:

- **Norms.YES**
- **Norms.NO**

The default for this attribute is **Norms.YES**. Disabling norms conserves memory, however no index time boosting information will be available.

termVector

Describes collections of term-frequency pairs. This attribute enables the storing of the term vectors within the documents during indexing. The default value is **TermVector.NO**. Available settings for this attribute are:

- **TermVector.YES**: Stores the term vectors of each document. This produces two synchronized arrays, one contains document terms and the other contains the term's frequency.
- **TermVector.NO**: Does not store term vectors.

- **TermVector.WITH_OFFSETS**: Stores the term vector and token offset information. This is the same as **TermVector.YES** plus it contains the starting and ending offset position information for the terms.
- **TermVector.WITH_POSITIONS**: Stores the term vector and token position information. This is the same as **TermVector.YES** plus it contains the ordinal positions of each occurrence of a term in a document.
- **TermVector.WITH_POSITION_OFFSETS**: Stores the term vector, token position and offset information. This is a combination of the **YES**, **WITH_OFFSETS**, and **WITH_POSITIONS**.

indexNullAs

This attribute provides replacement values for null properties. The value must conform to the following format requirements:

- String values have no format requirement.
- Numeric values must use formats accepted by **Double.parseDouble()**, **Integer.parseInteger()**, and other primitive parsing methods, depending on the field type.
- Boolean values must be either **true** or **false**.
- Date values, such as **java.util.Calendar**, **java.util.Date**, and **java.time.***, must use the ISO-8601 format.

19.1.4. @NumericField

The **@NumericField** annotation can be specified in the same scope as **@Field**.

The **@NumericField** annotation can be specified for Integer, Long, Float, and Double properties. At index time the value will be indexed using a Trie structure. When a property is indexed as numeric field, it enables efficient range query and sorting, orders of magnitude faster than doing the same query on standard **@Field** properties. The **@NumericField** annotation accept the following optional parameters:

- **forField**: Specifies the name of the related **@Field** that will be indexed as numeric. It is mandatory when a property contains more than a **@Field** declaration.
- **precisionStep**: Changes the way that the Trie structure is stored in the index. Smaller **precisionSteps** lead to more disk space usage, and faster range and sort queries. Larger values lead to less space used, and range query performance closer to the range query in normal **@Fields**. The default value for **precisionStep** is 4.

@NumericField supports only **Double**, **Long**, **Integer**, and **Float**. It is not possible to take any advantage from a similar functionality in Lucene for the other numeric types, therefore remaining types must use the string encoding via the default or custom **TwoWayFieldBridge**.

Custom **NumericFieldBridge** can also be used. Custom configurations require approximation during type transformation. The following is an example defines a custom **NumericFieldBridge**.

Defining a custom NumericFieldBridge

```
public class BigDecimalNumericFieldBridge extends NumericFieldBridge {
   private static final BigDecimal storeFactor = BigDecimal.valueOf(100);

   @Override
```

```
    public void set(String name,
                    Object value,
                    Document document,
                    LuceneOptions luceneOptions) {
       if (value != null) {
          BigDecimal decimalValue = (BigDecimal) value;
          Long indexedValue = Long.valueOf(
             decimalValue
             .multiply(storeFactor)
             .longValue());
          luceneOptions.addNumericFieldToDocument(name, indexedValue, document);
       }
    }

    @Override
    public Object get(String name, Document document) {
       String fromLucene = document.get(name);
       BigDecimal storedBigDecimal = new BigDecimal(fromLucene);
       return storedBigDecimal.divide(storeFactor);
    }
}
```

19.2. MAPPING PROPERTIES MULTIPLE TIMES

Properties may need to be mapped multiple times per index, using different indexing strategies. For example, sorting a query by field requires that the field is not analyzed. To search by words in this property and sort it, the property will need to be indexed twice - once analyzed and once un-analyzed. **@Fields** can be used to perform this search. For example:

Using @Fields to map a property multiple times

```
@Indexed(index = "Book")
public class Book {
   @Fields( {
      @Field,
      @Field(name = "summary_forSort", analyze = Analyze.NO, store = Store.YES)
   })
   public String getSummary() {
      return summary;
   }
}
```

In the example above, the field **summary** is indexed twice - once as **summary** in a tokenized way, and once as **summary_forSort** in an untokenized way. **@Field** supports 2 attributes useful when **@Fields** is used:

- analyzer: defines a @Analyzer annotation per field rather than per property
- bridge: defines a @FieldBridge annotation per field rather than per property

19.3. EMBEDDED AND ASSOCIATED OBJECTS

19.3.1. Embedded and Associated Objects

Associated objects and embedded objects can be indexed as part of the root entity index. This allows searches of an entity based on properties of associated objects.

19.3.2. Indexing Associated Objects

The aim of the following example is to return places where the associated city is Atlanta via the Lucene query **address.city:Atlanta**. The place fields are indexed in the **Place** index. The **Place** index documents also contain the following fields:

- **address.street**
- **address.city**

These fields are also able to be queried.

Indexing associations

```
@Indexed
public class Place {

    @Field
    private String name;

    @IndexedEmbedded
    @ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
    private Address address;
}

public class Address {

    @Field
    private String street;

    @Field
    private String city;

    @ContainedIn
    @OneToMany(mappedBy = "address")
    private Set<Place> places;
}
```

19.3.3. @IndexedEmbedded

When using the **@IndexedEmbedded** technique, data is denormalized in the Lucene index. As a result, the Lucene-based Query API must be updated with any changes in the **Place** and **Address** objects to keep the index up to date. Ensure the **Place** Lucene document is updated when its **Address** changes by marking the other side of the bidirectional relationship with **@ContainedIn**. **@ContainedIn** can be used for both associations pointing to entities and on embedded objects.

The **@IndexedEmbedded** annotation can be nested. Attributes can be annotated with **@IndexedEmbedded**. The attributes of the associated class are then added to the main entity index. In the following example, the index will contain the following fields:

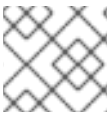
- name

- address.street
- address.city
- address.ownedBy_name

Nested usage of @IndexedEmbedded and @ContainedIn

```
@Indexed
public class Place {
   @Field
   private String name;

   @IndexedEmbedded
   @ManyToOne(cascade = {CascadeType.PERSIST, CascadeType.REMOVE})
   private Address address;
}

public class Address {
   @Field
   private String street;

   @Field
   private String city;

   @IndexedEmbedded(depth = 1, prefix = "ownedBy_")
   private Owner ownedBy;

   @ContainedIn
   @OneToMany(mappedBy = "address")
   private Set<Place> places;
}

public class Owner {
   @Field
   private String name;
}
```

The default prefix is **propertyName**, following the traditional object navigation convention. This can be overridden using the prefix attribute as it is shown on the **ownedBy** property.



NOTE

The prefix cannot be set to the empty string.

The **depth** property is used when the object graph contains a cyclic dependency of classes. For example, if **Owner** points to **Place**. the Query Module stops including attributes after reaching the expected depth, or object graph boundaries. A self-referential class is an example of cyclic dependency. In the provided example, because depth is set to 1, any **@IndexedEmbedded** attribute in **Owner** is ignored.

Using **@IndexedEmbedded** for object associations allows queries to be expressed using Lucene's query syntax. For example:

- Return places where name contains JBoss and where address city is Atlanta. In Lucene query this is:

```
+name:jboss +address.city:atlanta
```

- Return places where name contains JBoss and where owner's name contain Joe. In Lucene query this is:

```
+name:jboss +address.ownedBy_name:joe
```

This operation is similar to the relational join operation, without data duplication. Out of the box, Lucene indexes have no notion of association; the join operation does not exist. It may be beneficial to maintain the normalized relational model while benefiting from the full text index speed and feature richness.

An associated object can be also be **@Indexed**. When **@IndexedEmbedded** points to an entity, the association must be directional and the other side must be annotated using **@ContainedIn**. If not, the Lucene-based Query API cannot update the root index when the associated entity is updated. In the provided example, a **Place** index document is updated when the associated Address instance updates.

19.3.4. The targetElement Property

It is possible to override the object type targeted using the **targetElement** parameter. This method can be used when the object type annotated by **@IndexedEmbedded** is not the object type targeted by the data grid and the Lucene-based Query API. This occurs when interfaces are used instead of their implementation.

Using the targetElement property of @IndexedEmbedded

```
@Indexed
public class Address {

   @Field
   private String street;

   @IndexedEmbedded(depth = 1, prefix = "ownedBy_", targetElement = Owner.class)
   private Person ownedBy;

   ...
}

public class Owner implements Person { ... }
```

19.4. BOOSTING

19.4.1. Boosting

Lucene uses boosting to attach more importance to specific fields or documents over others. Lucene differentiates between index and search-time boosting.

19.4.2. Static Index Time Boosting

The **@Boost** annotation is used to define a static boost value for an indexed class or property. This annotation can be used within **@Field**, or can be specified directly on the method or class level.

In the following example:

- the probability of Essay reaching the top of the search list will be multiplied by 1.7.
- **@Field.boost** and **@Boost** on a property are cumulative, therefore the summary field will be 3.0 (2 x 1.5), and more important than the ISBN field.
- The text field is 1.2 times more important than the ISBN field.

Different ways of using @Boost

```
@Indexed
@Boost(1.7f)
public class Essay {

   @Field(name = "Abstract", store=Store.YES, boost = @Boost(2f))
   @Boost(1.5f)
   public String getSummary() { return summary; }

   @Field(boost = @Boost(1.2f))
   public String getText() { return text; }

   @Field
   public String getISBN() { return isbn; }

}
```

19.4.3. Dynamic Index Time Boosting

The **@Boost** annotation defines a static boost factor that is independent of the state of the indexed entity at runtime. However, in some cases the boost factor may depend on the actual state of the entity. In this case, use the **@DynamicBoost** annotation together with an accompanying custom **BoostStrategy**.

@Boost and **@DynamicBoost** annotations can both be used in relation to an entity, and all defined boost factors are cumulative. The **@DynamicBoost** can be placed at either class or field level.

In the following example, a dynamic boost is defined on class level specifying **VIPBoostStrategy** as implementation of the **BoostStrategy** interface used at indexing time. Depending on the annotation placement, either the whole entity is passed to the **defineBoost** method or only the annotated field/property value. The passed object must be cast to the correct type.

Dynamic boost example

```
public enum PersonType {
   NORMAL,
   VIP
}

@Indexed
@DynamicBoost(impl = VIPBoostStrategy.class)
public class Person {
   private PersonType type;
}

public class VIPBoostStrategy implements BoostStrategy {
   public float defineBoost(Object value) {
```

```
        Person person = (Person) value;
        if (person.getType().equals(PersonType.VIP)) {
            return 2.0f;
        }
        else {
            return 1.0f;
        }
    }
}
```

In the provided example all indexed values of a VIP would be twice the importance of the values of a non-VIP.

NOTE

The specified **BoostStrategy** implementation must define a public no argument constructor.

19.5. ANALYSIS

Analysis is the process of converting text strings into single terms that you can index and then query.

19.5.1. Default Analyzer and Analyzer by Class

The default analyzer class is used to index tokenized fields, and is configurable through the **default.analyzer** property. The default value for this property is **org.apache.lucene.analysis.standard.StandardAnalyzer**.

The analyzer class can be defined per entity, property, and per **@Field**, which is useful when multiple fields are indexed from a single property.

In the following example, **EntityAnalyzer** is used to index all tokenized properties, such as name except, summary and body, which are indexed with **PropertyAnalyzer** and **FieldAnalyzer** respectively.

Different ways of using @Analyzer

```
@Indexed
@Analyzer(impl = EntityAnalyzer.class)
public class MyEntity {

    @Field
    private String name;

    @Field
    @Analyzer(impl = PropertyAnalyzer.class)
    private String summary;

    @Field(analyzer = @Analyzer(impl = FieldAnalyzer.class))
    private String body;
}
```

NOTE

Avoid using different analyzers on a single entity. Doing so can create complications in building queries, and make results less predictable, particularly if using a **QueryParser**. Use the same analyzer for indexing and querying on any field.

19.5.2. Named Analyzers

The Query Module uses analyzer definitions to deal with the complexity of the Analyzer function. Analyzer definitions are reusable by multiple **@Analyzer** declarations and includes the following:

- a name: the unique string used to refer to the definition.
- a list of **CharFilters**: each **CharFilter** is responsible to pre-process input characters before the tokenization. **CharFilters** can add, change, or remove characters. One common usage is for character normalization.
- a **Tokenizer**: responsible for tokenizing the input stream into individual words.
- a list of filters: each filter is responsible to remove, modify, or sometimes add words into the stream provided by the **Tokenizer**.

The **Analyzer** separates these components into multiple tasks, allowing individual components to be reused and components to be built with flexibility using the following procedure:

The Analyzer Process

1. The **CharFilters** process the character input.
2. **Tokenizer** converts the character input into tokens.
3. The tokens are the processed by the **TokenFilters**.

The Lucene-based Query API supports this infrastructure by utilizing the Solr analyzer framework.

<analysis_default_analyzers><title>Default Analyzer Definitions</title>
JBoss Data Grid provides a set of default analyzers as follows:

Definition	Description
standard	Splits text fields into tokens, treating whitespace and punctuation as delimiters.
simple	Tokenizes input streams by delimiting at non-letters and then converting all letters to lowercase characters. Whitespace and non-letters are discarded.
whitespace	Splits text streams on whitespace and returns sequences of non-whitespace characters as tokens.
keyword	Treats entire text fields as single tokens.

Definition	Description
stemmer	Stems English words using the Snowball Porter filter.
ngram	Generates n-gram tokens that are 3 grams in size by default.

These analyzer definitions are based on Apache Lucene and are provided "as-is". For more information about tokenizers, filters, and CharFilters, see the appropriate Lucene documentation.

If you require custom analyzer definitions, create an implementation of the **ProgrammaticSearchMappingProvider** interface packaged in a **JAR** and deploy it to JBoss Data Grid. You must also specify the **JAR** in the cache container configuration, for example:

```
<cache-container name="mycache" default-cache="default">
  <modules>
    <module name="my.analyzers.jar"/>
  </modules>
...
```

</analysis_default_analyzers>

19.5.3. Referencing Analyzer Definitions

Use the **@Analyzer** annotation to reference an analyzer definition.

Referencing an analyzer by name

```
@Indexed
@AnalyzerDef(name = "standard")
public class Team {

   @Field
   private String name;

   @Field
   private String location;

   @Field
   @Analyzer(definition = "standard")
   private String description;
}
```

Analyzer instances declared by **@AnalyzerDef** are also available by their name in the **SearchFactory**, which is useful when building queries.

```
Analyzer analyzer = Search.getSearchManager(cache).getAnalyzer("standard")
```

When querying, fields must use the same analyzer that has been used to index the field. The same tokens are reused between the query and the indexing process.

19.5.4. @AnalyzerDef for Solr

When using Maven all required Apache Solr dependencies are now defined as dependencies of the artifact **org.hibernate:hibernate-search-analyzers**. Add the following dependency:

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-search-analyzers</artifactId>
   <version>${version.hibernate.search}</version>
<dependency>
```

In the following example, a **CharFilter** is defined by its factory. In this example, a mapping char filter is used, which will replace characters in the input based on the rules specified in the mapping file. Finally, a list of filters is defined by their factories. In this example, the **StopFilter** filter is built reading the dedicated words property file. The filter will ignore case.

@AnalyzerDef and the Solr framework

1. Configure the CharFilter
   Define a **CharFilter** by factory. In this example, a mapping **CharFilter** is used, which will replace characters in the input based on the rules specified in the mapping file.

```
@AnalyzerDef(name = "customanalyzer",
   charFilters = {
      @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
         @Parameter(name = "mapping",
            value =
               "org/hibernate/search/test/analyzer/solr/mapping-chars.properties")
      })
   },
```

2. Define the Tokenizer
   A **Tokenizer** is then defined using the **StandardTokenizerFactory.class**.

```
@AnalyzerDef(name = "customanalyzer",
   charFilters = {
      @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
         @Parameter(name = "mapping",
            value =
               "org/hibernate/search/test/analyzer/solr/mapping-chars.properties")
      })
   },

   tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class)
```

3. List of Filters
   Define a list of filters by their factories. In this example, the **StopFilter** filter is built reading the dedicated words property file. The filter will ignore case.

```
@AnalyzerDef(name = "customanalyzer",
   charFilters = {
      @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
         @Parameter(name = "mapping",
            value =
               "org/hibernate/search/test/analyzer/solr/mapping-chars.properties")
      })
```

```
    },

    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {

      @TokenFilterDef(factory = ISOLatin1AccentFilterFactory.class),
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = StopFilterFactory.class, params = {
        @Parameter(name = "words",
          value= "org/hibernate/search/test/analyzer/solr/stoplist.properties" ),
        @Parameter(name = "ignoreCase", value = "true")
      })
    })
  public class Team {
  }
```

NOTE

Filters and **CharFilters** are applied in the order they are defined in the **@AnalyzerDef** annotation.

19.5.5. Loading Analyzer Resources

Tokenizers, **TokenFilters**, and **CharFilters** can load resources such as configuration or metadata files using the **StopFilterFactory.class** or the synonym filter. The virtual machine default can be explicitly specified by adding a **resource_charset** parameter.

Use a specific charset to load the property file

```
@AnalyzerDef(name = "customanalyzer",
  charFilters = {
    @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
      @Parameter(name = "mapping",
        value =
          "org/hibernate/search/test/analyzer/solr/mapping-chars.properties")
    })
  },
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
  filters = {
    @TokenFilterDef(factory = ISOLatin1AccentFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class, params = {
      @Parameter(name="words",
        value= "org/hibernate/search/test/analyzer/solr/stoplist.properties"),
      @Parameter(name = "resource_charset", value = "UTF-16BE"),
      @Parameter(name = "ignoreCase", value = "true")
    })
  })
public class Team {
}
```

19.5.6. Dynamic Analyzer Selection

The Query Module uses the **@AnalyzerDiscriminator** annotation to enable the dynamic analyzer selection.

An analyzer can be selected based on the current state of an entity that is to be indexed. This is particularly useful in multilingual applications. For example, when using the **BlogEntry** class, the analyzer can depend on the language property of the entry. Depending on this property, the correct language-specific stemmer can then be chosen to index the text.

An implementation of the **Discriminator** interface must return the name of an existing Analyzer definition, or null if the default analyzer is not overridden.

The following example assumes that the language parameter is either '**de**' or '**en**', which is specified in the **@AnalyzerDefs**.

Configure the @AnalyzerDiscriminator

1. Predefine Dynamic Analyzers
   The **@AnalyzerDiscriminator** requires that all analyzers that are to be used dynamically are predefined via **@AnalyzerDef**. The **@AnalyzerDiscriminator** annotation can then be placed either on the class, or on a specific property of the entity, in order to dynamically select an analyzer. An implementation of the **Discriminator** interface can be specified using the **@AnalyzerDiscriminatorimpl** parameter.

```
@Indexed
@AnalyzerDefs({
   @AnalyzerDef(name = "en",
      tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
      filters = {
         @TokenFilterDef(factory = LowerCaseFilterFactory.class),
         @TokenFilterDef(factory = EnglishPorterFilterFactory.class)
      }),
   @AnalyzerDef(name = "de",
      tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
      filters = {
         @TokenFilterDef(factory = LowerCaseFilterFactory.class),
         @TokenFilterDef(factory = GermanStemFilterFactory.class)
      })
   })
public class BlogEntry {

   @Field
   @AnalyzerDiscriminator(impl = LanguageDiscriminator.class)
   private String language;

   @Field
   private String text;

   private Set<BlogEntry> references;

   // standard getter/setter
}
```

2. Implement the Discriminator Interface
   Implement the **getAnalyzerDefinitionName()** method, which is called for each field added to the Lucene document. The entity being indexed is also passed to the interface method.

The **value** parameter is set if the **@AnalyzerDiscriminator** is placed on the property level instead of the class level. In this example, the value represents the current value of this property.

```
public class LanguageDiscriminator implements Discriminator {
    public String getAnalyzerDefinitionName(Object value, Object entity, String field) {
        if (value == null || !(entity instanceof Article)) {
            return null;
        }
        return (String) value;
    }
}
```

19.5.7. Retrieving an Analyzer

Retrieving an analyzer can be used when multiple analyzers have been used in a domain model, in order to benefit from stemming or phonetic approximation, etc. In this case, use the same analyzers to building a query. Alternatively, use the Lucene-based Query API, which selects the correct analyzer automatically. See Building a Lucene Query .

The scoped analyzer for a given entity can be retrieved using either the Lucene programmatic API or the Lucene query parser. A scoped analyzer applies the right analyzers depending on the field indexed. Multiple analyzers can be defined on a given entity, each working on an individual field. A scoped analyzer unifies these analyzers into a context-aware analyzer.

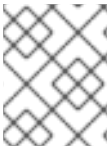
In the following example, the song title is indexed in two fields:

- Standard analyzer: used in the **title** field.
- Stemming analyzer: used in the **title_stemmed** field.

Using the analyzer provided by the search factory, the query uses the appropriate analyzer depending on the field targeted.

Using the scoped analyzer when building a full-text query

```
SearchManager manager = Search.getSearchManager(cache);

org.apache.lucene.queryparser.classic.QueryParser parser = new QueryParser(
        org.apache.lucene.util.Version.LUCENE_5_5_1,
        "title",
        manager.getAnalyzer(Song.class)
);

org.apache.lucene.search.Query luceneQuery =
        parser.parse("title:sky Or title_stemmed:diamond");

// wrap Lucene query in a org.infinispan.query.CacheQuery
CacheQuery cacheQuery = manager.getQuery(luceneQuery, Song.class);

List result = cacheQuery.list();
//return the list of matching objects
```



NOTE

Analyzers defined via **@AnalyzerDef** can also be retrieved by their definition name using **searchManager.getAnalyzer(String)**.

19.6. BRIDGE

19.6.1. Bridges

When mapping entities, Lucene represents all index fields as strings. All entity properties annotated with **@Field** are converted to strings to be indexed. Built-in bridges automatically translates properties for the Lucene-based Query API. The bridges can be customized to gain control over the translation process.

19.6.2. Built-in Bridges

The Lucene-based Query API includes a set of built-in bridges between a Java property type and its full text representation.

null

Per default **null** elements are not indexed. Lucene does not support null elements. However, in some situation it can be useful to insert a custom token representing the **null** value. See @Field for more information.

java.lang.String

Strings are indexed, as are:

- **short**, **Short**
- **integer**, **Integer**
- **long**, **Long**
- **float**, **Float**
- **double**, **Double**
- **BigInteger**
- **BigDecimal**

Numbers are converted into their string representation. Note that numbers cannot be compared by Lucene, or used in ranged queries out of the box, and must be padded

NOTE

Using a Range query has disadvantages. An alternative approach is to use a Filter query which will filter the result query to the appropriate range.

The Query Module supports using a custom **StringBridge**. See Custom Bridges.

java.util.Date

Dates are stored as **yyyyMMddHHmmssSSS** in GMT time (200611072203012 for Nov 7th of 2006 4:03PM and 12ms EST). When using a **TermRangeQuery**, dates are expressed in GMT.
@DateBridge defines the appropriate resolution to store in the index, for example: **@DateBridge(resolution=Resolution.DAY)**. The date pattern will then be truncated accordingly.

```
@Indexed
public class Meeting {
    @Field(analyze=Analyze.NO)
    @DateBridge(resolution=Resolution.MINUTE)
    private Date date;
```

The default **Date** bridge uses Lucene's **DateTools** to convert from and to **String**. All dates are expressed in GMT time. Implement a custom date bridge in order to store dates in a fixed time zone.

java.net.URI, java.net.URL

URI and URL are converted to their string representation

java.lang.Class

Class are converted to their fully qualified class name. The thread context classloader is used when the class is rehydrated

19.6.3. Custom Bridges

19.6.3.1. Custom Bridges

Custom bridges are available in situations where built-in bridges, or the bridge's String representation, do not sufficiently address the required property types.

19.6.3.2. FieldBridge

For improved flexibility, a bridge can be implemented as a **FieldBridge**. The **FieldBridge** interface provides a property value, which can then be mapped in the **Lucene Document**. For example, a property can be stored in two different document fields.

Implementing the FieldBridge Interface

```
public class DateSplitBridge implements FieldBridge {
    private final static TimeZone GMT = TimeZone.getTimeZone("GMT");

    public void set(String name,
                    Object value,
                    Document document,
                    LuceneOptions luceneOptions) {
        Date date = (Date) value;
        Calendar cal = GregorianCalendar.getInstance(GMT);
        cal.setTime(date);
        int year = cal.get(Calendar.YEAR);
        int month = cal.get(Calendar.MONTH) + 1;
        int day = cal.get(Calendar.DAY_OF_MONTH);

        // set year
        luceneOptions.addFieldToDocument(
            name + ".year",
            String.valueOf(year),
```

```
            document);

        // set month and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".month",
            month < 10 ? "0" : "" + String.valueOf(month),
            document);

        // set day and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".day",
            day < 10 ? "0" : "" + String.valueOf(day),
            document);
    }
}

//property
@FieldBridge(impl = DateSplitBridge.class)
private Date date;
```

In the following example, the fields are not added directly to the **Lucene Document**. Instead the addition is delegated to the **LuceneOptions** helper. The helper will apply the options selected on **@Field**, such as **Store** or **TermVector**, or apply the chosen **@Boost** value.

It is recommended that **LuceneOptions** is delegated to add fields to the **Document**, however the **Document** can also be edited directly, ignoring the **LuceneOptions**.

NOTE

LuceneOptions shields the application from changes in **Lucene API** and simplifies the code.

19.6.3.3. StringBridge

Use the **org.infinispan.query.bridge.StringBridge** interface to provide the Lucene-based Query API with an implementation of the expected **Object** to **String** bridge, or **StringBridge**. All implementations are used concurrently, and therefore must be thread-safe.

Custom StringBridge implementation

```
/**
 * Padding Integer bridge.
 * All numbers will be padded with 0 to match 5 digits
 *
 * @author Emmanuel Bernard
 */
public class PaddedIntegerBridge implements StringBridge {

   private int PADDING = 5;

   public String objectToString(Object object) {
      String rawInteger = ((Integer) object).toString();
      if (rawInteger.length() > PADDING)
         throw new IllegalArgumentException("Try to pad on a number too big");
      StringBuilder paddedInteger = new StringBuilder();
```

```
        for (int padIndex = rawInteger.length() ; padIndex < PADDING ; padIndex++) {
            paddedInteger.append('0');
        }
        return paddedInteger.append(rawInteger).toString();
    }
}
```

The **@FieldBridge** annotation allows any property or field in the provided example to use the bridge:

```
@FieldBridge(impl = PaddedIntegerBridge.class)
private Integer length;
```

19.6.3.4. Two-Way Bridge

A **TwoWayStringBridge** is an extended version of a **StringBridge**, which can be used when the bridge implementation is used on an ID property. The Lucene-based Query API reads the string representation of the identifier and uses it to generate an object. The **@FieldBridge** annotation is used in the same way.

Implementing a TwoWayStringBridge for ID Properties

```
public class PaddedIntegerBridge implements TwoWayStringBridge, ParameterizedBridge {

    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map parameters) {
        Object padding = parameters.get(PADDING_PROPERTY);
        if (padding != null) this.padding = (Integer) padding;
    }

    public String objectToString(Object object) {
        String rawInteger = ((Integer) object).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException("Try to pad on a number too big");
        StringBuilder paddedInteger = new StringBuilder();
        for (int padIndex = rawInteger.length(); padIndex < padding; padIndex++) {
            paddedInteger.append('0');
        }
        return paddedInteger.append(rawInteger).toString();
    }

    public Object stringToObject(String stringValue) {
        return new Integer(stringValue);
    }
}


@FieldBridge(impl = PaddedIntegerBridge.class,
             params = @Parameter(name = "padding", value = "10"))
private Integer id;
```

IMPORTANT

The two-way process must be idempotent (ie object = stringToObject(objectToString(object))).

19.6.3.5. Parameterized Bridge

A **ParameterizedBridge** interface passes parameters to the bridge implementation, making it more flexible. The **ParameterizedBridge** interface can be implemented by **StringBridge**, **TwoWayStringBridge**, **FieldBridge** implementations. All implementations must be thread-safe.

The following example implements a **ParameterizedBridge** interface, with parameters passed through the **@FieldBridge** annotation.

Configure the ParameterizedBridge Interface

```
public class PaddedIntegerBridge implements StringBridge, ParameterizedBridge {

   public static String PADDING_PROPERTY = "padding";
   private int padding = 5; //default

   public void setParameterValues(Map <String,String> parameters) {
      String padding = parameters.get(PADDING_PROPERTY);
      if (padding != null) this.padding = Integer.parseInt(padding);
   }

   public String objectToString(Object object) {
      String rawInteger = ((Integer) object).toString();
      if (rawInteger.length() > padding)
         throw new IllegalArgumentException("Try to pad on a number too big");
      StringBuilder paddedInteger = new StringBuilder();
      for (int padIndex = rawInteger.length() ; padIndex < padding ; padIndex++) {
         paddedInteger.append('0');
      }
      return paddedInteger.append(rawInteger).toString();
   }
}

//property
@FieldBridge(impl = PaddedIntegerBridge.class,
         params = @Parameter(name = "padding", value = "10")
        )
private Integer length;
```

19.6.3.6. Type Aware Bridge

Any bridge implementing **AppliedOnTypeAwareBridge** will get the type the bridge is applied on injected. For example:

- the return type of the property for field/getter-level bridges.
- the class type for class-level bridges.

The type injected does not have any specific thread-safety requirements.

19.6.3.7. ClassBridge

More than one property of an entity can be combined and indexed in a specific way to the Lucene index using the **@ClassBridge** annotation. **@ClassBridge** can be defined at class level, and supports the **termVector** attribute.

In the following example, the custom **FieldBridge** implementation receives the entity instance as the value parameter, rather than a particular property. The particular **CatFieldsClassBridge** is applied to the department instance.The **FieldBridge** then concatenates both branch and network, and indexes the concatenation.

Implementing a ClassBridge

```
@Indexed
@ClassBridge(name = "branchnetwork",
            store = Store.YES,
            impl = CatFieldsClassBridge.class,
            params = @Parameter(name = "sepChar", value = ""))
public class Department {
   private int id;
   private String network;
   private String branchHead;
   private String branch;
   private Integer maxEmployees;
}

public class CatFieldsClassBridge implements FieldBridge, ParameterizedBridge {
   private String sepChar;

   public void setParameterValues(Map parameters) {
      this.sepChar = (String) parameters.get("sepChar");
   }

   public void set(String name,
                  Object value,
                  Document document,
                  LuceneOptions luceneOptions) {

      Department dep = (Department) value;
      String fieldValue1 = dep.getBranch();
      if (fieldValue1 == null) {
         fieldValue1 = "";
      }
      String fieldValue2 = dep.getNetwork();
      if (fieldValue2 == null) {
         fieldValue2 = "";
      }
      String fieldValue = fieldValue1 + sepChar + fieldValue2;
      Field field = new Field(name, fieldValue, luceneOptions.getStore(),
         luceneOptions.getIndex(), luceneOptions.getTermVector());
      field.setBoost(luceneOptions.getBoost());
      document.add(field);
   }
}
```

CHAPTER 20. QUERYING

20.1. QUERYING

Infinispan Query can execute Lucene queries and retrieve domain objects from a Red Hat JBoss Data Grid cache.

Prepare and Execute a Query

1. Get **SearchManager** of an indexing enabled cache as follows:

```
SearchManager manager = Search.getSearchManager(cache);
```

2. Create a **QueryBuilder** to build queries for **Myth.class** as follows:

```
final org.hibernate.search.query.dsl.QueryBuilder queryBuilder =
   manager.buildQueryBuilderForClass(Myth.class).get();
```

3. Create an Apache Lucene query that queries the **Myth.class** class' atributes as follows:

```
org.apache.lucene.search.Query query = queryBuilder.keyword()
    .onField("history").boostedTo(3)
    .matching("storm")
    .createQuery();

// wrap Lucene query in a org.infinispan.query.CacheQuery
CacheQuery cacheQuery = manager.getQuery(query);

// Get query result
List<Object> result = cacheQuery.list();
```

20.2. BUILDING QUERIES

20.2.1. Building Queries

Query Module queries are built on Lucene queries, allowing users to use any Lucene query type. When the query is built, Infinispan Query uses org.infinispan.query.CacheQuery as the query manipulation API for further query processing.

20.2.2. Building a Lucene Query Using the Lucene-based Query API

With the Lucene API, use either the query parser (simple queries) or the Lucene programmatic API (complex queries). For details, see the online Lucene documentation or a copy of *Lucene in Action* or *Hibernate Search in Action* .

20.2.3. Building a Lucene Query

20.2.3.1. Building a Lucene Query

Using the Lucene programmatic API, it is possible to write full-text queries. However, when using Lucene programmatic API, the parameters must be converted to their string equivalent and must also

apply the correct analyzer to the right field. A ngram analyzer for example uses several ngrams as the tokens for a given word and should be searched as such. It is recommended to use the **QueryBuilder** for this task.

The Lucene-based query API is fluent. This API has a following key characteristics:

- Method names are in English. As a result, API operations can be read and understood as a series of English phrases and instructions.
- It uses IDE autocompletion which helps possible completions for the current input prefix and allows the user to choose the right option.
- It often uses the chaining method pattern.
- It is easy to use and read the API operations.

To use the API, first create a query builder that is attached to a given indexed type. This **QueryBuilder** knows what analyzer to use and what field bridge to apply. Several **QueryBuilder**s (one for each type involved in the root of your query) can be created. The **QueryBuilder** is derived from the **SearchManager**.

```
Search.getSearchManager(cache).buildQueryBuilderForClass(Myth.class).get();
```

The analyzer, used for a given field or fields can also be overridden.

```
SearchManager searchManager = Search.getSearchManager(cache);
   QueryBuilder mythQB = searchManager.buildQueryBuilderForClass(Myth.class)
      .overridesForField("history","stem_analyzer_definition")
      .get();
```

The query builder is now used to build Lucene queries.

20.2.3.2. Keyword Queries

The following example shows how to search for a specific word:

Keyword Search

```
Query luceneQuery = mythQB.keyword().onField("history").matching("storm").createQuery();
```

Table 20.1. Keyword query parameters

Parameter	Description
keyword()	Use this parameter to find a specific word
onField()	Use this parameter to specify in which lucene field to search the word
matching()	use this parameter to specify the match for search string
createQuery()	creates the Lucene query object

- The value "storm" is passed through the "history" **FieldBridge**. This is useful when numbers or dates are involved.
- The field bridge value is then passed to the analyzer used to index the field "history". This ensures that the query uses the same term transformation than the indexing (lower case, ngram, stemming and so on). If the analyzing process generates several terms for a given word, a boolean query is used with the **SHOULD** logic (roughly an **OR** logic).

To search a property that is not of type string.

```
@Indexed
public class Myth {
   @Field(analyze = Analyze.NO)
   @DateBridge(resolution = Resolution.YEAR)
   public Date getCreationDate() { return creationDate; }
   public void setCreationDate(Date creationDate) { this.creationDate = creationDate; }
   private Date creationDate;
}

Date birthdate = ...;
Query luceneQuery = mythQb.keyword()
   .onField("creationDate")
   .matching(birthdate)
   .createQuery();
```

NOTE

In plain Lucene, the **Date** object had to be converted to its string representation (in this case the year)

This conversion works for any object, provided that the **FieldBridge** has an **objectToString** method (and all built-in **FieldBridge** implementations do).

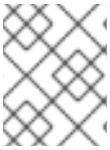
The next example searches a field that uses ngram analyzers. The ngram analyzers index succession of ngrams of words, which helps to avoid user typos. For example, the 3-grams of the word hibernate are hib, ibe, ber, rna, nat, ate.

Searching Using Ngram Analyzers

```
@AnalyzerDef(name = "ngram",
   tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
   filters = {
      @TokenFilterDef(factory = StandardFilterFactory.class),
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = StopFilterFactory.class),
      @TokenFilterDef(factory = NGramFilterFactory.class,
         params = {
            @Parameter(name = "minGramSize", value = "3"),
            @Parameter(name = "maxGramSize", value = "3")})
   })
public class Myth {
   @Field(analyzer = @Analyzer(definition = "ngram"))
   public String getName() { return name; }
   public String setName(String name) { this.name = name; }
   private String name;
```

```
}

Date birthdate = ...;
Query luceneQuery = mythQb.keyword()
    .onField("name")
    .matching("Sisiphus")
    .createQuery();
```

The matching word "Sisiphus" will be lower-cased and then split into 3-grams: sis, isi, sip, phu, hus. Each of these ngram will be part of the query. The user is then able to find the Sysiphus myth (with a **y**). All that is transparently done for the user.



NOTE

If the user does not want a specific field to use the field bridge or the analyzer then the **ignoreAnalyzer()** or **ignoreFieldBridge()** functions can be called.

To search for multiple possible words in the same field, add them all in the matching clause.

Searching for Multiple Words

```
//search document with storm or lightning in their history
Query luceneQuery =
    mythQB.keyword().onField("history").matching("storm lightning").createQuery();
```

To search the same word on multiple fields, use the **onFields** method.

Searching Multiple Fields

```
Query luceneQuery = mythQB
    .keyword()
    .onFields("history","description","name")
    .matching("storm")
    .createQuery();
```

In some cases, one field must be treated differently from another field even if searching the same term. In this case, use the **andField()** method.

Using the andField Method

```
Query luceneQuery = mythQB.keyword()
    .onField("history")
    .andField("name")
    .boostedTo(5)
    .andField("description")
    .matching("storm")
    .createQuery();
```

In the previous example, only field name is boosted to 5.

20.2.3.3. Fuzzy Queries

To execute a fuzzy query (based on the Levenshtein distance algorithm), start like a **keyword** query and add the fuzzy flag.

Fuzzy Query

```
Query luceneQuery = mythQB.keyword()
    .fuzzy()
    .withEditDistanceUpTo(1)
    .withPrefixLength(1)
    .onField("history")
    .matching("starm")
    .createQuery();
```

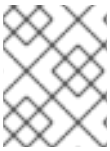
The **withEditDistanceUpTo** is the maximum value of the edit distance (Levenshtein distance) to consider two terms matching. It is an integer value between 0 and 2, with a default value of 2. The **prefixLength** is the length of the prefix ignored by the "fuzzyness". While the default value is 0, a non zero value is recommended for indexes containing a huge amount of distinct terms.

20.2.3.4. Wildcard Queries

Wildcard queries can also be executed (queries where some of parts of the word are unknown). The **?** represents a single character and ***** represents any character sequence. Note that for performance purposes, it is recommended that the query does not start with either **?** or *.

Wildcard Query

```
Query luceneQuery = mythQB.keyword()
    .wildcard()
    .onField("history")
    .matching("sto*")
    .createQuery();
```



NOTE

Wildcard queries do not apply the analyzer on the matching terms. Otherwise the risk of * or **?** being mangled is too high.

20.2.3.5. Phrase Queries

So far we have been looking for words or sets of words, the user can also search exact or approximate sentences. Use **phrase()** to do so.

Phrase Query

```
Query luceneQuery = mythQB.phrase()
    .onField("history")
    .sentence("Thou shalt not kill")
    .createQuery();
```

Approximate sentences can be searched by adding a slop factor. The slop factor represents the number of other words permitted in the sentence: this works like a within or near operator.

Adding Slop Factor

-

```
Query luceneQuery = mythQB.phrase()
    .withSlop(3)
    .onField("history")
    .sentence("Thou kill")
    .createQuery();
```

20.2.3.6. Range Queries

A range query searches for a value in between given boundaries (included or not) or for a value below or above a given boundary (included or not).

Range Query

```
//look for 0 <= starred < 3
Query luceneQuery = mythQB.range()
    .onField("starred")
    .from(0).to(3).excludeLimit()
    .createQuery();

//look for myths strictly BC
Date beforeChrist = ...;
Query luceneQuery = mythQB.range()
    .onField("creationDate")
    .below(beforeChrist).excludeLimit()
    .createQuery();
```

20.2.3.7. Combining Queries

Queries can be aggregated (combine) to create more complex queries. The following aggregation operators are available:

- **SHOULD**: the query should contain the matching elements of the subquery.
- **MUST**: the query must contain the matching elements of the subquery.
- **MUST NOT**: the query must not contain the matching elements of the subquery.

The subqueries can be any Lucene query including a boolean query itself. Following are some examples:

Combining Subqueries

```
//look for popular modern myths that are not urban
Date twentiethCentury = ...;
Query luceneQuery = mythQB.bool()
    .must(mythQB.keyword().onField("description").matching("urban").createQuery())
    .not()
    .must(mythQB.range().onField("starred").above(4).createQuery())
    .must(mythQB.range()
        .onField("creationDate")
        .above(twentiethCentury)
        .createQuery())
    .createQuery();

//look for popular myths that are preferably urban
Query luceneQuery = mythQB
```

```
    .bool()
    .should(mythQB.keyword()
       .onField("description")
       .matching("urban")
       .createQuery())
    .must(mythQB.range().onField("starred").above(4).createQuery())
    .createQuery();

//look for all myths except religious ones
Query luceneQuery = mythQB.all()
    .except(mythQB.keyword()
       .onField("description_stem")
       .matching("religion")
       .createQuery())
    .createQuery();
```

20.2.3.8. Query Options

The following is a summary of query options for query types and fields:

- **boostedTo** (on query type and on field) boosts the query or field to a provided factor.
- **withConstantScore** (on query) returns all results that match the query and have a constant score equal to the boost.
- **filteredBy(Filter)**(on query) filters query results using the **Filter** instance.
- **ignoreAnalyzer** (on field) ignores the analyzer when processing this field.
- **ignoreFieldBridge** (on field) ignores the field bridge when processing this field.

The following example illustrates how to use these options:

Querying Options

```
Query luceneQuery = mythQB
    .bool()
    .should(mythQB.keyword().onField("description").matching("urban").createQuery())
    .should(mythQB
       .keyword()
       .onField("name")
       .boostedTo(3)
       .ignoreAnalyzer()
       .matching("urban").createQuery())
    .must(mythQB
       .range()
       .boostedTo(5)
       .withConstantScore()
       .onField("starred")
       .above(4).createQuery())
    .createQuery();
```

20.2.4. Build a Query with Infinispan Query

20.2.4.1. Generality

After building the Lucene query, wrap it within a Infinispan CacheQuery. The query searches all indexed entities and returns all types of indexed classes unless explicitly configured not to do so.

Wrapping a Lucene Query in an Infinispan CacheQuery

```
CacheQuery cacheQuery = Search.getSearchManager(cache).getQuery(luceneQuery);
```

For improved performance, restrict the returned types as follows:

Filtering the Search Result by Entity Type

```
CacheQuery cacheQuery =
   Search.getSearchManager(cache).getQuery(luceneQuery, Customer.class);
// or
CacheQuery cacheQuery =
   Search.getSearchManager(cache).getQuery(luceneQuery, Item.class, Actor.class);
```

The first part of the second example only returns the matching **Customer** instances. The second part of the same example returns matching **Actor** and **Item** instances. The type restriction is polymorphic. As a result, if the two subclasses **Salesman** and **Customer** of the base class **Person** return, specify **Person.class** to filter based on result types.

20.2.4.2. Pagination

To avoid performance degradation, it is recommended to restrict the number of returned objects per query. A user navigating from one page to another page is a very common use case. The way to define pagination is similar to defining pagination in a plain HQL or Criteria query.

Defining pagination for a search query

```
CacheQuery cacheQuery = Search.getSearchManager(cache)
                        .getQuery(luceneQuery, Customer.class);
cacheQuery.firstResult(15); //start from the 15th element
cacheQuery.maxResults(10); //return 10 elements
```

NOTE

The total number of matching elements, despite the pagination, is accessible via **cacheQuery.getResultSize()**.

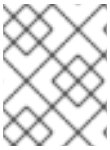
20.2.4.3. Sorting

Apache Lucene contains a flexible and powerful result sorting mechanism. The default sorting is by relevance and is appropriate for a large variety of use cases. The sorting mechanism can be changed to sort by other properties using the Lucene Sort object to apply a Lucene sorting strategy.

Specifying a Lucene Sort

```
org.infinispan.query.CacheQuery cacheQuery =
Search.getSearchManager(cache).getQuery(luceneQuery, Book.class);
org.apache.lucene.search.Sort sort = new Sort(
```

```
    new SortField("title", SortField.STRING_FIRST));
cacheQuery.sort(sort);
List results = cacheQuery.list();
```



NOTE

Fields used for sorting must not be tokenized. For more information about tokenizing, see @Field.

20.2.4.4. Projection

In some cases, only a small subset of the properties is required. Use Infinispan Query to return a subset of properties as follows:

Using Projection Instead of Returning the Full Domain Object

```
SearchManager searchManager = Search.getSearchManager(cache);
CacheQuery cacheQuery = searchManager.getQuery(luceneQuery, Book.class);
cacheQuery.projection("id", "summary", "body", "mainAuthor.name");
List results = cacheQuery.list();
Object[] firstResult = (Object[]) results.get(0);
Integer id = (Integer) firstResult[0];
String summary = (String) firstResult[1];
String body = (String) firstResult[2];
String authorName = (String) firstResult[3];
```

The Query Module extracts properties from the Lucene index and converts them to their object representation and returns a list of **Object[]**. Projections prevent a time consuming database round-trip. However, they have following constraints:

- The properties projected must be stored in the index (**@Field(store=Store.YES)**), which increases the index size.
- The properties projected must use a **FieldBridge** implementing **org.infinispan.query.bridge.TwoWayFieldBridge** or **org.infinispan.query.bridge.TwoWayStringBridge**, the latter being the simpler version.

  NOTE

  All Lucene-based Query API built-in types are two-way.

- Only the simple properties of the indexed entity or its embedded associations can be projected. Therefore a whole embedded entity cannot be projected.
- Projection does not work on collections or maps which are indexed via **@IndexedEmbedded**

Lucene provides metadata information about query results. Use projection constants to retrieve the metadata.

Using Projection to Retrieve Metadata

```
SearchManager searchManager = Search.getSearchManager(cache);
CacheQuery cacheQuery = searchManager.getQuery(luceneQuery, Book.class);
cacheQuery.projection("mainAuthor.name");
```

```
List results = cacheQuery.list();
Object[] firstResult = (Object[]) results.get(0);
float score = (Float) firstResult[0];
Book book = (Book) firstResult[1];
String authorName = (String) firstResult[2];
```

Fields can be mixed with the following projection constants:

- **FullTextQuery.THIS** returns the initialized and managed entity as a non-projected query does.
- **FullTextQuery.DOCUMENT** returns the Lucene Document related to the projected object.
- **FullTextQuery.OBJECT_CLASS** returns the indexed entity's class.
- **FullTextQuery.SCORE** returns the document score in the query. Use scores to compare one result against another for a given query. However, scores are not relevant to compare the results of two different queries.
- **FullTextQuery.ID** is the ID property value of the projected object.
- **FullTextQuery.DOCUMENT_ID** is the Lucene document ID. The Lucene document ID changes between two IndexReader openings.
- **FullTextQuery.EXPLANATION** returns the Lucene Explanation object for the matching object/document in the query. This is not suitable for retrieving large amounts of data. Running **FullTextQuery.EXPLANATION** is as expensive as running a Lucene query for each matching element. As a result, projection is recommended.

20.2.4.5. Limiting the Time of a Query

Limit the time a query takes in Infinispan Query as follows:

- Raise an exception when arriving at the limit.
- Limit to the number of results retrieved when the time limit is raised.

20.2.4.6. Raise an Exception on Time Limit

If a query uses more than the defined amount of time, a custom exception might be defined to be thrown.

To define the limit when using the CacheQuery API, use the following approach:

Defining a Timeout in Query Execution

```
SearchManagerImplementor searchManager = (SearchManagerImplementor)
Search.getSearchManager(cache);
searchManager.setTimeoutExceptionFactory(new MyTimeoutExceptionFactory());
CacheQuery cacheQuery = searchManager.getQuery(luceneQuery, Book.class);

//define the timeout in seconds
cacheQuery.timeout(2, TimeUnit.SECONDS);

try {
   cacheQuery.list();
}
```

```
catch (MyTimeoutException e) {
   //do something, too slow
}

private static class MyTimeoutExceptionFactory implements TimeoutExceptionFactory {
   @Override
   public RuntimeException createTimeoutException(String message, String queryDescription) {
      return new MyTimeoutException();
   }
}

public static class MyTimeoutException extends RuntimeException {
}
```

The **getResultSize()**, **iterate()** and **scroll()** honor the timeout until the end of the method call. As a result, **Iterable** or the **ScrollableResults** ignore the timeout. Additionally, **explain()** does not honor this timeout period. This method is used for debugging and to check the reasons for slow performance of a query.

IMPORTANT

The example code does not guarantee that the query stops at the specified results amount.

20.3. RETRIEVING THE RESULTS

20.3.1. Retrieving the Results

After building the Infinispan Query, it can be executed in the same way as a HQL or Criteria query. The same paradigm and object semantic apply to Lucene Query query and all the common operations like **list()**.

20.3.2. Performance Considerations

list() can be used to receive a reasonable number of results (for example when using pagination) and to work on them all. **list()** works best if the **batch-size** entity is correctly set up. If **list()** is used, the Query Module processes all Lucene Hits elements within the pagination.

20.3.3. Result Size

Some use cases require information about the total number of matching documents. Consider the following examples:

Retrieving all matching documents is costly in terms of resources. The Lucene-based Query API retrieves all matching documents regardless of pagination parameters. Since it is costly to retrieve all the matching documents, the Lucene-based Query API can retrieve the total number of matching documents regardless of the pagination parameters. All matching elements are retrieved without triggering any object loads.

Determining the Result Size of a Query

```
CacheQuery cacheQuery = Search.getSearchManager(cache).getQuery(luceneQuery,
         Book.class);
```

```
//return the number of matching books without loading a single one
assert 3245 == cacheQuery.getResultSize();

CacheQuery cacheQueryLimited =
      Search.getSearchManager(cache).getQuery(luceneQuery, Book.class);
cacheQuery.maxResults(10);
List results = cacheQuery.list();
assert 10 == results.size();
//return the total number of matching books regardless of pagination
assert 3245 == cacheQuery.getResultSize();
```

The number of results is an approximation if the index is not correctly synchronized with the database. An ansychronous cluster is an example of this scenario.

20.3.4. Understanding Results

Luke can be used to determine why a result appears (or does not appear) in the expected query result. The Query Module also offers the Lucene **Explanation** object for a given result (in a given query). This is an advanced class. Access the **Explanation** object as follows:

cacheQuery.explain(int) method

This method requires a document ID as a parameter and returns the **Explanation** object.

NOTE

In terms of resources, building an explanation object is as expensive as running the Lucene query. Do not build an explanation object unless it is necessary for the implementation.

20.4. FILTERS

20.4.1. Filters

Apache Lucene is able to filter query results according to a custom filtering process. This is a powerful way to apply additional data restrictions, especially since filters can be cached and reused. Applicable use cases include:

- security
- temporal data (example, view only last month's data)
- population filter (example, search limited to a given category)
- and many more

20.4.2. Defining and Implementing a Filter

The Lucene-based Query API includes transparent caches named filters which include parameters. The API is similar to the Hibernate Core filters:

Enabling Fulltext Filters for a Query

```
cacheQuery = Search.getSearchManager(cache).getQuery(query, Driver.class);
```

```
cacheQuery.enableFullTextFilter("bestDriver");
cacheQuery.enableFullTextFilter("security").setParameter("login", "andre");
cacheQuery.list(); //returns only best drivers where andre has credentials
```

In the provided example, two filters are enabled in the query. Enable or disable filters to customize the query.

Declare filters using the **@FullTextFilterDef** annotation. This annotation applies to **@Indexed** entities irrespective of the filter's query. Filter definitions are global therefore each filter must have a unique name. If two **@FullTextFilterDef** annotations with the same name are defined, a **SearchException** is thrown. Each named filter must specify its filter implementation.

Defining and Implementing a Filter

```
@FullTextFilterDefs({
   @FullTextFilterDef(name = "bestDriver", impl = BestDriversFilter.class),
   @FullTextFilterDef(name = "security", impl = SecurityFilterFactory.class)
})
public class Driver { ... }
```

```
public class BestDriversFilter extends org.apache.lucene.search.Filter {

   public DocIdSet getDocIdSet(IndexReader reader) throws IOException {
      OpenBitSet bitSet = new OpenBitSet(reader.maxDoc());
      TermDocs termDocs = reader.termDocs(new Term("score", "5"));
      while (termDocs.next()) {
         bitSet.set(termDocs.doc());
      }
      return bitSet;
   }
}
```

BestDriversFilter is a Lucene filter that reduces the result set to drivers where the score is **5**. In the example, the filter implements the **org.apache.lucene.search.Filter** directly and contains a no-arg constructor.

20.4.3. The @Factory Filter

Use the following factory pattern if the filter creation requires further steps, or if the filter does not have a no-arg constructor:

Creating a filter using the factory pattern

```
@FullTextFilterDef(name = "bestDriver", impl = BestDriversFilterFactory.class)
public class Driver { ... }

public class BestDriversFilterFactory {

   @Factory
   public Filter getFilter() {
      //some additional steps to cache the filter results per IndexReader
      Filter bestDriversFilter = new BestDriversFilter();
```

```
        return new CachingWrapperFilter(bestDriversFilter);
    }
}
```

The Lucene-based Query API uses a **@Factory** annotated method to build the filter instance. The factory must have a no argument constructor.

Named filters come in handy where parameters have to be passed to the filter. For example a security filter might want to know which security level you want to apply:

Passing parameters to a defined filter

```
cacheQuery = Search.getSearchManager(cache).getQuery(query, Driver.class);
cacheQuery.enableFullTextFilter("security").setParameter("level", 5);
```

Each parameter name should have an associated setter on either the filter or filter factory of the targeted named filter definition.

Using parameters in the actual filter implementation

```
public class SecurityFilterFactory {
    private Integer level;

    /**
     * injected parameter
     */
    public void setLevel(Integer level) {
        this.level = level;
    }

    @Key
    public FilterKey getKey() {
        StandardFilterKey key = new StandardFilterKey();
        key.addParameter(level);
        return key;
    }

    @Factory
    public Filter getFilter() {
        Query query = new TermQuery(new Term("level", level.toString()));
        return new CachingWrapperFilter(new QueryWrapperFilter(query));
    }
}
```

Note the method annotated **@Key** returns a **FilterKey** object. The returned object has a special contract: the key object must implement **equals()** / **hashCode()** so that two keys are equal if and only if the given **Filter** types are the same and the set of parameters are the same. In other words, two filter keys are equal if and only if the filters from which the keys are generated can be interchanged. The key object is used as a key in the cache mechanism.

20.4.4. Key Objects

@Key methods are needed only if:

- the filter caching system is enabled (enabled by default)

- the filter has parameters

The **StandardFilterKey** delegates the **equals()** / **hashCode()** implementation to each of the parameters equals and hashcode methods.

The defined filters are per default cached. The cache uses a combination of hard and soft references to allow disposal of memory when needed. The hard reference cache keeps track of the most recently used filters and transforms the ones least used to **SoftReferences** when needed. Once the limit of the hard reference cache is reached additional filters are cached as **SoftReferences**. To adjust the size of the hard reference cache, use **default.filter.cache_strategy.size** (defaults to 128). For advanced use of filter caching, you can implement your own **FilterCachingStrategy**. The classname is defined by **default.filter.cache_strategy**.

This filter caching mechanism should not be confused with caching the actual filter results. In Lucene it is common practice to wrap filters using the **IndexReader** around a **CachingWrapperFilter.** The wrapper will cache the **DocIdSet** returned from the **getDocIdSet(IndexReader reader)** method to avoid expensive recomputation. It is important to mention that the computed **DocIdSet** is only cachable for the same **IndexReader** instance, because the reader effectively represents the state of the index at the moment it was opened. The document list cannot change within an opened **IndexReader**. A different/new**IndexReader** instance, however, works potentially on a different set of **Document**s (either from a different index or simply because the index has changed), hence the cached **DocIdSet** has to be recomputed.

20.4.5. Full Text Filter

The Lucene-based Query API uses the **cache** flag of **@FullTextFilterDef**, set to **FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS** which automatically caches the filter instance and wraps the filter around a Hibernate specific implementation of **CachingWrapperFilter**. Unlike Lucene's version of this class, **SoftReference**s are used with a hard reference count (see discussion about filter cache). The hard reference count is adjusted using **default.filter.cache_docidresults.size** (defaults to **5**). Wrapping is controlled using the **@FullTextFilterDef.cache** parameter. There are three different values for this parameter:

Value	Definition
FilterCacheModeType.NONE	No filter instance and no result is cached by Hibernate Search. For every filter call, a new filter instance is created. This setting might be useful for rapidly changing data sets or heavily memory constrained environments.
FilterCacheModeType.INSTANCE_ONLY	The filter instance is cached and reused across concurrent **Filter.getDocIdSet()** calls. **DocIdSet** results are not cached. This setting is useful when a filter uses its own specific caching mechanism or the filter results change dynamically due to application specific events making **DocIdSet** caching in both cases unnecessary.
FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS	Both the filter instance and the **DocIdSet** results are cached. This is the default value.

Filters should be cached in the following situations:

- The system does not update the targeted entity index often (in other words, the IndexReader is reused a lot).
- The Filter's DocIdSet is expensive to compute (compared to the time spent to execute the query).

20.4.6. Using Filters in a Sharded Environment

Execute queries on a subset of the available shards in a sharded environment as follows:

1. Create a sharding strategy to select a subset of **IndexManager**s depending on filter configurations.
2. Activate the filter when running the query.

The following is an example of sharding strategy that queries a specific shard if the customer filter is activated:

Querying a Specific Shard

```
public class CustomerShardingStrategy implements IndexShardingStrategy {

   // stored IndexManagers in a array indexed by customerID
   private IndexManager[] indexManagers;

   public void initialize(Properties properties, IndexManager[] indexManagers) {
      this.indexManagers = indexManagers;
   }

   public IndexManager[] getIndexManagersForAllShards() {
      return indexManagers;
   }

   public IndexManager getIndexManagerForAddition(
      Class<?> entity, Serializable id, String idInString, Document document) {
      Integer customerID = Integer.parseInt(document.getFieldable("customerID")
                                            .stringValue());
      return indexManagers[customerID];
   }

   public IndexManager[] getIndexManagersForDeletion(
      Class<?> entity, Serializable id, String idInString) {
      return getIndexManagersForAllShards();
   }

   /**
    * Optimization; don't search ALL shards and union the results; in this case, we
    * can be certain that all the data for a particular customer Filter is in a single
    * shard; return that shard by customerID.
    */
   public IndexManager[] getIndexManagersForQuery(
      FullTextFilterImplementor[] filters) {
      FullTextFilter filter = getCustomerFilter(filters, "customer");
      if (filter == null) {
         return getIndexManagersForAllShards();
      }
```

```
        else {
            return new IndexManager[] { indexManagers[Integer.parseInt(
                filter.getParameter("customerID").toString())] };
        }
    }

    private FullTextFilter getCustomerFilter(FullTextFilterImplementor[] filters,
                                             String name) {
        for (FullTextFilterImplementor filter: filters) {
            if (filter.getName().equals(name)) return filter;
        }
        return null;
    }
}
```

If the **customer** filter is present in the example, the query only uses the shard dedicated to the customer. The query returns all shards if the **customer** filter is not found. The sharding strategy reacts to each filter depending on the provided parameters.

Activate the filter when the query must be run. The filter is a regular filter (as defined in Filters), which filters Lucene results after the query. As an alternate, use a special filter that is passed to the sharding strategy and then ignored for duration of the query. Use the **ShardSensitiveOnlyFilter** class to declare the filter.

Using the ShardSensitiveOnlyFilter Class

```
@Indexed
@FullTextFilterDef(name = "customer", impl = ShardSensitiveOnlyFilter.class)
public class Customer {
   ...
}

CacheQuery cacheQuery = Search.getSearchManager(cache).getQuery(query,
   Customer.class);
cacheQuery.enableFullTextFilter("customer").setParameter("CustomerID", 5);
@SuppressWarnings("unchecked")
List results = cacheQuery.list();
```

If the **ShardSensitiveOnlyFilter** filter is used, Lucene filters do not need to be implemented. Use filters and sharding strategies reacting to these filters for faster query execution in a sharded environment.

20.5. CONTINUOUS QUERIES

20.5.1. Continuous Query

Continuous Querying allows an application to receive the entries that currently match a query, and be continuously notified of any changes to the queried data set. This includes both incoming matches, for values that have joined the set, and outgoing matches, for values that have left the set, that resulted from further cache operations. By using a Continuous Query the application receives a steady stream of events instead of repeatedly executing the same query to look for changes, resulting in a more efficient use of resources.

For instance, all of the following use cases could utilize Continuous Queries:

1. Return all persons with an age between 18 and 25 (assuming the **Person** entity has an **age** property and is updated by the user application).
2. Return all transactions higher than $2000.
3. Return all times where the lap speed of F1 racers were less than 1:45.00s (assuming the cache contains **Lap** entries and that laps are entered live during the race).

20.5.2. Continuous Query Evaluation

A Continuous Query uses a listener that receives a notification when:

- An entry starts matching the specified query, represented by a **Join** event.
- An entry stops matching the specified query, represented by a **Leave** event.

When a client registers a Continuous Query Listener it immediately begins to receive the results currently matching the query, received as **Join** events as described above. In addition, it will receive subsequent notifications when other entries begin matching the query, as **Join** events, or stop matching the query, as **Leave** events, as a consequence of any cache operations that would normally generate creation, modification, removal, or expiration events.

To determine if the listener receives a **Join** or **Leave** event the following logic is used:

1. If the query on both the old and new values evaluate false, then the event is suppressed.
2. If the query on both the old and new values evaluate true, then the event is suppressed.
3. If the query on the old value evaluates false and the query on the new value evaluates true, then a **Join** event is sent.
4. If the query on the old value evaluates true and the query on the new value evaluates false, then a **Leave** event is sent.
5. If the query on the old value evaluates true and the entry is removed, then a **Leave** event is sent.

NOTE

Continuous Queries cannot use grouping, aggregation, or sorting operations.

20.5.3. Using Continuous Queries

The following instructions apply to both Library and Remote Client-Server modes.

Adding Continuous Queries

To create a Continuous Query the **Query** object will be created similar to other querying methods; however, ensure that the **Query** is registered with a **org.infinispan.query.api.continuous.ContinuousQuery** and a **org.infinispan.query.api.continuous.ContinuousQueryListener** is in use.

The **ContinuousQuery** object associated to a cache can be obtained by calling the static method **org.infinispan.client.hotrod.Search.getContinuousQuery(RemoteCache<K, V> cache)** if running in Client-Server mode or **org.infinispan.query.Search.getContinuousQuery(Cache<K, V> cache)** when running in Library mode.

Once the **ContinuousQueryListener** has been defined it may be added by using the **addContinuousQueryListener** method of **ContinuousQuery**:

```
continuousQuery.addContinuousQueryListener(query, listener)
```

The following example demonstrates a simple method of implementing and adding a Continuous Query in Library mode:

Defining and Adding a Continuous Query

```
import org.infinispan.query.api.continuous.ContinuousQuery;
import org.infinispan.query.api.continuous.ContinuousQueryListener;
import org.infinispan.query.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

[...]

// To begin we create a ContinuousQuery instance on the cache
ContinuousQuery<Integer, Person> continuousQuery = Search.getContinuousQuery(cache);

// Define our query. In this case we will be looking for any
// Person instances under 21 years of age.
QueryFactory queryFactory = Search.getQueryFactory(cache);
Query query = queryFactory.from(Person.class)
    .having("age").lt(21)
    .build();

final Map<Integer, Person> matches = new ConcurrentHashMap<Integer, Person>();

// Define the ContinuousQueryListener
ContinuousQueryListener<Integer, Person> listener = new ContinuousQueryListener<Integer,
Person>() {
    @Override
    public void resultJoining(Integer key, Person value) {
        matches.put(key, value);
    }

    @Override
    public void resultLeaving(Integer key) {
        matches.remove(key);
    }
};

// Add the listener and generated query
continuousQuery.addContinuousQueryListener(query, listener);

[...]

// Remove the listener to stop receiving notifications
continuousQuery.removeContinuousQueryListener(listener);
```

As **Person** instances are added to the cache that contain an **Age** less than 21 they will be placed into **matches**, and when these entries are removed from the cache they will be also be removed from **matches**.

Removing Continuous Queries

To stop the query from further execution remove the listener:

```
continuousQuery.removeContinuousQueryListener(listener);
```

20.5.4. C++ and C# Continuous Queries

In addition to native Java based continuous queries, JBoss Data Grid also supports C++ and C# based continuous queries.

20.5.4.1. C++ Continous Queries

C++ continuous queries can be setup using the following code:

C++ Continuous Query setup

```
ContinuousQueryListener<int, sample_bank_account::User> cql(testCache,"select id from
sample_bank_account.User");
std::function<void(int, sample_bank_account::User)> join = [](int k, sample_bank_account::User u) {
    std::cout << "JOINING: key="<< u.id() << " value="<< u.name() << std::endl;
};
std::function<void(int, sample_bank_account::User)> leave =[](int k, sample_bank_account::User u) {
    std::cout << "LEAVING: key="<< u.id() << " value="<< u.name() << std::endl;
};
std::function<void(int, sample_bank_account::User)> change =[](int k, sample_bank_account::User
u) {
    std::cout << "CHANGING: key="<< u.id() << " value="<< u.name() << std::endl;
};

cql.setJoiningListener(join);
cql.setLeavingListener(leave);
cql.setUpdatedListener(change);
testCache.addContinuousQueryListener(cql);
```

C++ continuous queries can be removed using the following code:

C++ Continuous Query Removal

```
testCache.addContinuousQueryListener(cql);

[...]

// Remove the listener to stop receiving notifications
testCache.removeContinuousQueryListener(cql);
```

20.5.4.2. C# Continuous Queries

C# continuous queries can be setup using the following code:

C# Continuous Query setup

```
qr.QueryString = "from sample_bank_account.User";

Event.ContinuousQueryListener<int, User> cql = new Event.ContinuousQueryListener<int, User>
(qr.QueryString);
cql.JoiningCallback = (int k, User v) => { Console.WriteLine("JOINING: " + k + ", " + v); s.Release(); };
cql.LeavingCallback = (int k, User v) => { Console.WriteLine("LEAVING: " + k + ", " + v); };
cql.UpdatedCallback = (int k, User v) => { Console.WriteLine("UPDATED: " + k + ", " + v); };
userCache.AddContinuousQueryListener(cql);
```

C# continuous queries can be removed using the following code:

C# Continuous Query Removal

```
userCache.AddContinuousQueryListener(cql);

[...]

// Remove the listener to stop receiving notifications
userCache.RemoveContinuousQueryListener(cql);
```

20.5.5. Performance Considerations with Continuous Queries

Continuous Queries are designed to constantly keep any applications updated where it is implemented, potentially resulting in a large number of events generated for particularly broad queries. In addition, a new memory allocation is made for each event. This behavior may result in memory pressure, including potential errors, if queries are not carefully designed.

To prevent these issues it is strongly recommended to ensure that each query captures only the information needed, and that each **ContinuousQueryListener** is designed to quickly process all received events.

20.6. BROADCAST QUERIES

20.6.1. Broadcast Queries

The broadcast query feature allows each node to index its own data during writes, and at query time, it sends, or "broadcasts", the query to each node. The results from each node are then combined before being returned to the caller. This is ideal for **DIST** caches with large indices since the amount of data transferred is the query itself and the results.

20.6.1.1. Using Broadcast Queries

To use broadcast queries include **IndexedQueryMode.BROADCAST** as an argument to your query. An example of this is shown below:

```
CacheQuery<Person> broadcastQuery = Search.getSearchManager(cache).getQuery(new
MatchAllDocsQuery(), IndexedQueryMode.BROADCAST);

List<Person> result = broadcastQuery.list();
```

CHAPTER 21. THE INFINISPAN QUERY DSL

21.1. THE INFINISPAN QUERY DSL

The Infinispan Query DSL provides an unified way of querying a cache. It can be used in Library mode for both indexed and indexless queries, as well as for Remote Querying (via the Hot Rod Java client). The Infinispan Query DSL allows queries without relying on Lucene native query API or Hibernate Search query API.

Indexless queries are only available with the Infinispan Query DSL for both the JBoss Data Grid remote and embedded mode. Indexless queries do not require a configured index (see Enabling Infinispan Query DSL-based Queries). The Hibernate Search/Lucene-based API cannot use indexless queries.

21.2. CREATING QUERIES WITH INFINISPAN QUERY DSL

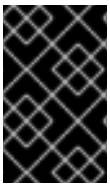
The new query API is located in the org.infinispan.query.dsl package. A query is created with the assistance of the **QueryFactory** instance, which is obtained using **Search.getQueryFactory()**. Each **QueryFactory** instance is bound to the one cache instance, and is a stateless and thread-safe object that can be used for creating multiple parallel queries.

The Infinispan Query DSL uses the following steps to perform a query.

1. A query is created by invocating the **from(Class entityType)** method, which returns a **QueryBuilder** object that is responsible for creating queries for the specified entity class from the given cache.

2. The **QueryBuilder** accumulates search criteria and configuration specified through invoking its DSL methods, and is used to build a **Query** object by invoking the **QueryBuilder.build()** method, which completes the construction. The **QueryBuilder** object cannot be used for constructing multiple queries at the same time except for nested queries, however it can be reused afterwards.

3. Invoke the **list()** method of the **Query** object to execute the query and fetch the results. Once executed, the **Query** object is not reusable. If new results must be fetched, a new instance must be obtained by calling **QueryBuilder.build()**.



IMPORTANT

A query targets a single entity type and is evaluated over the contents of a single cache. Running a query over multiple caches, or creating queries targeting several entity types is not supported.

21.3. ENABLING INFINISPAN QUERY DSL-BASED QUERIES

In library mode, running Infinispan Query DSL-based queries is almost identical to running Lucene-based API queries. Prerequisites are:

- All libraries required for Infinispan Query on the classpath. Refer to the Administration and Configuration Guide for details.

- Indexing enabled and configured for caches (optional). Refer to the Administration and Configuration Guide for details.

- Annotated POJO cache values (optional). If indexing is not enabled, POJO annotations are also not required and are ignored if set. If indexing is not enabled, all fields that follow JavaBeans conventions are searchable instead of only the fields with Hibernate Search annotations.

21.4. RUNNING INFINISPAN QUERY DSL-BASED QUERIES

Once Infinispan Query DSL-based queries have been enabled, obtain a **QueryFactory** from the **Search** in order to run a DSL-based query.

Obtain a QueryFactory for a Cache

In Library mode, obtain a **QueryFactory** as follows:

```
QueryFactory qf = org.infinispan.query.Search.getQueryFactory(cache)
```

Constructing a DSL-based Query

```
import org.infinispan.query.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

QueryFactory qf = Search.getQueryFactory(cache);
Query q = qf.from(User.class)
   .having("name").eq("John")
   .build();
List list = q.list();
assertEquals(1, list.size());
assertEquals("John", list.get(0).getName());
assertEquals("Doe", list.get(0).getSurname());
```

When using Remote Querying in Remote Client-Server mode, the **Search** object resides in package **org.infinispan.client.hotrod**. See the example in Performing Remote Queries via the Hot Rod Java Client for details.

It is also possible to combine multiple conditions with boolean operators, including sub-conditions. For example:

Combining Multiple Conditions

```
Query q = qf.from(User.class)
   .having("name").eq("John")
   .and().having("surname").eq("Doe")
   .and().not(qf.having("address.street").like("%Tanzania%")
   .or().having("address.postCode").in("TZ13", "TZ22"))
   .build();
```

This query API simplifies the way queries are written by not exposing the user to the low level details of constructing Lucene query objects. It also has the benefit of being available to remote Hot Rod clients.

The following example shows how to write a query for the **Book** entity.

Querying the Book Entity

```
import org.infinispan.query.Search;
```

```
import org.infinispan.query.dsl.*;

// get the DSL query factory, to be used for constructing the Query object:
QueryFactory qf = Search.getQueryFactory(cache);
// create a query for all the books that have a title which contains the word "engine":
Query query = qf.from(Book.class)
    .having("title").like("%engine%")
    .build();
// get the results
List<Book> list = query.list();
```

21.5. PROJECTION QUERIES

In many cases returning the full domain object is unnecessary, and only a small subset of attributes are desired by the application. Projection Queries allow a specific subset of attributes (or attribute paths) to be returned. If a projection query is used then the **Query.list()** will not return the whole domain entity (**List<Object>**), but instead will return a **List<Object[]>**, with each entry in the array corresponding to a projected attribute.

To define a projection query use the **select(...)** method when building the query, as seen in the following example:

Retrieving title and publication year

```
// Match all books that have the word "engine" in their title or description
// and return only their title and publication year.
Query query = queryFactory.from(Book.class)
    .select(Expression.property("title"), Expression.property("publicationYear"))
    .having("title").like("%engine%")
    .or().having("description").like("%engine%")
    .build();

// results.get(0)[0] contains the first matching entry's title
// results.get(0)[1] contains the first matching entry's publication year
List<Object[]> results = query.list();
```

21.6. GROUPING AND AGGREGATION OPERATIONS

The Infinispan Query DSL has the ability to group query results according to a set of grouping fields and construct aggregations of the results from each group by applying an aggregation function to the set of values. Grouping and aggregation can only be used with projection queries.

The set of grouping fields is specified by calling the method **groupBy(field)** multiple times. The order of grouping fields is not relevant.

All non-grouping fields selected in the projection must be aggregated using one of the grouping functions described below.

Grouping Books by author and counting them

```
Query query = queryFactory.from(Book.class)
    .select(Expression.property("author"), Expression.count("title"))
    .having("title").like("%engine%")
    .groupBy("author")
```

```
    .build();

// results.get(0)[0] will contain the first matching entry's author
// results.get(0)[1] will contain the first matching entry's title
List<Object[]> results = query.list();
```

Aggregation Operations

The following aggregation operations may be performed on a given field:

- **avg()** - Computes the average of a set of **Number**s, represented as a **Double**. If there are no non-null values the result is null instead.
- **count()** - Returns the number of non-null rows as a **Long**. If there are no non-null values the result is 0 instead.
- **max()** - Returns the greatest value found in the specified field, with a return type equal to the field in which it was applied. If there are no non-null values the result is null instead.

NOTE

Values in the given field must be of type **Comparable**, otherwise an **IllegalStateException** will be thrown.

- **min()** - Returns the smallest value found in the specified field, with a return type equal to the field in which it was applied. If there are no non-null values the result is null instead.

NOTE

Values in the given field must be of type **Comparable**, otherwise an **IllegalStateException** will be thrown.

- **sum()** - Computes and returns the sum of a set of **Number**s, with a return type dependent on the indicated field's type. If there are no non-null values the result is null instead.
  The following table indicates the return type based on the specified field.

Table 21.1. Sum Return Type

Field Type	Return Type
Integral (other than **BigInteger**)	**Long**
Floating Point	**Double**
BigInteger	**BigInteger**
BigDecimal	**BigDecimal**

Projection Query Special Cases

The following cases items describe special use cases with projection queries:

- A projection query in which all selected fields are aggregated and none is used for grouping is legal. In this case the aggregations will be computed globally instead of being computed per each group.
- A grouping field can be used in an aggregation. This is a degenerated case in which the aggregation will be computed over a single data point, the value belonging to current group.
- A query that selects only grouping fields but no aggregation fields is legal.

Evaluation of grouping and aggregation queries

Aggregation queries can include filtering conditions, like usual queries, which may be optionally performed before and after the grouping operation.

All filter conditions specified before invoking the **groupBy** method will be applied directly to the cache entries before the grouping operation is performed. These filter conditions may refer to any properties of the queried entity type, and are meant to restrict the data set that is going to be later used for grouping.

All filter conditions specified after invoking the **groupBy** method will be applied to the projection that results from the grouping operation. These filter conditions can either reference any of the fields specified by **groupBy** or aggregated fields. Referencing aggregated fields that are not specified in the **select** clause is allowed; however, referencing non-aggregated and non-grouping fields is forbidden. Filtering in this phase will reduce the amount of groups based on their properties.

Ordering may also be specified similar to usual queries. The ordering operation is performed after the grouping operation and can reference any fields that are allowed for post-grouping filtering as described earlier.

21.7. USING NAMED PARAMETERS

Instead of creating a new query for every request it is possible to include parameters in the query which may be replaced with each execution. This allows a query to be defined a single time and adjust variables in the query as needed.

Parameters are defined when the query is created by using the **Expression.param(...)** operator on the right hand side of any comparison operator from the **having(...)**:

Defining Named Parameters

```
import org.infinispan.query.Search;
import org.infinispan.query.dsl.*;
[...]

QueryFactory queryFactory = Search.getQueryFactory(cache);
// Defining a query to search for various authors
Query query = queryFactory.from(Book.class)
    .select("title")
    .having("author").eq(Expression.param("authorName"))
    .build()
[...]
```

Setting the values of Named Parameters

By default all declared parameters are null, and all defined parameters must be updated to non-null values before the query must be executed. Once the parameters have been declared they may then be

updated by invoking either **setParameter(parameterName, value)** or **setParameters(parameterMap)** on the query with the new values; in addition, the query does not need to be rebuilt. It may be executed again after the new parameters have been defined.

Updating Parameters Individually

```
[...]
query.setParameter("authorName","Smith");

// Rerun the query and update the results
resultList = query.list();
[...]
```

Updating Parameters as a Map

```
[...]
parameterMap.put("authorName","Smith");

query.setParameters(parameterMap);

// Rerun the query and update the results
resultList = query.list();
[...]
```

CHAPTER 22. BUILDING A QUERY USING THE ICKLE QUERY LANGUAGE

22.1. BUILDING A QUERY USING THE ICKLE QUERY LANGUAGE

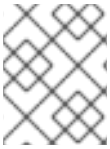
Create relational and full-text queries in both Library and Remote Client-Server mode with the Ickle query language.

Ickle is string-based and has the following characteristics:

- Queres Java classes and supports Protocol Buffers.
- Queries can target a single entity type.
- Queries can filter on properties of embedded objects, including collections.
- Supports projections, aggregations, sorting, named parameters.
- Supports indexed and non-indexed execution.
- Supports complex boolean expressions.
- Supports full-text queries.
- Does not support computations in expressions, such as **user.age > sqrt(user.shoeSize+3)**.
- Does not support joins.
- Does not support subqueries.
- Is supported across various JBoss Data Grid APIs. Whenever a Query is produced by the **QueryBuilder** is accepted, including continuous queries or in event filters for listeners.

To use the API, first obtain a **QueryFactory** to the cache and then call the **.create()** method, passing in the string to use in the query. For instance:

```
QueryFactory qf = Search.getQueryFactory(remoteCache);
Query q = qf.create("from sample_bank_account.Transaction where amount > 20");
```



NOTE

When using Ickle all fields used with full-text operators must be both Indexed and Analysed.

22.2. ICKLE QUERY LANGUAGE PARSER SYNTAX

The parser syntax for the Ickle query language has some notable rules:

- Whitespace is not significant.
- Wildcards are not supported in field names.
- A field name or path must always be specified, as there is no default field.

- **&&** and || are accepted instead of **AND** or **OR** in both full-text and JPA predicates.
- **!** may be used instead of **NOT**.
- A missing boolean operator is interpreted as **OR**.
- String terms must be enclosed with either single or double quotes.
- Fuzziness and boosting are not accepted in arbitrary order; fuzziness always comes first.
- **!=** is accepted instead of **<>**.
- Boosting cannot be applied to **>**,**>=**,**<**,**⇐** operators. Ranges may be used to achieve the same result.

22.3. FUZZY QUERIES

To execute a fuzzy query add ~ along with an integer, representing the distance from the term used, after the term. For instance

```
Query fuzzyQuery = qf.create("from sample_bank_account.Transaction where description : 'cofee'~2");
```

22.4. RANGE QUERIES

To execute a range query define the given boundaries within a pair of braces, as seen in the following example:

```
Query rangeQuery = qf.create("from sample_bank_account.Transaction where amount : [20 to 50]");
```

22.5. PHRASE QUERIES

A group of words may be searched by surrounding them in quotation marks, as seen in the following example:

```
Query q = qf.create("from sample_bank_account.Transaction where description : 'bus fare'");
```

22.6. PROXIMITY QUERIES

To execute a proximity query, finding two terms within a specific distance, add a ~ along with the distance after the phrase. For instance, the following example will find the words **canceling** and **fee** provided they are not more than 3 words apart:

```
Query proximityQuery = qf.create("from sample_bank_account.Transaction where description : 'canceling fee'~3 ");
```

22.7. WILDCARD QUERIES

Both single-character and multi-character wildcard searches may be performed:

- A single-character wildcard search may be used with the **?** character.

- A multi-character wildcard search may be used with the ***** character.

To search for **text** or **test** the following single-character wildcard search would be used:

```
Query wildcardQuery = qf.create("from sample_bank_account.Transaction where description : 'te?t'");
```

To search for **test**, **tests**, or **tester** the following multi-character wildcard search would be used:

```
Query wildcardQuery = qf.create("from sample_bank_account.Transaction where description :
'test*'");
```

NOTE

Full-text wildcard queries match terms as they are stored in the index, which varies depending on the analyzer you use.

You should also be aware that JBoss Data Grid does not analyze arguments in wildcard operators. Use arguments that resemble the output of the analysis for the index.

For example, most analyzers converts text to lowercase before indexing it. In this case, any wildcard searches that use arguments with capital letters return no matches.

In general, wildcard queries are also slower than other types of full-text queries and should be avoided whenever possible.

22.8. REGULAR EXPRESSION QUERIES

Regular expression queries may be performed by specifing a pattern between **/**. Ickle uses Lucene's regular expression syntax, so to search for the words **moat** or **boat** the following could be used:

```
Query regExpQuery = qf.create("from sample_library.Book  where title : /[mb]oat/");
```

22.9. BOOSTING QUERIES

Terms may be boosted by adding a **^** after the term to increase their relevance in a given query, the higher the boost factor the more relevant the term will be. For instance to search for titles containing **beer** and **wine** with a higher relevance on **beer**, by a factor of 3, the following could be used:

```
Query boostedQuery = qf.create("from sample_library.Book where title : beer^3 OR wine");
```

CHAPTER 23. REMOTE QUERYING

23.1. REMOTE QUERYING

Red Hat JBoss Data Grid's Hot Rod protocol allows remote, language neutral querying, using either the Infinispan Query Domain-specific Language (DSL) or the Ickle query language. Querying in either method allows remote, language-neutral querying, and is implementable in all languages currently available for the Hot Rod client.

The Infinispan Query Domain-specific Language

JBoss Data Grid uses its own query language based on an internal DSL. The Infinispan Query DSL provides a simplified way of writing queries, and is agnostic of the underlying query mechanisms. Additional information on the Infinispan Query DSL is available at The Infinispan Query DSL .

Ickle

Ickle is a string based query language allowing full-text and relational searches. Additional information on Ickle is available at Constructing Ickle Queries.

Protobuf Encoding

Google's Protocol Buffers is used as an encoding format for both storing and querying data. The Infinispan Query DSL can be used remotely via the Hot Rod client that is configured to use the Protobuf marshaller. Protocol Buffers are used to adopt a common format for storing cache entries and marshalling them. Remote clients that need to index and query their stored entities must use the Protobuf encoding format. It is also possible to store Protobuf entities for the benefit of platform independence without indexing enabled if it is not required.

23.2. QUERYING COMPARISON

In Library mode, both Lucene Query-based and DSL querying is available. In Remote Client-Server mode, only Remote Querying using DSL is available. The following table is a feature comparison between Lucene Query-based querying, Infinispan Query DSL and Remote Querying.

Table 23.1. Embedded querying and Remote querying

Feature	Library Mode/Lucene Query	Library Mode/DSL Query	Remote Client-Server Mode/DSL Query	Library Mode/Ickle Query	Remote Client-Server Mode/Ickle Query
Indexing	Mandatory	Optional but highly recommended	Optional but highly recommended	Optional but highly recommended	Optional but highly recommended
Index contents	Selected fields	Selected fields	Selected fields	Selected fields	Selected fields
Data Storage Format	Java objects	Java objects	Protocol buffers	Java objects	Protocol buffers
Keyword Queries	Yes	No	No	Yes	Yes

Feature	Library Mode/Lucene Query	Library Mode/DSL Query	Remote Client-Server Mode/DSL Query	Library Mode/Ickle Query	Remote Client-Server Mode/Ickle Query
Range Queries	Yes	Yes	Yes	Yes	Yes
Fuzzy Queries	Yes	No	No	Yes	Yes
Wildcard	Yes	Limited to like queries (Matches a wildcard pattern that follows JPA rules).	Limited to like queries (Matches a wildcard pattern that follows JPA rules).	Yes	Yes
Phrase Queries	Yes	No	No	Yes	Yes
Combining Queries	AND, OR, NOT, SHOULD	AND, OR, NOT	AND, OR, NOT	AND, OR, NOT	AND, OR, NOT
Sorting Results	Yes	Yes	Yes	Yes	Yes
Filtering Results	Yes, both within the query and as appended operator	Within the query	Within the query	Within the query	Within the query
Pagination of Results	Yes	Yes	Yes	Yes	Yes
Continuous Queries	No	Yes	Yes	No	No
Query Aggregation Operations	No	Yes	Yes	Yes	Yes

23.3. PERFORMING REMOTE QUERIES VIA THE HOT ROD JAVA CLIENT

Remote querying over Hot Rod can be enabled once the **RemoteCacheManager** has been configured with the Protobuf marshaller.

The following procedure describes how to enable remote querying over its caches.

Prerequisites

RemoteCacheManager must be configured to use the Protobuf Marshaller.

Enabling Remote Querying via Hot Rod

1. Add the infinispan-remote.jar
   The *infinispan-remote.jar* is an uberjar, and therefore no other dependencies are required for this feature.

2. Enable indexing on the cache configuration
   Indexing is not mandatory for Remote Queries, but it is highly recommended because it makes searches on caches that contain large amounts of data significantly faster. Indexing can be configured at any time. Enabling and configuring indexing is the same as for Library mode.

   Add the following configuration within the **cache-container** element loated inside the Infinispan subsystem element.

```
<!-- A basic example of an indexed local cache
   that uses the RAM Lucene directory provider -->
<local-cache name="an-indexed-cache">
   <!-- Enable indexing using the RAM Lucene directory provider -->
   <indexing index="ALL">
      <property name="default.directory_provider">ram</property>
   </indexing>
</local-cache>
```

3. Register the Protobuf schema definition files
   Register the Protobuf schema definition files by adding them in the **___protobuf_metadata** system cache. The cache key is a string that denotes the file name and the value is *.proto* file, as a string. Alternatively, protobuf schemas can also be registered by invoking the **registerProtofile** methods of the server's **ProtobufMetadataManager** MBean. There is one instance of this MBean per cache container and is backed by the **___protobuf_metadata**, so that the two approaches are equivalent.

   For an example of providing the protobuf schema via **___protobuf_metadata** system cache, see Registering a Protocol Buffers schema file.

NOTE

Writing to the **___protobuf_metadata** cache requires the **___schema_manager** role be added to the user performing the write.

The following example demonstrates how to invoke the **registerProtofile** methods of the **ProtobufMetadataManager** MBean.

Registering Protobuf schema definition files via JMX

```
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXServiceURL;

...

String serverHost = ...      // The address of your JDG server
```

```
int serverJmxPort = ...        // The JMX port of your server
String cacheContainerName = ... // The name of your cache container
String schemaFileName = ...     // The name of the schema file
String schemaFileContents = ... // The Protobuf schema file contents

JMXConnector jmxConnector = JMXConnectorFactory.connect(new JMXServiceURL(
   "service:jmx:remoting-jmx://" + serverHost + ":" + serverJmxPort));
MBeanServerConnection jmxConnection = jmxConnector.getMBeanServerConnection();

ObjectName protobufMetadataManagerObjName =
   new ObjectName("jboss.infinispan:type=RemoteQuery,name=" +
   ObjectName.quote(cacheContainerName) +
   ",component=ProtobufMetadataManager");

jmxConnection.invoke(protobufMetadataManagerObjName,
                     "registerProtofile",
                     new Object[]{schemaFileName, schemaFileContents},
                     new String[]{String.class.getName(), String.class.getName()});
jmxConnector.close();
```

Result

All data placed in the cache is immediately searchable, whether or not indexing is in use. Entries do not need to be annotated, unlike embedded queries. The entity classes are only meaningful to the Java client and do not exist on the server.

Once remote querying has been enabled, the **QueryFactory** can be obtained using the following:

Obtaining the QueryFactory

```
import org.infinispan.client.hotrod.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;
import org.infinispan.query.dsl.SortOrder;
...
remoteCache.put(2, new User("John", 33));
remoteCache.put(3, new User("Alfred", 40));
remoteCache.put(4, new User("Jack", 56));
remoteCache.put(4, new User("Jerry", 20));

QueryFactory qf = Search.getQueryFactory(remoteCache);
Query query = qf.from(User.class)
   .orderBy("age", SortOrder.ASC)
   .having("name").like("J%")
   .and().having("age").gte(33)
   .build();

List<User> list = query.list();
assertEquals(2, list.size());
assertEquals("John", list.get(0).getName());
assertEquals(33, list.get(0).getAge());
assertEquals("Jack", list.get(1).getName());
assertEquals(56, list.get(1).getAge());
```

Queries can now be run over Hot Rod similar to Library mode.

23.4. REMOTE QUERYING IN THE HOT ROD C++ CLIENT

For instructions on using remote querying in the Hot Rod C++ Client refer to Performing Remote Queries in the Hot Rod C++ Client.

23.5. REMOTE QUERYING IN THE HOT ROD C# CLIENT

For instructions on using remote querying in the Hot Rod C# Client refer to Performing Remote Queries in the Hot Rod C# Client.

23.6. PROTOBUF ENCODING

23.6.1. Protobuf Encoding

The Infinispan Query DSL can be used remotely via the Hot Rod client. In order to do this, protocol buffers are used to adopt a common format for storing cache entries and marshalling them.

For more information, see https://developers.google.com/protocol-buffers/docs/overview

23.6.2. Storing Protobuf Encoded Entities

Protobuf requires data to be structured. This is achieved by declaring Protocol Buffer message types in *.proto* files

For example:

.library.proto

```
package book_sample;
message Book {
   required string title = 1;
   required string description = 2;
   required int32 publicationYear = 3; // no native Date type available in Protobuf

   repeated Author authors = 4;
}
message Author {
   required string name = 1;
   required string surname = 2;
}
```

The provided example:

1. An entity named **Book** is placed in a package named **book_sample**.

   ```
   package book_sample;
   message Book {
   ```

2. The entity declares several fields of primitive types and a repeatable field named **authors**.

   ```
      required string title = 1;
      required string description = 2;
      required int32 publicationYear = 3; // no native Date type available in Protobuf
   ```

```
    repeated Author authors = 4;
}
```

3. The **Author** message instances are embedded in the **Book** message instance.

```
message Author {
    required string name = 1;
    required string surname = 2;
}
```

23.6.3. About Protobuf Messages

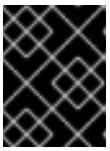
There are a few important things to note about Protobuf messages:

- Nesting of messages is possible, however the resulting structure is strictly a tree, and never a graph.
- There is no type inheritance.
- Collections are not supported, however arrays can be easily emulated using repeated fields.

23.6.4. Using Protobuf with Hot Rod

Protobuf can be used with JBoss Data Grid's Hot Rod using the following two steps:

1. Configure the client to use a dedicated marshaller, in this case, the **ProtoStreamMarshaller**. This marshaller uses the **ProtoStream** library to assist in encoding objects.



IMPORTANT

If the **infinispan-remote** jar is not in use, then the infinispan-remote-query-client Maven dependency must be added to use the **ProtoStreamMarshaller**.

2. Instruct **ProtoStream** library on how to marshall message types by registering per entity marshallers.

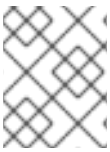
Use the ProtoStreamMarshaller to Encode and Marshall Messages

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.marshall.ProtoStreamMarshaller;
import org.infinispan.protostream.FileDescriptorSource;
import org.infinispan.protostream.SerializationContext;
...
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1").port(11234)
    .marshaller(new ProtoStreamMarshaller());

RemoteCacheManager remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
SerializationContext serCtx =
    ProtoStreamMarshaller.getSerializationContext(remoteCacheManager);
serCtx.registerProtoFiles(FileDescriptorSource.fromResources("/library.proto"));
```

```
serCtx.registerMarshaller(new BookMarshaller());
serCtx.registerMarshaller(new AuthorMarshaller());
// Book and Author classes omitted for brevity
```

In the provided example,

- The **SerializationContext** is provided by the **ProtoStream** library.
- The **SerializationContext.registerProtofile** method receives the name of a *.proto* classpath resource file that contains the message type definitions.
- The **SerializationContext** associated with the **RemoteCacheManager** is obtained, then **ProtoStream** is instructed to marshall the protobuf types.



NOTE

A **RemoteCacheManager** has no **SerializationContext** associated with it unless it was configured to use **ProtoStreamMarshaller**.

23.6.5. Registering Per Entity Marshallers

When using the **ProtoStreamMarshaller** for remote querying purposes, registration of per entity marshallers for domain model types must be provided by the user for each type or marshalling will fail. When writing marshallers, it is essential that they are stateless and threadsafe, as a single instance of them is being used.

The following example shows how to write a marshaller.

BookMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;
...
public class BookMarshaller implements MessageMarshaller<Book> {
   @Override
   public String getTypeName() {
      return "book_sample.Book";
   }
   @Override
   public Class<? extends Book> getJavaClass() {
      return Book.class;
   }
   @Override
   public void writeTo(ProtoStreamWriter writer, Book book) throws IOException {
      writer.writeString("title", book.getTitle());
      writer.writeString("description", book.getDescription());
      writer.writeCollection("authors", book.getAuthors(), Author.class);
   }
   @Override
   public Book readFrom(ProtoStreamReader reader) throws IOException {
      String title = reader.readString("title");
      String description = reader.readString("description");
      int publicationYear = reader.readInt("publicationYear");
      Set<Author> authors = reader.readCollection("authors",
         new HashSet<Author>(), Author.class);
```

```
        return new Book(title, description, publicationYear, authors);
    }
}
```

Once the client has been set up, reading and writing Java objects to the remote cache will use the entity marshallers. The actual data stored in the cache will be protobuf encoded, provided that marshallers were registered with the remote client for all involved types. In the provided example, this would be **Book** and **Author**.

Objects stored in protobuf format are able to be utilized with compatible clients written in different languages.

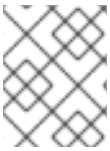
23.6.6. Indexing Protobuf Encoded Entities

You can configure indexing for caches on the JBoss Data Grid server after you configure the client to use Protobuf.

To index entries in a cache, JBoss Data Grid must have access to the message types defined in a Protobuf schema, which is a file with a *.proto* extension.

You provide JBoss Data Grid with a Protobuf schema by placing it in the **___protobuf_metadata** cache with a **put**, **putAll**, **putIfAbsent**, or **replace** operation. Alternatively you can invoke the **ProtobufMetadataManager** MBean via JMX.

Both keys and values of the **___protobuf_metadata** cache are Strings. The key is the file name and the value is contents of the schema file.



NOTE

Users that perform write operations to the **___protobuf_metadata** cache require the **___schema_manager** role.

Registering a Protocol Buffers schema file

```
import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.query.remote.client.ProtobufMetadataManagerConstants;

RemoteCacheManager remoteCacheManager = ... // obtain a RemoteCacheManager

// obtain the '__protobuf_metadata' cache
RemoteCache<String, String> metadataCache =
   remoteCacheManager.getCache(
      ProtobufMetadataManagerConstants.PROTOBUF_METADATA_CACHE_NAME);

String schemaFileContents = ... // this is the contents of the schema file
metadataCache.put("my_protobuf_schema.proto", schemaFileContents);
```

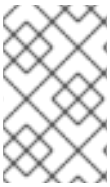
The **ProtobufMetadataManager** is a cluster-wide replicated repository of Protobuf schema definitions or[path].*proto* files. For each running cache manager, a separate **ProtobufMetadataManager** MBean instance exists, and is backed by the **___protobuf_metadata** cache. The **ProtobufMetadataManager** ObjectName uses the following pattern:

```
<jmx domain>:type=RemoteQuery,
   name=<cache manager<methodname>putAllname>,
   component=ProtobufMetadataManager
```

The following signature is used by the method that registers the Protobuf schema file:

```
void registerProtofile(String name, String contents)
```

If indexing is enabled for a cache, all fields of Protobuf-encoded entries are indexed. All Protobuf-encoded entries are searchable, regardless of whether indexing is enabled.



NOTE

Indexing is recommended for improved performance but is not mandatory when using remote queries. Using indexing improves the searching speed but can also reduce the insert/update speeds due to the overhead required to maintain the index.

23.6.7. Controlling Field Indexing

After you enable indexing for a cache, all Protobuf type fields are indexed and stored by default. However, this indexing can degrade performance and result in inefficient querying for Protobuf message types that contain many fields or very large fields.

You can control which fields are indexed using the **@Indexed** and **@Field** annotations directly in the Protobuf schema in comment definitions on the last line of the comment before the message or field to annotate.

@Indexed

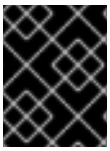
- Applies to message types only.
- Has a boolean value. The default value is **true** so specifying **@Indexed** has the same result as **@Indexed(true)**. If you specify **@Indexed(false)** all field annotations are ignored and no fields are indexed.
- Lets you specify the fields of the message type which are indexed. Using **@Indexed(false)** indicates that no fields are to be indexed. As a result, the **@Field** annotations are ignored.

@Field

- Applies to fields only.
- Has three attributes: **index**, **store**, and **analyze**. Each attribute can have a value of **NO** or **YES**.
  - **index** specifies if the field is indexed, which includes the field in indexed queries.
  - **store** specifies if the field is stored in the index, which allows the field to be used for projections.
  - **analyze** specifies if the field is included in full text searches.
- Defaults to **@Field(index=Index.YES, store=Store.NO, analyze=Analyze.NO)**.
- Replaces the **@IndexedField** annotation.

As of this release, **@IndexedField** is deprecated. If you include this annotation, JBoss Data Grid throws a warning message. You can replace **@IndexedField** annotations with **@Field** annotations as follows:

- **@IndexedField** is equivalent to **@Field(store=Store.YES)**
- **@IndexedField(store=false)** is equivalent to **@Field**
- **@IndexedField(index=false, store=false)** is equivalent to **@Field(index=Index.NO)**



IMPORTANT

If you specify the **@Indexed** and **@Field** annotations, you must include annotations for the message type and each field. Otherwise the entire message is not indexed.

23.6.7.1. Example of an Annotated Message Type

The following is an example of a message type that contains the **@Indexed** and **@Field** annotations:

```
/*
  This type is indexed but not all fields are indexed.
  @Indexed
*/
message Note {

  /*
    This field is indexed but not stored.
    @Field
  */
  optional string text = 1;

  /*
    This field is indexed and stored.
    @Field(store=Store.YES)
  */
  optional string author = 2;

  /*
    This field is stored but not indexed.
    @Field(index=Index.NO, store=Store.YES)
  */
  optional bool isRead = 3;

  /*
    This field is not indexed or stored.
    @Field(index=Index.NO)
  */
  optional int32 priority;
}
```

23.6.7.2. Disabling Indexing for All Protobuf Message Types

You can disable indexing for all Protobuf message types that are not annotated. Set the value of the **indexed_by_default** Protobuf schema option to **false** at the start of each schema file, as follows:

```
option indexed_by_default = false;  //Disable indexing of all types that are not annotated for indexing.
```

23.6.8. Defining Protocol Buffers Schemas With Java Annotations

You can declare Protobuf metadata using Java annotations. Instead of providing a **MessageMarshaller** implementation and a *.proto* schema file, you can add minimal annotations to a Java class and its fields.

The objective of this method is to marshal Java objects to protobuf using the **ProtoStream** library. The **ProtoStream** library internally generates the marshallar and does not require a manually implemented one. The Java annotations require minimal information such as the Protobuf tag number. The rest is inferred based on common sense defaults (Protobuf type, Java collection type, and collection element type) and is possible to override.

The auto-generated schema is registered with the **SerializationContext** and is also available to the users to be used as a reference to implement domain model classes and marshallers for other languages.

The following are examples of Java annotations

User.Java

```
package sample;

import org.infinispan.protostream.annotations.ProtoEnum;
import org.infinispan.protostream.annotations.ProtoEnumValue;
import org.infinispan.protostream.annotations.ProtoField;
import org.infinispan.protostream.annotations.ProtoMessage;

@ProtoMessage(name = "ApplicationUser")
public class User {

   @ProtoEnum(name = "Gender")
   public enum Gender {
      @ProtoEnumValue(number = 1, name = "M")
      MALE,

      @ProtoEnumValue(number = 2, name = "F")
      FEMALE
   }

   @ProtoField(number = 1, required = true)
   public String name;

   @ProtoField(number = 2)
   public Gender gender;
}
```

Note.Java

```
package sample;

import org.infinispan.protostream.annotations.ProtoDoc;
import org.infinispan.protostream.annotations.ProtoField;

@ProtoDoc("@Indexed")
```

```
public class Note {

   private String text;

   private User author;

   @ProtoDoc("@Field")
   @ProtoField(number = 1)
   public String getText() {
      return text;
   }

   public void setText(String text) {
      this.text = text;
   }

   @ProtoDoc("@Field(store = Store.YES)")
   @ProtoField(number = 2)
   public User getAuthor() {
      return author;
   }

   public void setAuthor(User author) {
      this.author = author;
   }
}
```

ProtoSchemaBuilderDemo.Java

```
import org.infinispan.protostream.SerializationContext;
import org.infinispan.protostream.annotations.ProtoSchemaBuilder;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.marshall.ProtoStreamMarshaller;

...

RemoteCacheManager remoteCacheManager = ... // we have a RemoteCacheManager
SerializationContext serCtx =
   ProtoStreamMarshaller.getSerializationContext(remoteCacheManager);

// generate and register a Protobuf schema and marshallers based
// on Note class and the referenced classes (User class)
ProtoSchemaBuilder protoSchemaBuilder = new ProtoSchemaBuilder();
String generatedSchema = protoSchemaBuilder
   .fileName("sample_schema.proto")
   .packageName("sample_package")
   .addClass(Note.class)
   .build(serCtx);

// the types can be marshalled now
assertTrue(serCtx.canMarshall(User.class));
assertTrue(serCtx.canMarshall(Note.class));
assertTrue(serCtx.canMarshall(User.Gender.class));

// display the schema file
System.out.println(generatedSchema);
```

The following is the *.proto* file that is generated by the *ProtoSchemaBuilderDemo.java* example.

Sample_Schema.Proto

```
package sample_package;

 /* @Indexed */
message Note {

   /* @Field */
   optional string text = 1;

   /* @Field(store = Store.YES) */
   optional ApplicationUser author = 2;
}

message ApplicationUser {

   enum Gender {
      M = 1;
      F = 2;
   }

   required string name = 1;
   optional Gender gender = 2;
}
```

The following table lists the supported Java annotations with its application and parameters.

Table 23.2. Java Annotations

Annotation	Applies To	Purpose	Requirement	Parameters
@ProtoDoc	Class/Field/Enum/ Enum member	Specifies the documentation comment that will be attached to the generated Protobuf schema element (message type, field definition, enum type, enum value definition)	Optional	A single String parameter, the documentation text
@ProtoMessage	Class	Specifies the name of the generated message type. If missing, the class name if used instead	Optional	name (String), the name of the generated message type; if missing the Java class name is used by default

Annotation	Applies To	Purpose	Requirement	Parameters
@ProtoField	Field, Getter or Setter	Specifies the Protobuf field number and its Protobuf type. Also indicates if the field is repeated, optional or required and its (optional) default value. If the Java field type is an interface or an abstract class, its actual type must be indicated. If the field is repeatable and the declared collection type is abstract then the actual collection implementation type must be specified. If this annotation is missing, the field is ignored for marshalling (it is transient). A class must have at least one **@ProtoField** annotated field to be considered Protobuf marshallable.	Required	number (int, mandatory), the Protobuf number type (org.infinispan.protostream.descriptors.Type, optional), the Protobuf type, it can usually be inferred required (boolean, optional)name (String, optional), the Protobuf namejavaType (Class, optional), the actual type, only needed if declared type is abstract collectionImplementation (Class, optional), the actual collection type, only needed if declared type is abstract defaultValue (String, optional), the string must have the proper format according to the Java field type
@ProtoEnum	Enum	Specifies the name of the generated enum type. If missing, the Java enum name if used instead	Optional	name (String), the name of the generated enum type; if missing the Java enum name is used by default

Annotation	Applies To	Purpose	Requirement	Parameters
@ProtoEnumValue	Enum member	Specifies the numeric value of the corresponding Protobuf enum value	Required	number (int, mandatory), the Protobuf number name (String), the Protobuf name; if missing the name of the Java member is used

NOTE

The **@ProtoDoc** annotation can be used to provide documentation comments in the generated schema and also allows to inject the **@Indexed** and **@Field** annotations where needed. See Custom Fields Indexing with Protobuf for additional information.

PART III. SECURING DATA IN RED HAT JBOSS DATA GRID

CHAPTER 24. 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).

CHAPTER 25. RED HAT JBOSS DATA GRID SECURITY: AUTHORIZATION AND AUTHENTICATION

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

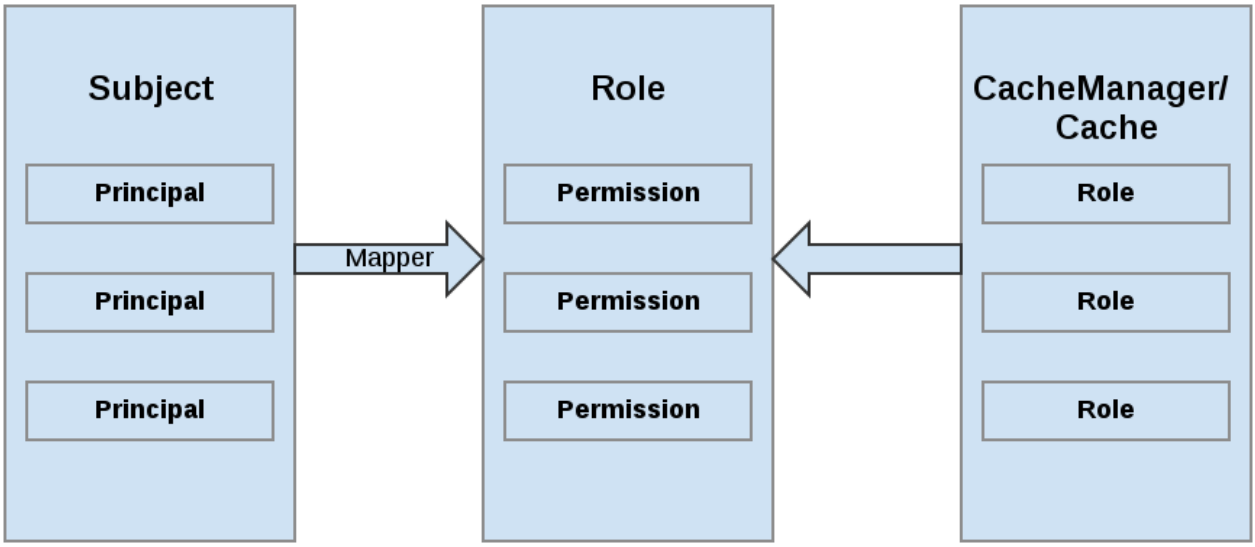
Red Hat JBoss Data Grid is able to perform authorization on CacheManagers and Caches. JBoss Data Grid authorization is built on standard security features available in a JDK, such as JAAS and the SecurityManager.

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 25.1. Roles and Permissions Mapping



25.2. 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 25.1. 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 25.2. 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.

25.3. 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 25.3. 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**

25.4. CONFIGURING AUTHENTICATION AND ROLE MAPPING USING LOGIN MODULES

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

Mapping a Principal

```
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;
      }
    }
```

IMPORTANT

For information on configuring an LDAP server, or specifying users and roles in an LDAP server, refer to the Red Hat Directory Server *Administration Guide* .

25.5. CONFIGURING RED HAT JBOSS DATA GRID FOR AUTHORIZATION

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

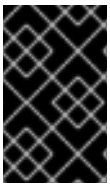
Each cache container determines:

- 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 may be applied on a cache-per-cache basis, using the roles defined at the cache-container level, as follows:



IMPORTANT

Any cache that is intended to require authentication must have a listing of roles defined; otherwise authentication is not enforced as the no-anonymous policy is defined by the cache's authorization.

Programmatic CacheManager Authorization (Library Mode)

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

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");
```

IMPORTANT

The REST protocol is not supported for use with authorization, and any attempts to access a cache with authorization enabled will result in a **SecurityException**.

25.6. DATA SECURITY FOR LIBRARY MODE

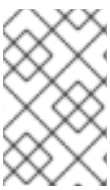
25.6.1. Subject and Principal Classes

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.

25.6.2. 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.

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**.

25.6.3. 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 **LoginModule**s.
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 **LoginModule**s. 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 **LoginModule**s 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 **LoginModule**s 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 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 **LoginModule**s 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 interface has one method to implement:

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

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. **LoginModule**s pass an array of **Callback**s 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.

25.7. SECURING INTERFACES

25.7.1. Securing Interfaces

While the Hot Rod interface may be secured programmatically, both the memcached and REST interfaces must be secured declaratively. Instructions for securing these interfaces are located in the JBoss Data Grid *Administration and Configuration Guide* .

25.7.2. Hot Rod Interface Security

25.7.2.1. 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.

Secure Hot Rod Using SSL/TLS

```
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.UnsupportedCallbackException;
import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.impl.ConfigurationProperties;

[...]

public class SslConfiguration {

   public static final String ISPN_IP = "127.0.0.1";
   public static final String SERVER_NAME = "node0";
   public static final String SASL_MECH = "EXTERNAL";

   private static final String KEYSTORE_PATH = "./keystore_client.jks";
   private static final String KEYSTORE_PASSWORD = "secret";
   private static final String TRUSTSTORE_PATH = "./truststore_client.jks";
   private static final String TRUSTSTORE_PASSWORD = "secret";

   SslConfiguration(boolean enabled,
                    String keyStoreFileName,
                    char[] keyStorePassword,
                    SSLContext sslContext,
                    String trustStoreFileName,
                    char[] trustStorePassword) {
      ConfigurationBuilder builder = new ConfigurationBuilder();
      builder.addServer()
         .host(ISPN_IP)
         .port(ConfigurationProperties.DEFAULT_HOTROD_PORT);
      //setup auth
      builder.security()
         .authentication()
         .serverName(SERVER_NAME)
         .saslMechanism(SASL_MECH)
         .enable()
         .callbackHandler(new VoidCallbackHandler());
      //setup encrypt
      builder.security()
         .ssl()
         .enable()
         .keyStoreFileName(KEYSTORE_PATH)
         .keyStorePassword(KEYSTORE_PASSWORD.toCharArray())
         .trustStoreFileName(TRUSTSTORE_PATH)
         .trustStorePassword(TRUSTSTORE_PASSWORD.toCharArray());

      RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());
      RemoteCache<Object, Object> cache =
cacheManager.getCache(RemoteCacheManager.DEFAULT_CACHE_NAME);
   }

   private static class VoidCallbackHandler implements CallbackHandler {
      @Override
      public void handle(Callback[] clbcks) throws IOException, UnsupportedCallbackException {
```

```
      }
   }
}
```

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 Password Vault section of the JBoss Enterprise Application Platform security documentation. .

25.7.2.2. Securing Hot Rod to LDAP Server using SSL

When connecting to an LDAP server with SSL enabled it may be necessary to specify a trust store or key store containing the appropriate certificates.

PLAIN authentication over SSL may be used for Hot Rod client authentication against an LDAP server. The Hot Rod client sends plain text credentials to the JBoss Data Grid server over SSL, and the server subsequently verifies the provided credentials against the specified LDAP server. In addition, a secure connection must be configured between the JBoss Data Grid server and the LDAP server. Refer to the JBoss Data Grid *Administration and Configuration Guide* for additional information on configuring the server to communicate to an LDAP backend. The example below demonstrates configuring **PLAIN** authentication over SSL on the Hot Rod client side:

Hot Rod Client Authentication to LDAP Server

```
import static org.infinispan.demo.util.CacheOps.dumpCache;
import static org.infinispan.demo.util.CacheOps.onCache;
import static org.infinispan.demo.util.CacheOps.putTestKV;
import static org.infinispan.demo.util.CmdArgs.LOGIN_KEY;
import static org.infinispan.demo.util.CmdArgs.PASS_KEY;
import static org.infinispan.demo.util.CmdArgs.getCredentials;

import java.util.Map;

import javax.net.ssl.SSLContext;

import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.impl.ConfigurationProperties;
import org.infinispan.commons.util.SslContextFactory;
import org.infinispan.demo.util.SaslUtils.SimpleLoginHandler;

public class HotRodPlainAuthOverSSL {

   public static final String ISPN_IP = "127.0.0.1";
   public static final String SERVER_NAME = "node0";
   public static final String SASL_MECH = "PLAIN";
   private static final String SECURITY_REALM = "ApplicationRealm";

   private static final String TRUSTSTORE_PATH = "./truststore_client.jks";
   private static final String TRUSTSTORE_PASSWORD = "secret";

   public static void main(String[] args) {
```

```
        Map<String, String> userArgs = null;
        try {
            userArgs = getCredentials(args);
        } catch (IllegalArgumentException e) {
            System.err.println(e.getMessage());
            System.err.println(
                    "Invalid credentials format, plase provide credentials (and optionally cache name) with --
cache=<cache> --user=<user> --password=<password>");
            System.exit(1);
        }

        ConfigurationBuilder builder = new ConfigurationBuilder();
        builder.addServer().host(ISPN_IP).port(ConfigurationProperties.DEFAULT_HOTROD_PORT);

        //set up PLAIN auth

builder.security().authentication().serverName(SERVER_NAME).saslMechanism(SASL_MECH).enable
().callbackHandler(
                new SimpleLoginHandler(userArgs.get(LOGIN_KEY), userArgs.get(PASS_KEY),
SECURITY_REALM));

        //set up SSL
        SSLContext cont = SslContextFactory.getContext(null, null, TRUSTSTORE_PATH,
TRUSTSTORE_PASSWORD.toCharArray());
        builder.security().ssl().sslContext(cont).enable();

        RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());
        RemoteCache<Object, Object> cache =
cacheManager.getCache(RemoteCacheManager.DEFAULT_CACHE_NAME);

        onCache(cache, putTestKV.andThen(dumpCache));

        cacheManager.stop();
        System.exit(0);
    }

}
```

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* .

25.7.2.3. User Authentication over Hot Rod Using SASL

25.7.2.3.1. 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 than 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.

25.7.2.3.2. Configure Hot Rod Authentication (GSSAPI/Kerberos)

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

Configure SASL GSSAPI/Kerberos Authentication - Client-side Configuration

1. Ensure that the Server-Side configuration has been completed. As this is configured declaratively this configuration is found in the JBoss Data Grid *Administration and Configuration Guide* .
2. Define a login module in a login configuration file (*gss.conf*) on the client side:
[source],options="nowrap"

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

1. 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.

2. Implement the **CallbackHandler**:

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

3. Configure the Hot Rod Client, as seen in the below snippet:

```
LoginContext lc = new LoginContext("GssExample", new MyCallbackHandler("krb_user",
"krb_password".toCharArra()));
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")
         .saslMechanism("GSSAPI")
         .clientSubject(clientSubject)
         .callbackHandler(new MyCallbackHandler());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

25.7.2.3.3. Configure Hot Rod Authentication (MD5)

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

1. Ensure that the server has been configured for MD5 Authentication. Instructions for performing this configuration on the server are found in JBoss Data Grid's *Administration and Configuration Guide* .

2. Implement the **CallbackHandler**:

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

3. Connect the client to the configured Hot Rod connector as seen below:

```
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");
```

25.7.2.3.4. Configure Hot Rod C++ Authentication (GSSAPI/Kerberos)

Use the following steps to set up Hot Rod C++ client authentication using the SASL GSSAPI/Kerberos mechanism:

Configure SASL GSSAPI/Kerberos Authentication - Client-side Configuration

1. Ensure that the Server-Side configuration has been completed. As this is configured declaratively this configuration is found in the JBoss Data Grid *Administration and Configuration Guide* .

Below is a complete example of using Kerberos with the Hot Rod C++ client:

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"

#include "infinispan/hotrod/JBasicMarshaller.h"
#include <sasl/saslplug.h>
#include <krb5.h>
#include <err.h>
#include <stdlib.h>

using namespace infinispan::hotrod;

int kinit();
void kdestroy();

 /* Hotrod SASL is based on Cyrus Sasl libraries.
  * Check cyrus docs for more info on how to setup callbacks
  * https://www.cyrusimap.org/sasl/
  */
static int simple(void*  context , int id, const char **result, unsigned *len) {
    *result = *(char**)context;
    if (len)
        *len = strlen(*result);
    return SASL_OK;
}

static int getsecret(void* /* conn */, void*  context, int id, sasl_secret_t **psecret) {
    char *secret_data=*(char**)context;
    size_t len = strlen(secret_data);
    static sasl_secret_t *x;
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    x->len = len;
    strcpy((char *) x->data, secret_data);
    *psecret = x;
    return SASL_OK;
}

char *pusername;
char *psecret;

static std::vector<sasl_callback_t> callbackHandler {
        { SASL_CB_USER, (sasl_callback_ft) &simple, &pusername },
        {SASL_CB_PASS, (sasl_callback_ft) &getsecret, &psecret },
        {SASL_CB_LIST_END, NULL, NULL } };

int kinit();
void kdestroy();
```

```
int main(int argc, char** argv) {
    int result = 0;
    {
        kinit();
        ConfigurationBuilder builder;
        char username[]="supervisor@INFINISPAN.ORG";
        char secret_data[]="lessStrongPassword";
        pusername=username;
        psecret=secret_data;
        builder.addServer().host(argc > 1 ? argv[1] : "127.0.0.1").port(argc > 2 ? atoi(argv[2]) : 11222);
        builder.protocolVersion(Configuration::PROTOCOL_VERSION_24);
        builder.security().authentication().saslMechanism("GSSAPI").serverFQDN(
                "node0").callbackHandler(callbackHandler).enable();
        builder.balancingStrategyProducer(nullptr);
        RemoteCacheManager cacheManager(builder.build(), false);
        BasicMarshaller<std::string> *km = new BasicMarshaller<std::string>();
        BasicMarshaller<std::string> *vm = new BasicMarshaller<std::string>();
        RemoteCache<std::string, std::string> cache = cacheManager.getCache<std::string, std::string>
(km,
                &Marshaller<std::string>::destroy, vm, &Marshaller<std::string>::destroy,
std::string("authCache"));
        cacheManager.start();
        try {
            cache.put("key", "value");
            std::shared_ptr<std::string> ret(cache.get("key"));
            result = 0;
        } catch (Exception& ex) {
            std::cerr << "FAIL: 'supervisor' should read and write" << std::endl;
            result = -1;
        }
        cacheManager.stop();
        std::cout << "PASS: 'GSSAPI' sasl authorization" << std::endl;
        kdestroy();
    }
    return result;
}

krb5_context context;
krb5_creds creds;
krb5_principal client_princ = NULL;

int kinit() {
    // Delegate Kerberos setup to the system
    setenv("KRB5CCNAME", "krb5cc_hotrod", 1);
    setenv("KRB5_CONFIG", "krb5.conf", 1);
    std::system("echo lessStrongPassword | kinit -c krb5cc_hotrod supervisor@INFINISPAN.ORG");
}
void kdestroy() {
    std::system("kdestroy");
}
```

25.7.2.3.5. Configure Hot Rod C++ Authentication (MD5)

Use the following steps to set up Hot Rod C++ client authentication using the SASL MD5 mechanism:

Configure SASL MD5 Authentication - Client-side Configuration

1. Ensure that the Server-Side configuration has been completed. As this is configured declaratively this configuration is found in the JBoss Data Grid *Administration and Configuration Guide* .

Below is a complete example of using SASL MD5 with the Hot Rod C++ client:

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"

#include "infinispan/hotrod/JBasicMarshaller.h"
#include <sasl/saslplug.h>
#include <krb5.h>


using namespace infinispan::hotrod;

 /* Hotrod SASL is based on Cyrus Sasl libraries.
  * Check cyrus docs for more info on how to setup callbacks
  * https://www.cyrusimap.org/sasl/
  */
static int simple(void*  context , int id, const char **result, unsigned *len) {
    *result = *(char**)context;
    if (len)
        *len = strlen(*result);
    return SASL_OK;
}

static int getsecret(void* /* conn */, void*  context, int id, sasl_secret_t **psecret) {
    char *secret_data=*(char**)context;
    size_t len = strlen(secret_data);
    static sasl_secret_t *x;
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    x->len = len;
    strcpy((char *) x->data, secret_data);
    *psecret = x;
    return SASL_OK;
}

char *pusername;
char *psecret;

static std::vector<sasl_callback_t> callbackHandler {
        { SASL_CB_AUTHNAME, (sasl_callback_ft) &simple, &pusername },
        {SASL_CB_PASS, (sasl_callback_ft) &getsecret, &psecret },
        {SASL_CB_LIST_END, NULL, NULL } };

/* This sample authenticates the client with
 * user=reader
 * password=password
 * credential, which is an account that can only do WRITE
 * on the server.
 */
```

```
int main(int argc, char** argv) {
   int result = 0;
   {
      ConfigurationBuilder builder;
      char username[]="reader";
      char secret_data[]="password";
      pusername=username;
      psecret=secret_data;
      builder.addServer().host("127.0.0.1").port(11222);
      builder.protocolVersion(Configuration::PROTOCOL_VERSION_24);
      builder.security().authentication().saslMechanism("DIGEST-
MD5").serverFQDN("node0").callbackHandler(callbackHandler).enable();
      RemoteCacheManager cacheManager(builder.build(), false);
      BasicMarshaller<std::string> *km = new BasicMarshaller<std::string>();
      BasicMarshaller<std::string> *vm = new BasicMarshaller<std::string>();
      auto cache = cacheManager.getCache<std::string, std::string>(km,
&Marshaller<std::string>::destroy, vm, &Marshaller<std::string>::destroy, std::string("authCache"));
      cacheManager.start();
      std::shared_ptr<std::string> ret(cache.get("key"));
      try {
         cache.put("key", "value");
         std::cerr << "FAIL: 'reader' should not write" << std::endl;
         return -1;
      } catch (Exception& ex) {

      }
      std::cout << "PASS: 'DIGEST-MD5' sasl authorization" << std::endl;
      cacheManager.stop();
   }
   return result;
}
```

25.7.2.3.6. Configure Hot Rod C++ Authentication (PLAIN)

Use the following steps to set up Hot Rod C++ client authentication using the SASL PLAIN mechanism:

Configure SASL PLAIN Authentication - Client-side Configuration

1. Ensure that the Server-Side configuration has been completed. As this is configured declaratively this configuration is found in the JBoss Data Grid *Administration and Configuration Guide* .

Below is a complete example of using SASL PLAIN with the Hot Rod C++ client:

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"

#include "infinispan/hotrod/JBasicMarshaller.h"
#include <sasl/saslplug.h>
#include <krb5.h>

using namespace infinispan::hotrod;
```

```
 /* Hotrod SASL is based on Cyrus Sasl libraries.
  * Check cyrus docs for more info on how to setup callbacks
  * https://www.cyrusimap.org/sasl/
  */
static int simple(void*  context , int id, const char **result, unsigned *len) {
    *result = *(char**)context;
    if (len)
        *len = strlen(*result);
    return SASL_OK;
}

static int getsecret(void* /* conn */, void*  context, int id, sasl_secret_t **psecret) {
    char *secret_data=*(char**)context;
    size_t len = strlen(secret_data);
    static sasl_secret_t *x;
    x = (sasl_secret_t *) realloc(x, sizeof(sasl_secret_t) + len);
    x->len = len;
    strcpy((char *) x->data, secret_data);
    *psecret = x;
    return SASL_OK;
}

char *pusername;
char *psecret;

static std::vector<sasl_callback_t> callbackHandler {
        { SASL_CB_AUTHNAME, (sasl_callback_ft) &simple, &pusername },
        {SASL_CB_PASS, (sasl_callback_ft) &getsecret, &psecret },
        {SASL_CB_LIST_END, NULL, NULL } };

/* This sample authenticates the client with
 * user=writer
 * password=somePassword
 * credential, which is an account that can only do WRITE
 * on the server.
 */
int main(int argc, char** argv) {
    int result = 0;
    {
        ConfigurationBuilder builder;
        char username[]="writer";
        char secret_data[]="somePassword";
        pusername=username;
        psecret=secret_data;
        builder.addServer().host("127.0.0.1").port(11222);
        builder.protocolVersion(Configuration::PROTOCOL_VERSION_24);

builder.security().authentication().saslMechanism("PLAIN").serverFQDN("node0").callbackHandler(call
backHandler).enable();
        RemoteCacheManager cacheManager(builder.build(), false);
        BasicMarshaller<std::string> *km = new BasicMarshaller<std::string>();
        BasicMarshaller<std::string> *vm = new BasicMarshaller<std::string>();
        auto cache = cacheManager.getCache<std::string, std::string>(km,
&Marshaller<std::string>::destroy, vm, &Marshaller<std::string>::destroy, std::string("authCache"));
        cacheManager.start();
        cache.put("key", "value");
```

```
        try {
            std::shared_ptr<std::string> ret(cache.get("key"));
            std::cerr << "FAIL: 'writer' should not read" << std::endl;
            return -1;
        } catch (Exception& ex) {

        }
        std::cout << "PASS: 'PLAIN' sasl authorization" << std::endl;
        cacheManager.stop();
    }
    return result;
}
```

25.7.2.3.7. Configure Hot Rod C# Authentication (EXTERNAL)

Use the following steps to set up Hot Rod C# client authentication using the SASL EXTERNAL mechanism:

Configure SASL EXTERNAL Authentication - Client-side Configuration

1. Ensure that the Server-Side configuration has been completed. As this is configured declaratively this configuration is found in the JBoss Data Grid *Administration and Configuration Guide* .

Below is a complete example of using SASL EXTERNAL with the Hot Rod C# client:

```
using Infinispan.HotRod;
using Infinispan.HotRod.Config;
using System;
using System.Text;

namespace Authentication
{
    class Program
    {

        static void Main(string[] args)
        {
            ConfigurationBuilder conf = new ConfigurationBuilder();
            conf.AddServer()
                    .Host("127.0.0.1")
                    .Port(11222)
                    .ConnectionTimeout(90000)
                    .SocketTimeout(900);
// Enable EXTERNAL mechanism for SASL
            conf.Security().Authentication()
                            .Enable()
                            .SaslMechanism("EXTERNAL")
                            .ServerFQDN("node0");
// Enable SSL (EXTERNAL is based on the client certificate)
            conf.Ssl().Enable()
                .ServerCAFile("infinispan-ca.pem")
                .ClientCertificateFile("keystore_client.p12");
// end of SASL configuration
// The subject specified in the truststore_client.p12 cert will be used to identify the client
```

```
            IMarshaller marshaller = new JBasicMarshaller();
            conf.Marshaller(marshaller);
            Configuration c = conf.Build();
            RemoteCacheManager remoteManager = new RemoteCacheManager(c, true);
            IRemoteCache<string, string> authCache = remoteManager.GetCache<string, string>
("authCache");
            authCache.Put("K1", "V1");
            authCache.Get("K1");
            authCache.Clear();

        }
    }
}
```

25.7.2.3.8. Configure Hot Rod C# Authentication (MD5)

Use the following steps to set up Hot Rod C# client authentication using the SASL MD5 mechanism:

Configure SASL MD5 Authentication - Client-side Configuration

1. Ensure that the Server-Side configuration has been completed. As this is configured declaratively this configuration is found in the JBoss Data Grid *Administration and Configuration Guide* .

Below is a complete example of using SASL MD5 with the Hot Rod C# client:

```
using Infinispan.HotRod;
using Infinispan.HotRod.Config;
using System;
using System.Text;

namespace Authentication
{
    class Program
    {
        static void Main(string[] args)
        {
            ConfigurationBuilder conf = new ConfigurationBuilder();
            conf.AddServer()
                    .Host("127.0.0.1")
                    .Port(11222)
                    .ConnectionTimeout(90000)
                    .SocketTimeout(900);
// Enable authentication use PLAIN as mechanism  (DIGEST-MD5 can be used the same way)
// and setup user password and realm
            conf.Security().Authentication()
                        .Enable()
                        .SaslMechanism("DIGEST-MD5")
                        .ServerFQDN("node0")
                        .SetupCallback("writer", "somePassword", "ApplicationRealm");
// end of SASL configuration
            IMarshaller marshaller = new JBasicMarshaller();
            conf.Marshaller(marshaller);
            Configuration c = conf.Build();
            RemoteCacheManager remoteManager = new RemoteCacheManager(c, true);
```

```
            IRemoteCache<string, string> authCache = remoteManager.GetCache<string, string>
("authCache");
            authCache.Put("K1", "V1");
            authCache.Get("K1");
            authCache.Clear();
        }
    }
}
```

25.7.3. Hot Rod C++ Client Encryption

By default all communication with the remote server is unencrypted; however, TLS encryption may be enabled by defining the server's key via the **serverCAFile** method on the **SslConfigurationBuilder**. Additionally, the client's certificate may be defined with the **clientCertificateFile**, allowing for client authentication.

The following example demonstrates defining a server key with an optional client certificate:

Hot Rod C++ TLS Example

```
#include "infinispan/hotrod/ConfigurationBuilder.h"
#include "infinispan/hotrod/RemoteCacheManager.h"
#include "infinispan/hotrod/RemoteCache.h"
#include "infinispan/hotrod/Version.h"

#include "infinispan/hotrod/JBasicMarshaller.h"
#include <stdlib.h>
#include <iostream>
#include <memory>
#include <typeinfo>

using namespace infinispan::hotrod;

int main(int argc, char** argv) {
    std::cout << "TLS Test" << std::endl;
    if (argc < 2) {
        std::cerr << "Usage: " << argv[0] << " server_ca_file [client_ca_file]" << std::endl;
        return 1;
    }
    {
      ConfigurationBuilder builder;

builder.addServer().host("127.0.0.1").port(11222).protocolVersion(Configuration::PROTOCOL_VERSI
ON_24);
      // Enable the TLS layer and install the server public key
      // this ensure that the server is authenticated
      builder.ssl().enable().serverCAFile(argv[1]);
      if (argc > 2) {
          // Send a client certificate for authentication (optional)
          // without this the socket will only be encrypted
          std::cout << "Using supplied client certificate for authentication against the server" << std::endl;
          builder.ssl().clientCertificateFile(argv[2]);
      }
      // That's all. Now do business as usual
      RemoteCacheManager cacheManager(builder.build(), false);
```

```
    BasicMarshaller<std::string> *km = new BasicMarshaller<std::string>();
    BasicMarshaller<std::string> *vm = new BasicMarshaller<std::string>();
    RemoteCache<std::string, std::string> cache = cacheManager.getCache<std::string, std::string>
(km,
        &Marshaller<std::string>::destroy, vm, &Marshaller<std::string>::destroy );
    cacheManager.start();
    cache.clear();
    std::string k1("key13");
    std::string v1("boron");

    cache.put(k1, v1);
    std::unique_ptr<std::string> rv(cache.get(k1));
    if (rv->compare(v1)) {
      std::cerr << "get/put fail for " << k1 << " got " << *rv << " expected " << v1 << std::endl;
      return 1;
    }
    cacheManager.stop();
  }
  return 0;
}
```

The client may also indicate which hostname it is attempting to connect to at the start of the TLS/SNI handshaking process by providing a value to the **sniHostName** function. For instance, the following could be used:

```
[...]
  builder.ssl().enable().serverCAFile(argv[1]).sniHostName("sni");
[...]
```

25.7.4. Hot Rod C# Client Encryption

By default all communication with the remote server is unencrypted; however, TLS encryption may be enabled by defining the server's key via the **ServerCAFile** method on the **SslConfigurationBuilder**. Additionally, the client's certificate may be defined with the **ClientCertificateFile**, allowing for client authentication.

The following example demonstrates defining a server key with an optional client certificate:

Hot Rod C# TLS Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Infinispan.HotRod;
using Infinispan.HotRod.Config;

namespace TLS
{
  /// <summary>
  /// This sample code shows how to perform operations over TLS using the C# client
  /// </summary>

  class TLS
```

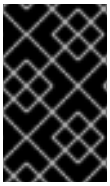
```
    {
        static void Main(string[] args)
        {
            // Cache manager setup
            RemoteCacheManager remoteManager;
            ConfigurationBuilder conf = new ConfigurationBuilder();

conf.AddServer().Host("127.0.0.1").Port(11222).ConnectionTimeout(90000).SocketTimeout(900);
            SslConfigurationBuilder sslConfB = conf.Ssl();
            // Retrieve the server public certificate, needed to do server authentication. Mandatory
            if (!System.IO.File.Exists("resources/infinispan-ca.pem"))
            {
                Console.WriteLine("File not found: resources/infinispan-ca.pem.");
                Environment.Exit(-1);
            }
            sslConfB.Enable().ServerCAFile("resources/infinispan-ca.pem");
            // Retrieve the client public certificate, needed if the server requires client authentication.
Optional
            if (!System.IO.File.Exists("resources/keystore_client.p12"))
            {
                Console.WriteLine("File not found: resources/keystore_client.p12.");
                Environment.Exit(-1);
            }
            sslConfB.ClientCertificateFile("resources/keystore_client.p12");

            // Usual business now
            conf.Marshaller(new JBasicMarshaller());
            remoteManager = new RemoteCacheManager(conf.Build(), true);
            IRemoteCache<string, string> testCache = remoteManager.GetCache<string, string>();
            testCache.Clear();
            string k1 = "key13";
            string v1 = "boron";
            testCache.Put(k1, v1);
        }
    }
}
```

The client may also indicate which hostname it is attempting to connect to at the start of the TLS/SNI handshaking process by providing a value to **SniHostName**. For instance, the following call could be included immediately after defining the **ServerCAFile**:

```
[...]
sslConfB.ServerCAFile("resources/infinispan-ca.pem").SniHostName("sni");
[...]
```

25.7.5. Hot Rod Node.js Encryption

The Node.js client supports encryption via SSL/TLS with optional TLS/SNI support. To configure this on the client it is necessary to create a Java KeyStore (JKS) using the **keytool** application included in the JDK. The created keystore must contain the keys and certificates necessary for the JBoss Data Grid server to authorize connections, and the JBoss Data Grid server must be configured for encryption. For details on configuring the server for encryption, refer to the JBoss Data Grid ***Administration and Configuration Guide***.



IMPORTANT

The Node.js client implementation of TLS/SSL does not allow self-signed certificates. It is recommended to either configure a local Certificate Authority to sign certificates, or to use a free, open Certificate Authority, if certificates were previously self-signed.

By defining the location of a trusted certificate the client connection may be authorized by the server:

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
  {
    ssl: {
      enabled: true,
      trustCerts: ['my-root-ca.crt.pem']
    }
  }
);
```

In addition, the client may also read trusted certificates from **PKCS#12** or **PFX** format key stores:

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
  {
    ssl: {
      enabled: true,
      cryptoStore: {
        path: 'my-truststore.p12',
        passphrase: 'secret'
      }
    }
  }
);
```

In addition, the client may be configured with encrypted authentication. To configure authentication it is necessary to provide the location of the private key, the passphrase, and the certificate key of the client:

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
  {
    ssl: {
      enabled: true,
      trustCerts: ['my-root-ca.crt.pem'],
      clientAuth: {
        key: 'privkey.pem',
        passphrase: 'secret',
        cert: 'cert.pem'
      }
    }
  }
);
```

The client may also indicate which hostname it is attempting to connect to at the start of the TLS/SNI handshaking process by including the **sniHostName** directive:

```
var connected = infinispan.client({port: 11222, host: '127.0.0.1'},
  {
```

```
    ssl: {
      enabled: true,
      trustCerts: ['my-root-ca.crt.pem']
      sniHostName: 'example.com'
    }
  }
);
```

NOTE

If no **sniHostName** is provided then the client will send **localhost** as the SNI parameter. If the server's default realm does not match **localhost** an error will be thrown.

25.8. THE SECURITY AUDIT LOGGER

25.8.1. 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.

25.8.2. Configure the Security Audit Logger (Library Mode)

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

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

25.8.3. Custom Audit Loggers

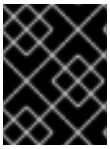
Users can implement custom audit loggers in Red Hat JBoss Data Grid Library and Remote Client-Server Mode. The custom logger must implement the **org.infinispan.security.AuditLogger** interface. If no custom logger is provided, the default logger (**DefaultAuditLogger**) is used.

CHAPTER 26. SECURITY FOR CLUSTER TRAFFIC

26.1. 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.

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"**).

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.

26.2. 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.

Implementing Node Authorization

```
public class AuthorizingServerCallbackHandler implements CallbackHandler {

@Override
public void handle(Callback[] callbacks) throws IOException,
UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            <!-- Additional configuration information here -->
            if (callback instanceof AuthorizeCallback) {
                AuthorizeCallback acb = (AuthorizeCallback) callback;
                if (!"myclusterrole".equals(acb.getAuthenticationID())) {
                    throw new SecurityException("Unauthorized node " +user);
                }
            <!-- Additional configuration information here -->
        }
    }
}
```

PART IV. ADVANCED FEATURES IN RED HAT JBOSS DATA GRID

CHAPTER 27. ADVANCED FEATURES IN RED HAT JBOSS DATA GRID

JBoss Data Grid includes advanced features. This section describes these features and provides instructions for using them.

CHAPTER 28. MONITORING

28.1. MONITORING

28.2. ABOUT JAVA MANAGEMENT EXTENSIONS (JMX)

28.2.1. About Java Management Extensions (JMX)

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

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

28.2.2. Using JMX with Red Hat JBoss Data Grid

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

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

28.2.3. Enabling JMX for Cache Instances

You can enable JMX statistics at the *Cache* level either declaratively or programmatically.

Declaratively Enabling JMX at the *Cache* Level

Add the **statistics** attribute to the target **<*-cache>** element as follows:

```
<*-cache statistics="true">
```

Programmatically Enabling JMX at the *Cache* Level

Programmatically enable JMX at the cache level as follows:

```
Configuration configuration = new
ConfigurationBuilder().jmxStatistics().enable().build();
```

28.2.4. Enabling JMX for CacheManagers

You can enable JMX statistics at the *CacheManager* level either declaratively or programmatically.

Declaratively Enabling JMX at the *CacheManager* Level

Add the **statistics** attribute to the **<cache-container>** element as follows:

```
<cache-container statistics="true">
```

Programmatically Enabling JMX at the *CacheManager* Level

Programmatically enable JMX at the *CacheManager* level as follows:

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

28.2.5. Multiple JMX Domains

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

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

Set a CacheManager Name Programmatically

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

```
GlobalConfiguration globalConfiguration = new
GlobalConfigurationBuilder().globalJmxStatistics().enable().
cacheManagerName("Hibernate2LC").build();
```

28.2.6. Registering MBeans in Non-Default MBean Servers

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

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

To Add the Fully Qualified Domain Name Programmatically

Add the following code:

```
GlobalConfiguration globalConfiguration = new
GlobalConfigurationBuilder().globalJmxStatistics().enable().
mBeanServerLookup("com.acme.MyMBeanServerLookup").build();
```

28.3. STATISTICSINFOMBEAN

The **StatisticsInfoMBean** MBean accesses the **Statistics** object as described in the previous section.

CHAPTER 29. RED HAT JBOSS DATA GRID AS LUCENE DIRECTORY

29.1. RED HAT JBOSS DATA GRID AS LUCENE DIRECTORY

Red Hat JBoss Data Grid can be used as a shared, in-memory index (Infinispan Directory) for Hibernate Search queries on a relational database. By default, Hibernate Search uses a local filesystem to store the Lucene indexes but optionally it can be configured to use JBoss Data Grid as a storage to achieve real-time replication across multiple server nodes.

In the Infinispan Directory, the index is stored in memory and shared across multiple nodes. The Infinispan Directory acts as a single directory distributed across all participating nodes. An index update on one node updates the index on all the nodes. Index can be searched immediately after the node update across the cluster. The default Hibernate Search configuration replicates the data defining the index across all the nodes.

Data distribution for large indexes may be enabled to consume less memory; however, this will come at a cost of locality resulting in query operations less efficient. The indexed data can also be offloaded to a CacheStore configured on each node or configure a single centralized CacheStore shared by each node.

NOTE

While enabling distribution rather than replication might save memory, the queries will be slower. Enabling a CacheStore might save even more memory, but at cost of additional performance if used for passivation.

29.2. CONFIGURATION

The directory provider is enabled by specifying it per index. If the **default** index is specified then all indexes will use the directory provider unless specified:

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

This gives a cluster-replicated index, but the default configuration does not enable any form of permanent persistence for the index. To enable such a feature provide an Infinispan configuration file.

Hibernate Search requires a *CacheManager* to use Infinispan. It can look up and reuse an existing *CacheManager* , via JNDI, or start and manage a new one. When looking up an existing *CacheManager* this will be provided from the Infinispan subsystem where it was originally registered; for instance, if this was registered via JBoss EAP, then JBoss EAP's Infinispan subsystem will provide the *CacheManager* .

NOTE

When using JNDI to register a *CacheManager* , it must be done using Red Hat JBoss Data Grid configuration files only.

To use an existing CacheManager via JNDI (optional parameter):

hibernate.search.infinispan.cachemanager_jndiname = [jndiname]

To start a new *CacheManager* from a configuration file (optional parameter):

hibernate.search.infinispan.configuration_resourcename = [infinispan configuration filename]

If both the parameters are defined, JNDI will have priority. If none of these are defined, Hibernate Search will use the default Infinispan configuration which does not store the index in a persistent cache store.

29.3. RED HAT JBOSS DATA GRID MODULES

Red Hat JBoss Data Grid directory provider for Hibernate Search are distributed as part of the JBoss Data Grid Library Modules for JBoss EAP. Download the files from the Red Hat Customer Portal .

Unpack the archive into the modules/ directory in the target JBoss Enterprise Application Platform folder.

Add the following entry to the *MANIFEST.MF* file in the project archive:

```
Dependencies: org.hibernate.search.orm services
```

For more information, see the Generate MANIFEST.MF entries using Maven section in the Red Hat JBoss EAP *Development Guide*.

29.4. LUCENE DIRECTORY CONFIGURATION FOR REPLICATED INDEXING

Define the following properties in the Hibernate configuration and in the Persistence unit configuration file when using standard JPA. For instance, to change all of the default storage indexes the following property could be configured:

```
hibernate.search.default.directory_provider=infinispan
```

This may also be performed on unique indexes. In the following example **tickets** and **actors** are index names:

```
hibernate.search.tickets.directory_provider=infinispan
hibernate.search.actors.directory_provider=filesystem
```

Lucene's **DirectoryProvider** uses the following options to configure the cache names:

- **locking_cachename** - Cache name where Lucene's locks are stored. Defaults to **LuceneIndexesLocking**.
- **data_cachename** - Cache name where Lucene's data is stored, including the largest data chunks and largest objects. Defaults to **LuceneIndexesData**.
- **metadata_cachename** - Cache name where Lucene's metadata is stored. Defaults to **LuceneIndexesMetadata**.

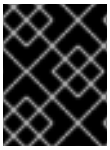
To adjust the name of the locking cache to **CustomLockingCache** use the following:

```
hibernate.search.default.directory_provider.locking_cachname="CustomLockingCache"
```

In addition, large files of the index are split into a smaller, configurable, chunk. It is often recommended to set the index's **chunk_size** to the highest value that may be handled efficiently by the network.

Hibernate Search already contains internally a default configuration which uses replicated caches to hold the indexes.

It is important that if more than one node writes to the index at the same time, configure a JMS backend. For more information on the configuration, see the Hibernate Search documentation.



IMPORTANT

In settings where distribution mode is needed to configure, the **LuceneIndexesMetadata** and **LuceneIndexesLocking** caches should always use replication mode in all the cases.

29.5. JMS MASTER AND SLAVE BACK END CONFIGURATION

While using an Infinispan directory, it is recommended to use the JMS Master/Slave backend. In Infinispan, all nodes share the same index and since **IndexWriter** is active on different nodes, it acquires the lock on the same index. So instead of sending updates directly to the index, send it to a JMS queue and make a single node apply all changes on behalf of all other nodes.

WARNING

Not enabling a JMS based backend will lead to timeout exceptions when multiple nodes attempt to write to the index.

To configure a JMS slave, replace only the backend and set the directory provider to Infinispan. Set the same directory provider on the master and it will connect without the need to set up the copy job across nodes.

For Master and Slave backend configuration examples, see the *Back End Setup and Operations* section of the Red Hat JBoss EAP *Developing Hibernate Applications* document.

CHAPTER 30. TRANSACTIONS

30.1. 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.

30.2. CONFIGURE TRANSACTIONS (LIBRARY MODE)

In Red Hat JBoss Data Grid, transactions in Library mode can be configured with synchronization and transaction recovery. Transactions in their entirety (which includes synchronization and transaction recovery) are not available in Remote Client-Server mode.

In 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.

In Library mode, transactions are configured as follows:

Configure Transactions in Library Mode (Programmatic Configuration)

1. Enable Transactions

```
Configuration config = new ConfigurationBuilder()/* ... */.transaction()
      .transactionMode(TransactionMode.TRANSACTIONAL)
      .transactionManagerLookup(new GenericTransactionManagerLookup())
      .lockingMode(LockingMode.OPTIMISTIC)
      .useSynchronization(true)
      .recovery()
         .recoveryInfoCacheName("anotherRecoveryCacheName").build();
```

   a. Set the transaction mode.

   b. Select and set a lookup class. See the table below this procedure for a list of available lookup classes.

   c. The **lockingMode** value determines whether optimistic or pessimistic locking is used. If the cache is non-transactional, the locking mode is ignored. The default value is **OPTIMISTIC**.

   d. The **useSynchronization** value configures the cache to register a synchronization with the transaction manager, or register itself as an XA resource. The default value is **true** (use synchronization).

   e. The **recovery** parameter enables recovery for the cache when set to **true**.

The **recoveryInfoCacheName** sets the name of the cache where recovery information is held. The default name of the cache is specified by **RecoveryConfiguration.DEFAULT_RECOVERY_INFO_CACHE**.

2. Configure Write Skew Check
   The **writeSkew** check determines if a modification to the entry from a different transaction should roll back the transaction. Write skew set to **true** requires **isolation_level** set to **REPEATABLE_READ**. The default value for **writeSkew** and **isolation_level** are **false** and **READ_COMMITTED** respectively.

```
Configuration config = new ConfigurationBuilder()/* ... */.locking()
      .isolationLevel(IsolationLevel.REPEATABLE_READ).writeSkewCheck(true);
```

3. Configure Entry Versioning
   For clustered caches, enable write skew check by enabling entry versioning and setting its value to **SIMPLE**.

```
Configuration config = new ConfigurationBuilder()/* ... */.versioning()
      .enable()
      .scheme(VersioningScheme.SIMPLE);
```

Table 30.1. Transaction Manager Lookup Classes

Class Name	Details
org.infinispan.transaction.lookup.DummyTransactionManagerLookup	Used primarily for testing environments. This testing transaction manager is not for use in a production environment and is severely limited in terms of functionality, specifically for concurrent transactions and recovery.
org.infinispan.transaction.lookup.JBossStandaloneJTAManagerLookup	The default transaction manager when Red Hat JBoss Data Grid runs in a standalone environment. It is a fully functional JBoss Transactions based transaction manager that overcomes the functionality limits of the **DummyTransactionManager**.
org.infinispan.transaction.lookup.GenericTransactionManagerLookup	GenericTransactionManagerLookup is used by default when no transaction lookup class is specified. This lookup class is recommended when using JBoss Data Grid with Java EE-compatible environment that provides a TransactionManager interface, and is capable of locating the Transaction Manager in most Java EE application servers. If no transaction manager is located, it defaults to **DummyTransactionManager**.

Class Name	Details
org.infinispan.transaction.lookup.JBossTransactionManagerLookup	The **JbossTransactionManagerLookup** finds the standard transaction manager running in the application server. This lookup class uses JNDI to look up the TransactionManager instance, and is recommended when custom caches are being used in JTA transactions.

NOTE

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.

30.3. 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.

30.4. 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:

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();
```

CHAPTER 31. MARSHALLING

31.1. 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.

31.2. ABOUT THE JBOSS MARSHALLING FRAMEWORK

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

31.3. 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

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

31.4. 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.

31.5. 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**.

WARNING

If developing your own custom marshaller, protect it from potential injection attacks by verifying that any class names read, before instantiating, are amongst the expected/allowed class names.

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

Define a Marshaller

1. Create a Configuration Builder
   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);
```

   a. 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.

31.6. RESTRICTING DESERIALIZATION TO SPECIFIC JAVA CLASSES

The Red Hat JBoss Data Grid server allows deserialization only for standard Java classes and primitives in addition to the Java classes that you specify in a whitelist.

Clients, on the other hand, can deserialize objects that belong to any Java class unless you restrict deserialization to specific classes. To do this, use the **addJavaSerialWhiteList** method in the **org.infinispan.client.hotrod.configuration.ConfigurationBuilder** class.

For example, to restrict deserialization to only Java classes with fully qualified names that contain either Person or Employee, specify the following configuration:

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

...
ConfigurationBuilder configBuilder = ...
configBuilder.addJavaSerialWhiteList(".*Person.*", ".*Employee.*");
```

For information on configuring the deserialization whitelist in the JBoss Data Grid server, see Configuring the Deserialization Whitelist in the Administration and Configuration Guide.

31.7. TROUBLESHOOTING

31.7.1. 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.

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(ReplicableCommandExternaliz
er.java:54)
at
org.infinispan.marshall.jboss.ConstantObjectTable$ExternalizerAdapter.writeObject(ConstantObjectTabl
e.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(VersionAwareMarshalle
rTest.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:

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:

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:10
4)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(VersionAwareMarshaller.java:17
7)
at
org.infinispan.marshall.VersionAwareMarshallerTest.testErrorUnmarshalling(VersionAwareMarshallerT
est.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:

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

31.7.2. 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:

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(StateTransferManagerImpl.java:1
75)
    at
org.infinispan.remoting.InboundInvocationHandlerImpl.generateState(InboundInvocationHandlerImpl.jav
a:119)
    at
org.infinispan.remoting.transport.jgroups.JGroupsTransport.getState(JGroupsTransport.java:586)
    at
org.jgroups.blocks.MessageDispatcher$ProtocolAdapter.handleUpEvent(MessageDispatcher.java:691)

    at org.jgroups.blocks.MessageDispatcher$ProtocolAdapter.up(MessageDispatcher.java:772)
    at org.jgroups.JChannel.up(JChannel.java:1465)
    at org.jgroups.stack.ProtocolStack.up(ProtocolStack.java:954)
    at org.jgroups.protocols.pbcast.FLUSH.up(FLUSH.java:478)
    at
org.jgroups.protocols.pbcast.STREAMING_STATE_TRANSFER$StateProviderHandler.process(STRE
AMING_STATE_TRANSFER.java:653)
    at
org.jgroups.protocols.pbcast.STREAMING_STATE_TRANSFER$StateProviderThreadSpawner$1.run(
STREAMING_STATE_TRANSFER.java:582)
    at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
    at java.lang.Thread.run(Thread.java:680)
Caused by: java.util.concurrent.TimeoutException: Could not obtain exclusive processing lock
    at
org.infinispan.remoting.transport.jgroups.JGroupsDistSync.acquireProcessingLock(JGroupsDistSync.jav
a:71)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.generateTransactionLog(StateTransferManager
Impl.java:202)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.generateState(StateTransferManagerImpl.java:1
65)
    ... 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:

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(AbstractUnmarshaller.java:184)
    at
org.jboss.marshalling.AbstractUnmarshaller.readUnsignedByteDirect(AbstractUnmarshaller.java:319)
    at org.jboss.marshalling.AbstractUnmarshaller.readUnsignedByte(AbstractUnmarshaller.java:280)
    at org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshaller.java:207)
```

```
    at org.jboss.marshalling.AbstractUnmarshaller.readObject(AbstractUnmarshaller.java:85)
    at
org.infinispan.marshall.jboss.GenericJBossMarshaller.objectFromObjectStream(GenericJBossMarshalle
r.java:175)
    at
org.infinispan.marshall.VersionAwareMarshaller.objectFromObjectStream(VersionAwareMarshaller.java
:184)
    at
org.infinispan.statetransfer.StateTransferManagerImpl.processCommitLog(StateTransferManagerImpl.j
ava: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:1
02)
    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.

CHAPTER 32. THE INFINISPAN CDI MODULE

32.1. 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.

32.2. USING INFINISPAN CDI

32.2.1. 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.

32.2.2. Set the CDI Maven Dependency

The CDI module is included in the Infinispan jar for each deployment type, and no additional dependencies are required.

Library Mode

In Library mode the **infinispan-embedded** artifact contains the CDI module, and should be added as a dependency as seen in the below example:

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

Remote Client-Server Mode

In Remote Client-Server mode the **infinispan-remote** artifact contains the CDI module, and should be added as a dependency as seen in the below example:

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

32.3. USING THE INFINISPAN CDI MODULE

32.3.1. Using the Infinispan CDI Module

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

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

32.3.2. Configure and Inject Infinispan Caches

32.3.2.1. Inject an Infinispan Cache

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

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

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

32.3.2.2. Inject a Remote Infinispan Cache

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

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

32.3.2.3. Set the Injection's Target Cache

32.3.2.3.1. 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.

32.3.2.3.2. Create a Qualifier Annotation

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

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.

32.3.2.3.3. Add a Producer Class

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

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.Produces;

public class CacheCreator {
   @ConfigureCache("smallcache")
   @SmallCache
   @Produces
   public Configuration specialCacheCfg() {
      return new ConfigurationBuilder()
                .memory()
                   .size(10)
                .build();
   }
}
```

The elements in the code snippet are:

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

32.3.2.3.4. Inject the Desired Class

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

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

32.3.3. Configure Cache Managers with CDI

32.3.3.1. Configure Cache Managers with CDI

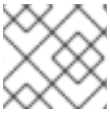
A Red Hat JBoss Data Grid Cache Manager (both embedded and remote) can be configured using CDI. Whether configuring an embedded or remote cache manager, the first step is to specify a default configuration that is annotated to act as a producer.

32.3.3.2. 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:

Specifying the Default Configuration

```
public class Config {
   @Produces
   public Configuration defaultEmbeddedConfiguration () {
      return new ConfigurationBuilder()
                  .memory()
                     size(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.

32.3.3.3. Override the Creation of the Embedded Cache Manager

Prerequisites

See Specify the Default Configuration.

Creating Non Clustered Caches

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

```
public class Config {

   @Produces
   @ApplicationScoped
   public EmbeddedCacheManager defaultEmbeddedCacheManager() {
      Configuration cfg = new ConfigurationBuilder()
                           .memory()
                              .size(150)
                           .build();
      return new DefaultCacheManager(cfg);
   }
}
```

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

Creating Clustered Caches

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

```
public class Config {

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

Invoke the Method to Generate an EmbeddedCacheManager

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

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

Generate an EmbeddedCacheManager

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

32.3.3.4. Configure a Remote Cache Manager

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

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);
 }
}}
```

32.3.3.5. Configure Multiple Cache Managers with a Single Class

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

Configure Multiple Cache Managers

```
public class Config {
   @Produces
   @ApplicationScoped
   public org.infinispan.manager.EmbeddedCacheManager
   defaultEmbeddedCacheManager() {
      Configuration cfg = new ConfigurationBuilder()
            .memory()
            .size(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()
            .memory()
            .size(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(POR
T).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_AW
AY).port(PORT).build();
      return new RemoteCacheManager(confid);
   }
}
```

32.4. STORAGE AND RETRIEVAL USING CDI ANNOTATIONS

32.4.1. Configure Cache Annotations

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

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

IMPORTANT

CDI is supported in both Remote Client-Server Mode and Library Mode; however, annotations such as @CachePut, @CacheRemove, @CacheRemoveAll, and @CacheResult cannot be used in Remote Client-Server Mode.

32.4.2. Enable Cache Annotations

JBoss Data Grid includes two sets of interceptors depending on how they are used. Interceptors can be added to the CDI bean archive using the *beans.xml* file.

Option 1: CDI Interceptors

Adding the following code adds interceptors such as the **InjectedCacheResultInterceptor**, **InjectedCachePutInterceptor**, **InjectedCacheRemoveEntryInterceptor** and the **InjectedCacheRemoveAllInterceptor**:

Adding CDI Interceptors

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

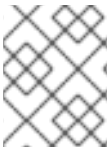
Option 2: JCache Interceptors

Adding the following code adds interceptors such as the **CacheResultInterceptor**, **CachePutInterceptor**, **CacheRemoveEntryInterceptor** and the **CacheRemoveAllInterceptor**:

Adding JCache 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.jcache.annotation.CacheResultInterceptor</class>
```

```
    <class>org.infinispan.jcache.annotation.CachePutInterceptor</class>
    <class>org.infinispan.jcache.annotation.CacheRemoveEntryInterceptor</class>
    <class>org.infinispan.jcache.annotation.CacheRemoveAllInterceptor</class>
  </interceptors>

</beans>
```



NOTE

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

32.4.3. Caching the Result of a Method Invocation

32.4.3.1. 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";
}
```

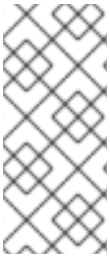
A common approach is to cache the results of this method call and to check the cache when the result is next required. The following is an example of a code snippet that looks up the result of such an operation in a cache. If the results are not found, the code snippet runs the **toCelsiusFormatted** 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.annotation.CacheResult
public String toCelsiusFormatted(float fahrenheit) {
 return NumberFormat.getInstance()
 .format((fahrenheit * 5 / 9) - 32)
 + " degrees Celsius";
}
```

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



NOTE

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

32.4.3.2. Specify the Cache Used

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

```
@CacheResult(cacheName = "mySpecialCache")
public String doSomething(String parameter) {
<!-- Additional configuration information here -->
}
```

32.4.3.3. Cache Keys for Cached Results

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

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

Create a Custom Key

```
@CacheResult
public String doSomething
 (@CacheKey String p1,
 @CacheKey String p2,
 String dontCare) {
<!-- Additional configuration information here -->
}
```

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.

32.4.3.4. Generate a Custom Key

Generate a custom key as follows:

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

public class MyCacheKeyGenerator implements CacheKeyGenerator {

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

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

```
        );
    }
}
```

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

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

```
@CacheResult(cacheKeyGenerator = MyCacheKeyGenerator.class)
public void doSomething(String p1, String p2) {
<!-- Additional configuration information here -->
}
```

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

32.4.4. Cache Operations

32.4.4.1. 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.

Sample @CachePut Annotated Method

```
import javax.cache.annotation.CachePut;
import javax.cache.annotation.CacheKey;
import javax.cache.annotation.CacheValue;

@CachePut (cacheName = "personCache")
public void updatePerson
 (@CacheKey long personId,
 @CacheValue Person newPerson) {
<!-- Additional configuration information here -->
 }
```

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: Cache keys for Cached Results

32.4.4.2. 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:

Removing an Entry from the Cache

```
import javax.cache.annotation.CacheRemoveEntry;
import javax.cache.annotation.CacheKey;

@CacheRemoveEntry (cacheName = "cacheOfPeople")
```

```
public void changePersonName
 (@CacheKey long personId,
 string newName) {
<!-- Additional configuration information here -->
 }
```

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

32.4.4.3. Clear the Cache

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

Clear All Entries from the Cache with @CacheRemoveAll

```
import javax.cache.annotation.CacheRemoveAll;

@CacheRemoveAll (cacheName = "statisticsCache")
public void resetStatistics() {
<!-- Additional configuration information here -->
}
```

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

CHAPTER 33. INTEGRATION WITH THE SPRING FRAMEWORK

Red Hat JBoss Data Grid provides integration with the Spring Framework through a set of modules that enable you to use JBoss Data Grid as a cache provider.

33.1. ENABLING SPRING CACHE SUPPORT

The first step to integrating Red Hat JBoss Data Grid with Spring is to enable cache support in the application context. This step lets you use the **@Cacheable** and **@CacheEvict** annotations for adding and removing entries from the cache.

33.1.1. Declaratively Enabling Spring Cache Support

To declaratively enable Spring cache support, add **<cache:annotation-driven/>** to the application context, as in the following example:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:cache="http://www.springframework.org/schema/cache"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/cache
http://www.springframework.org/schema/cache/spring-cache.xsd">

<cache:annotation-driven />
```

33.1.2. Programmatically Enabling Spring Cache Support

Programmatically enable Spring cache support as follows:

```
@EnableCaching @Configuration
public class Config {
}
```

33.2. ADDING THE SPRING INTEGRATION MODULE

Add the appropriate dependencies for Red Hat JBoss Data Grid and the Spring integration module to your **pom.xml** as follows:

Library mode

```
<dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-spring4-embedded</artifactId>
    <version>${version.spring}</version>
</dependency>
```

Remote Client-Server mode

```
<dependency>
   <groupId>org.infinispan</groupId>
   <artifactId>infinispan-spring4-remote</artifactId>
   <version>${version.spring}</version>
</dependency>
```

33.3. CONFIGURING RED HAT JBOSS DATA GRID AS THE SPRING CACHING PROVIDER

The Spring cache provider SPI has two interfaces through which it interacts with Red Hat JBoss Data Grid, **org.springframework.cache.CacheManager** and **org.springframework.cache.Cache**. The **CacheManager** interface acts as a factory for named **Cache** instances.

To use JBoss Data Grid acts as the caching provider, the Spring Framework requires a **CacheManager** implementation with a bean named **cacheManager** in the application context.

The following examples show how you can configure your application context either declaratively or programmatically:

33.3.1. Declaratively Configuring JBoss Data Grid as the Spring Caching Provider

Library mode

```
<bean id="cacheManager"
class="org.infinispan.spring.provider.SpringEmbeddedCacheManagerFactoryBean"
      p:configurationFileLocation="classpath:/path/to/cache-config.xml" />
```

Remote Client-Server mode

```
<bean id="cacheManager"
class="org.infinispan.spring.provider.SpringRemoteCacheManagerFactoryBean"
      p:configurationFileLocation="classpath:/path/to/hotrod-client.properties" />
```

33.3.2. Programmatically Configuring JBoss Data Grid as the Spring Caching Provider

Library mode

```
@EnableCaching
@Configuration
public class Config {

  @Bean
  public CacheManager cacheManager() {
    return new SpringEmbeddedCacheManager(infinispanCacheManager());
  }

  private EmbeddedCacheManager infinispanCacheManager() {
    return new DefaultCacheManager();
```

```
    }

}
```

Remote Client-Server mode

```
@EnableCaching
@Configuration
public class Config {

  @Bean
  public CacheManager cacheManager() {
    return new SpringRemoteCacheManager(infinispanCacheManager());
  }

  private RemoteCacheManager infinispanCacheManager() {
    return new DefaultCacheManager();
  }

}
```

33.4. ADDING CACHING TO YOUR APPLICATION CODE

You can add caching to your application with Spring annotations.

Adding Cache Entries

To add entries to the cache add the **@Cacheable** annotation to select methods. This annotation will add any returned values to the indicated cache. For instance, consider a method that returns a **Book** based on a particular key.

By annotating this method with **@Cacheable**:

```
@Transactional
@Cacheable(value = "books", key = "#bookId")
public Book findBook(Integer bookId) {...}
```

Any **Book** instances returned from **findBook(Integer bookId)** will be placed in a named cache **books**, using the **bookId** as the value's key.

IMPORTANT

If the key attribute is not specified then Spring will generate a hash from the supplied arguments and use this generated value as the cache key. If your application needs to reference the entries directly it is recommended to include the key attribute so that entries may be easily obtained.

Deleting Cache Entries

To remove entries from the cache annotate the desired methods with **@CacheEvict**. This annotation can be configured to evict all entries in a cache, or to only affect entries with the indicated key. Consider the following examples:

```
// Evict all entries in the "books" cache
```

```
@Transactional
@CacheEvict (value="books", key = "#bookId", allEntries = true)
public void deleteBookAllEntries() {...}

// Evict any entries in the "books" cache that match the passed in bookId
@Transactional
@CacheEvict (value="books", key = "#bookId")
public void deleteBook(Integer bookId) {...]}
```

33.5. CONFIGURING TIMEOUTS FOR CACHE OPERATIONS

The Red Hat JBoss Data Grid Spring Cache provider defaults to blocking behaviour when performing read and write operations. By default operations are synchronous and do not time out. However, you might want to set a maximum time to wait for operations before timing out in some situations. For example, timeouts are useful if you need to ensure that an operation completes within a certain time and you can ignore the cached value.

The following properties let you set timeouts for read and write operations:

- **infinispan.spring.operation.read.timeout** specifies the time, in milliseconds, to wait for read operations to complete. The default is **0** which means unlimited wait time.
- **infinispan.spring.operation.write.timeout** specifies the time, in milliseconds, to wait for write operations to complete. The default is **0** which means unlimited wait time.

To configure timeouts for cache operations, set the properties in the context XML for your application on either **SpringEmbeddedCacheManagerFactoryBean** or **SpringRemoteCacheManagerFactoryBean** as follows:

Library mode

```
<bean id="springEmbeddedCacheManagerConfiguredUsingConfigurationProperties"
class="org.infinispan.spring.provider.SpringEmbeddedCacheManagerFactoryBean">
   <property name="configurationProperties">
      <props>
         <prop key="infinispan.spring.operation.read.timeout">500</prop>
         <prop key="infinispan.spring.operation.write.timeout">700</prop>
      </props>
   </property>
</bean>
```

Remote Client-Server mode

```
<bean id="springRemoteCacheManagerConfiguredUsingConfigurationProperties"
class="org.infinispan.spring.provider.SpringRemoteCacheManagerFactoryBean">
   <property name="configurationProperties">
      <props>
         <prop key="infinispan.spring.operation.read.timeout">500</prop>
         <prop key="infinispan.spring.operation.write.timeout">700</prop>
      </props>
   </property>
</bean>
```

NOTE

In remote client-server mode you can also set these properties in **hotrod-client.properties**.

33.6. EXTERNALIZING SESSIONS TO RED HAT JBOSS DATA GRID CLUSTERS

Spring Session lets you externalize user session information to JBoss Data Grid in both library mode and remote client-server mode.

To configure Spring Session integration in your application, do the following:

1. Add the following dependencies to your **pom.xml**:

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-session</artifactId>
   <version>${version.spring}</version>
</dependency>
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-web</artifactId>
   <version>${version.spring}</version>
</dependency>
```

2. Specify the appropriate FactoryBean to expose a CacheManager instance.
   - Library mode: **SpringEmbeddedCacheManagerFactoryBean**
   - Remote Client-Server mode: **SpringRemoteCacheManagerFactoryBean**
3. Enable Spring Session with the appropriate annotation.
   - Library mode: **@EnableInfinispanEmbeddedHttpSession**
   - Remote Client-Server mode: **@EnableInfinispanRemoteHttpSession**
     These annotations have the following optional parameters:
     - **maxInactiveIntervalInSeconds** sets session expiration time in seconds. The default is **1800**.
     - **cacheName** specifies the name of the cache that stores sessions. The default is **sessions**.

The following provides an example configuration for JBoss Data Grid in library mode:

```
@EnableInfinispanEmbeddedHttpSession
@Configuration
public class Config {

  @Bean
  public SpringEmbeddedCacheManagerFactoryBean springCacheManager() {
    return new SpringEmbeddedCacheManagerFactoryBean();
  }
```

```
    //An optional configuration bean that replaces the default cookie
    //for obtaining configuration.
    //For more information refer to Spring Session documentation.
    @Bean
    public HttpSessionStrategy httpSessionStrategy() {
      return new HeaderHttpSessionStrategy();
    }
}
```

CHAPTER 34. INTEGRATION WITH APACHE SPARK

34.1. THE JBOSS DATA GRID APACHE SPARK CONNECTOR

JBoss Data Grid includes a Spark connector, providing tight integration with Apache Spark, and allowing applications written either in Java or Scala to utilize JBoss Data Grid as a backing data store.

There actually are two connectors, one that supports Apache Spark 1.6.x, and one that supports Apache Spark 2.x, which in turn support Scala 2.10.x, and 2.11.x, respectively. Both of these connectors are shipped separately from the main distribution.

The Apache Spark 1.6 connector includes support for the following:

- Create an RDD from any cache
- Write a key/value RDD to a cache
- Create a DStream from cache-level events
- Write a key/value DStream to a cache

In addition to the above features, the Apache Spark 2 connector supports these features:

- Use JDG server side filters to create a cache based RDD
- Spark serializer based on JBoss Marshalling
- Dataset API with push down predicates support

NOTE

Support for Apache Spark is only available in Remote Client-Server Mode.

34.2. SPARK DEPENDENCIES

The following Maven configuration should be used depending on the desired version of Apache Spark:

pom.xml for Spark 1.6.x

```
<dependency>
   <groupId>org.infinispan</groupId>
   <artifactId>infinispan-spark</artifactId>
   <version>0.3.0.Final-redhat-2</version>
</dependency>
```

pom.xml for Spark 2.x

```
<dependency>
   <groupId>org.infinispan</groupId>
   <artifactId>infinispan-spark</artifactId>
   <version>0.6.0.Final-redhat-9</version>
</dependency>
```

34.3. CONFIGURING THE SPARK CONNECTOR

The Apache Spark version 1.6 and version 2 connectors do not use the same interfaces for configuration. The version 1.6 connector uses properties, and the version 2 connector uses methods.

34.3.1. Properties to Configure the Version 1.6 Connector

Table 34.1. Properties to Configure the Version 1.6 Connector

Property Name	Description	Default Value
infinispan.client.hotrod.server_list	List of JBoss Data Grid nodes	localhost:11222
infinispan.rdd.cacheName	The name of the cache that will back the RDD	default cache
infinispan.rdd.read_batch_size	Batch size (number of entries) when reading from the cache	10000
infinispan.rdd.write_batch_size	Batch size (number of entries) when writing to the cache	500
infinispan.rdd.number_server_partitions	Numbers of partitions created per JBoss Data Grid server	2
infinispan.rdd.query.proto.protofiles	Map with protobuf file names and contents	Can be omitted if entities are annotated with protobuf encoding information. Protobuf encoding is required to filter the RDD by Query.
infinispan.rdd.query.proto.marshallers	List of protostream marshallers classes for the objects in the cache	Can be omitted if entities are annotated with protobuf encoding information. Protobuf encoding is required to filter the RDD by Query.

34.3.2. Methods to Configure the Version 2 Connector

The following methods can be used to configure the version 2 connector. They are provided by the **org.infinispan.spark.config.ConnectorConfiguration** class.

Table 34.2. Methods to Configure the Version 2 Connector

Method Name	Description	Default Value
setServerList(String)	List of JBoss Data Grid nodes	localhost:11222

Method Name	Description	Default Value
setCacheName(String)	The name of the cache that will back the RDD	default cache
setReadBatchSize(Integer)	Batch size (number of entries) when reading from the cache	10000
setWriteBatchSize(Integer)	Batch size (number of entries) when writing to the cache	500
setPartitions(Integer)	Numbers of partitions created per JBoss Data Grid server	2
addProtoFile(String name, String contents)	Map with protobuf file names and contents	Can be omitted if entities are annotated with protobuf encoding information. Protobuf encoding is required to filter the RDD by Query.
addMessageMarshaller(Class)	List of protostream marshallers classes for the objects in the cache	Can be omitted if entities are annotated with protobuf encoding information. Protobuf encoding is required to filter the RDD by Query.
addProtoAnnotatedClass(Class)	Registers a Class containing protobuf annotations	Alternative to using **addProtoFile** and **addMessageMarshaller** methods, since both will be auto-generated based on the annotations.
setAutoRegisterProto()	Causes automatic registration of protobuf schemas in the server	None
addHotRodClientProperty(key, value)	Configures additional Hot Rod client properties when contacting the JBoss Data Grid Server	None
setTargetEntity(Class)	Used in conjunction with the Dataset API to specify the Query target	If omitted, and in case there is only one class annotated with protobuf configured, it will choose that class.

34.3.3. Connecting to a Secured JDG Cluster

If the JDG cluster is secured, Hot Rod must be configured with security for the Spark connector to work. There are several Hot Rod properties that can be set to configure Hot Rod security. They are described in the Hot Rod Properties table in the Appendix of the Administration and Configuration Guide, starting with **infinispan.client.hotrod.use_ssl**

IMPORTANT

These properties are exclusive to the version 2 Apache Spark connector.

34.4. CODE EXAMPLES FOR SPARK 1.6

34.4.1. Code Examples for Spark 1.6

Since the connector for Apache Spark 1.6 uses a different configuration mechanism than the version 2 connector, and also because the version 2 connector supports some features 1.6 doesn't, the code examples for each version are separated into their own sections. The following code examples work with version 1.6 of the Spark connector. Follow this link for code examples for Spark 2.

34.4.2. Creating and Using RDDs

With the Apache Spark 1.6 connector, RDDs are created by specifying a **Properties** instance with configurations described in Properties to Configure the Version 1.6 Connector , and then using it together with the Spark context to create a **InfinispanRDD** that is used with the normal Spark operations.

An example of this is below in both Java and Scala:

34.4.3. Creating an RDD

Creating an RDD (Java)

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.infinispan.spark.rdd.InfinispanJavaRDD;
import java.util.Properties;
[...]
// Begin by defining a new Spark configuration and creating a Spark context from this.
SparkConf conf = new SparkConf().setAppName("example-RDD");
JavaSparkContext jsc = new JavaSparkContext(conf);

// Create the Properties instance, containing the JBoss Data Grid node and cache name.
Properties properties = new Properties();
properties.put("infinispan.client.hotrod.server_list", "server:11222");
properties.put("infinispan.rdd.cacheName","exampleCache");

// Create the RDD
JavaPairRDD<Integer, Book> exampleRDD = InfinispanJavaRDD.createInfinispanRDD(jsc,
properties);

JavaRDD<Book> booksRDD = exampleRDD.values();
```

Creating an RDD (Scala)

```
import java.util.Properties

import org.apache.spark.{SparkConf, SparkContext}
```

```
import org.infinispan.spark.rdd.InfinispanRDD
import org.infinispan.spark._

// Begin by defining a new Spark configuration and creating a Spark context from this.
val conf = new SparkConf().setAppName("example-RDD-scala")
val sc = new SparkContext(conf)

// Create the Properties instance, containing the JBoss Data Grid node and cache name.
val properties = new Properties
properties.put("infinispan.client.hotrod.server_list", "server:11222")
properties.put("infinispan.rdd.cacheName", "exampleCache")

// Create an RDD from the DataGrid cache
val exampleRDD = new InfinispanRDD[Integer, Book](sc, properties)

val booksRDD = exampleRDD.values
```

34.4.4. Querying an RDD

Once the RDD is available entries in the backing cache may be obtained by using either the Spark RDD operations or Spark's SQL support. The above example is expanded to count the entries, per author, in the resulting RDD with an SQL query:

Querying an RDD (Java)

```
// The following imports should be added to the list from the previous example
import org.apache.spark.sql.DataFrame;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SQLContext;
[...]
// Continuing the previous example

// Create a SQLContext, registering the data frame and table
SQLContext sqlContext = new SQLContext(jsc);
DataFrame dataFrame = sqlContext.createDataFrame(booksRDD, Book.class);
dataFrame.registerTempTable("books");

// Run the Query and collect results
List<Row> rows = sqlContext.sql("SELECT author, count(*) as a from books WHERE author != 'N/A'
GROUP BY author ORDER BY a desc").collectAsList();
```

Querying an RDD (Scala)

```
import org.apache.spark.SparkContext
import org.apache.spark.sql.SQLContext
[...]
// Create a SQLContext, register a data frame and table
val sqlContext = new SQLContext(sc)
val dataFrame = sqlContext.createDataFrame(booksRDD, classOf[Book])
dataFrame.registerTempTable("books")

 // Run the Query and collect the results
val rows = sqlContext.sql("SELECT author, count(*) as a from books WHERE author != 'N/A' GROUP
BY author ORDER BY a desc").collect()
```

34.4.5. Writing an RDD to the Cache

Any key/value based RDD can be written to the Data Grid cache by using the static **InfinispanJavaRDD.write()** method. This will copy the contents of the RDD to the cache:

Writing an RDD (Java)

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.spark.domain.Address;
import org.infinispan.spark.domain.Person;
import org.infinispan.spark.rdd.InfinispanJavaRDD;
import scala.Tuple2;
import java.util.List;
import java.util.Properties;
[...]
// Define the location of the JBoss Data Grid node
Properties properties = new Properties();
properties.put("infinispan.client.hotrod.server_list", "localhost:11222");
properties.put("infinispan.rdd.cacheName","exampleCache");

// Create the JavaSparkContext
SparkConf conf = new SparkConf().setAppName("write-example-RDD");
JavaSparkContext jsc = new JavaSparkContext(conf);

// Defining two entries to be stored in a RDD
// Each Book will contain the title, author, and publicationYear
Book bookOne = new Book("Linux Bible", "Negus, Chris", "2015");
Book bookTwo = new Book("Java 8 in Action", "Urma, Raoul-Gabriel", "2014");

List<Tuple2<Integer, Book>> pairs = Arrays.asList(
   new Tuple2<>(1, bookOne),
   new Tuple2<>(2, bookTwo)
);

// Create the RDD using the newly created List
JavaPairRDD<Integer, Book> pairsRDD = jsc.parallelizePairs(pairs);

// Write the entries into the cache
InfinispanJavaRDD.write(pairsRDD, config);
```

Writing an RDD (Scala)

```
import java.util.Properties
import org.infinispan.spark._
import org.infinispan.spark.rdd.InfinispanRDD
[...]
// Define the location of the JBoss Data Grid node
val properties = new Properties
properties.put("infinispan.client.hotrod.server_list", "localhost:11222")
properties.put("infinispan.rdd.cacheName", "exampleCache")

// Create the SparkContext
```

```
val conf = new SparkConf().setAppName("write-example-RDD-scala")
val sc = new SparkContext(conf)

// Create an RDD of Books
val bookOne = new Book("Linux Bible", "Negus, Chris", "2015")
val bookTwo = new Book("Java 8 in Action", "Urma, Raoul-Gabriel", "2014")

val sampleBookRDD = sc.parallelize(Seq(bookOne,bookTwo))
val pairsRDD = sampleBookRDD.zipWithIndex().map(_.swap)

// Write the Key/Value RDD to the Data Grid
pairsRDD.writeToInfinispan(properties)
```

34.4.5.1. Creating and Using DStreams

DStreams represent a continuous stream of data, and are internally represented by a continuous series of RDDs, with each RDD containing data from a specific time interval.

To create a DStream a **StreamingContext** will be passed in along with **StorageLevel** and the JBoss Data Grid RDD configuration, as seen in the below example:

Creating a DStream (Scala)

```
import org.infinispan.spark.stream._
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.spark.storage.StorageLevel
import java.util.Properties

// Spark context
val sc = ...
// java.util.Properties with Infinispan RDD configuration
val props = ...
val ssc = new StreamingContext(sc, Seconds(1))

val stream = new InfinispanInputDStream[String, Book](ssc, StorageLevel.MEMORY_ONLY, props)
```

The **InfinispanInputDStream** can be transformed using the many Spark's DStream operations, and the processing will occur after calling "start" in the **StreamingContext**. For example, to display every 10 seconds the number of books inserted in the cache in the last 30 seconds:

Processing a DStream (Scala)

```
import org.infinispan.spark.stream._

val stream = ... // From previous sample

// Filter only created entries
val createdBooksRDD = stream.filter { case (_, _, t) => t ==
Type.CLIENT_CACHE_ENTRY_CREATED }

// Reduce last 30 seconds of data, every 10 seconds
val windowedRDD: DStream[Long] = createdBooksRDD.count().reduceByWindow(_ + _,
Seconds(30), Seconds(10))

// Prints the results, couting the number of occurences in each individual RDD
```

```
windowedRDD.foreachRDD { rdd => println(rdd.reduce(_ + _)) }

// Start the processing
ssc.start()
ssc.awaitTermination()
```

Writing to JBoss Data Grid with DStreams

Any DStream of Key/Value type can be written to JBoss Data Grid through the **InfinispanJavaDStream.writeToInfinispan()** Java method or in Scala using the implicit **writeToInfinispan(properties)** method directly on the DStream instance. Both methods take the JBoss Data Grid RDD configuration as input and will write each RDD contained within the DStream

34.4.6. Using the Infinispan Query DSL with Spark

The Infinispan Query DSL can be used as a filter for the InfinispanRDD, allowing data to be pre-filtered at the source rather than at RDD level.

IMPORTANT

Data in the cache must have been encoded with protobuf for the querying DSL to function correctly. Instructions on using protobuf encoding are found in Protobuf Encoding.

Consider the following example which retrieves a list of books that includes any author whose name contains **Doe**:

34.4.7. Filtering by a Query

Filtering by a Query (Scala)

```
import org.infinispan.client.hotrod.impl.query.RemoteQuery
import org.infinispan.client.hotrod.{RemoteCacheManager, Search}
import org.infinispan.spark.domain._
[...]
val query = Search.getQueryFactory(remoteCacheManager.getCache(getCacheName))
    .from(classOf[Book])
    .having("author").like("Doe")
    .toBuilder[RemoteQuery].build()

val rdd = createInfinispanRDD[Int, Book]
    .filterByQuery[Book]](query, classOf[Book])
```

Projections are also fully supported; for instance, the above example may be adjusted to only obtain the title and publication year, and sorting on the latter field:

34.4.8. Filtering with a Projection

Filtering with a Projection (Scala)

```
import org.infinispan.client.hotrod.impl.query.RemoteQuery
import org.infinispan.client.hotrod.{RemoteCacheManager, Search}
import org.infinispan.spark.domain._
```

```
[...]
val query = Search.getQueryFactory(remoteCacheManager.getCache(getCacheName))
   .select("title","publicationYear")
   .from(classOf[Book])
   .having("author").like("Doe")
   .groupBy("publicationYear")
   .toBuilder[RemoteQuery].build()

val rdd = createInfinispanRDD[Int, Book]
   .filterByQuery[Array[AnyRef]](query, classOf[Book])
```

In addition, if a filter has already been deployed to the JBoss Data Grid server it may be referenced by name, as seen below:

34.4.9. Filtering with a Deployed Filter

Filtering with a Deployed Filter (Scala)

```
val rdd = InfinispanRDD[String,Book] = ....
// "my-filter-factory" filter and converts Book to a String, and has two parameters
val filteredRDD = rdd.filterByCustom[String]("my-filter-factory", "param1", "param2")
```

34.5. CODE EXAMPLES FOR SPARK 2

34.5.1. Code Examples for Spark 2

Since the connector for Apache Spark 2 uses a different configuration mechanism than the 1.6 connector, and also because the version 2 connector supports some features 1.6 doesn't, the code examples for each version are separated into their own sections. The following code examples work with version 2 of the Spark connector. Follow this link for code examples for Spark 1.6 .

34.5.2. Creating and Using RDDs

In Apache Spark 2, Resilient Distributed Datasets (RDDs) are created by specifying a **ConnectorConfiguration** instance with configurations described in the table from Methods to Configure the Version 2 Connector.

Examples of this in both Java and Scala are below:

34.5.3. Creating an RDD

Creating an RDD (Java)

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.rdd.InfinispanJavaRDD;

JavaSparkContext jsc = new JavaSparkContext();

ConnectorConfiguration config = new ConnectorConfiguration()
```

```
            .setCacheName("exampleCache").setServerList("server:11222");

JavaPairRDD<String, MyEntity> infinispanRDD = InfinispanJavaRDD.createInfinispanRDD(jsc,
config);

JavaRDD<MyEntity> entitiesRDD = infinispanRDD.values();
```

Creating an RDD (Scala)

```
import org.apache.spark.SparkContext
import org.infinispan.spark.config.ConnectorConfiguration
import org.infinispan.spark.rdd.InfinispanRDD

val sc: SparkContext = new SparkContext()

val config = new ConnectorConfiguration().setCacheName("my-
cache").setServerList("10.9.0.8:11222")

val infinispanRDD = new InfinispanRDD[String, MyEntity](sc, config)

val entitiesRDD = infinispanRDD.values
```

34.5.4. Querying an RDD

34.5.4.1. SparkSQL Queries

Using SparkSQL Queries (Java)

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.rdd.InfinispanJavaRDD;

JavaSparkContext jsc = new JavaSparkContext();

ConnectorConfiguration conf = new ConnectorConfiguration();

// Obtain the values from an InfinispanRDD
JavaPairRDD<Long, MyEntity> infinispanRDD = InfinispanJavaRDD.createInfinispanRDD(jsc, conf);

JavaRDD<MyEntity> valuesRDD = infinispanRDD.values();

// Create a DataFrame from a SparkSession
SparkSession sparkSession = SparkSession.builder().config(new
SparkConf().setMaster("masterHost")).getOrCreate();
Dataset<Row> dataFrame = sparkSession.createDataFrame(valuesRDD, MyEntity.class);

// Create a view
dataFrame.createOrReplaceTempView("myEntities");
```

```
// Create and run the Query
Dataset<Row> rows = sparkSession.sql("SELECT field1, count(*) as c from myEntities WHERE
field1 != 'N/A' GROUP BY field1 ORDER BY c desc");
```

Using SparkSQL Queries (Scala)

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.{SparkConf, SparkContext}
import org.infinispan.spark.config.ConnectorConfiguration
import org.infinispan.spark.rdd._

val sc: SparkContext = // Initialize your SparkContext here

val config = new ConnectorConfiguration().setServerList("myserver1:port,myserver2:port")

// Obtain the values from an InfinispanRDD
val infinispanRDD = new InfinispanRDD[Long, MyEntity](sc, config)
val valuesRDD = infinispanRDD.values

// Create a DataFrame from a SparkSession
val sparkSession = SparkSession.builder().config(new
SparkConf().setMaster("masterHost")).getOrCreate()
val dataFrame = sparkSession.createDataFrame(valuesRDD, classOf[MyEntity])

// Create a view
dataFrame.createOrReplaceTempView("myEntities")

// Create and run the Query, collect and print results
sparkSession.sql("SELECT field1, count(*) as c from myEntities WHERE field1 != 'N/A' GROUP BY
field1 ORDER BY c desc")
                .collect().take(20).foreach(println)
```

34.5.5. Writing an RDD to the Cache

Any key/value based RDD can be written to the Data Grid cache by using the static **InfinispanJavaRDD.write()** method. This will copy the contents of the RDD to the cache:

Writing an RDD (Java)

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.rdd.InfinispanJavaRDD;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;

JavaSparkContext jsc = new JavaSparkContext();

ConnectorConfiguration connectorConfiguration = new ConnectorConfiguration();

List<Integer> numbers = IntStream.rangeClosed(1, 1000).boxed().collect(Collectors.toList());
```

```
JavaPairRDD<Integer, Long> numbersRDD = jsc.parallelize(numbers).zipWithIndex();

InfinispanJavaRDD.write(numbersRDD, connectorConfiguration);
```

Writing an RDD (Scala)

```
import org.apache.spark.SparkContext
import org.infinispan.spark._
import org.infinispan.spark.config.ConnectorConfiguration

val config: ConnectorConfiguration = // Initialize your ConnectorConfiguration here
val sc: SparkContext = // Initialize your SparkContext here

val simpleRdd = sc.parallelize(1 to 1000).zipWithIndex()
simpleRdd.writeToInfinispan(config)
```

34.5.6. Creating DStreams

DStreams represent a continuous stream of data, and are internally represented by a continuous series of RDDs, with each RDD containing data from a specific time interval.

To create a DStream a **StreamingContext** will be passed in along with **StorageLevel** and the JBoss Data Grid RDD configuration, as seen in the below example:

Creating a DStream (Java)

```
import org.apache.spark.SparkConf;
import org.apache.spark.streaming.Seconds;
import org.apache.spark.streaming.api.java.JavaStreamingContext;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.stream.InfinispanJavaDStream;
import static org.apache.spark.storage.StorageLevel.MEMORY_ONLY;

SparkConf conf = new SparkConf().setAppName("my-stream-app");

ConnectorConfiguration configuration = new ConnectorConfiguration();

JavaStreamingContext jsc = new JavaStreamingContext(conf, Seconds.apply(1));

InfinispanJavaDStream.createInfinispanInputDStream(jsc, MEMORY_ONLY(), configuration);
```

Creating a DStream (Scala)

```
import org.apache.spark.SparkContext
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.infinispan.spark.config.ConnectorConfiguration
import org.infinispan.spark.stream._

val sc = new SparkContext()
val config = new ConnectorConfiguration()
val ssc = new StreamingContext(sc, Seconds(1))
val stream = new InfinispanInputDStream[String, MyEntity](ssc, StorageLevel.MEMORY_ONLY,
config)
```

Writing to JBoss Data Grid with DStreams

Any DStream of Key/Value type can be written to JBoss Data Grid through the **InfinispanJavaDStream.writeToInfinispan()** Java method or in Scala using the implicit **writeToInfinispan(properties)** method directly on the DStream instance. Both methods take the JBoss Data Grid RDD configuration as input and will write each RDD contained within the DStream.

34.5.7. Using The Apache Spark Dataset API

In addition to the Resilient Distributed Dataset (RDD) programming interface, JBoss Data Grid includes the Apache Spark Dataset API, with support for pushing down predicates, similar to **rdd.filterByQuery**.

Dataset API Example (Java)

```
import org.apache.spark.SparkConf;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SparkSession;
import org.infinispan.spark.config.ConnectorConfiguration;

import java.util.List;

// Configure the connector using the ConnectorConfiguration: register entities annotated with
Protobuf,
// and turn on automatic registration of schemas
ConnectorConfiguration connectorConfig = new ConnectorConfiguration()
        .setServerList("server1:11222,server2:11222")
        .addProtoAnnotatedClass(User.class)
        .setAutoRegisterProto();

// Create the SparkSession
SparkSession sparkSession = SparkSession.builder().config(new
SparkConf().setMaster("masterHost")).getOrCreate();

// Load the "infinispan" datasource into a DataFame, using the infinispan config
Dataset<Row> df =
sparkSession.read().format("infinispan").options(connectorConfig.toStringsMap()).load();

// From here it's possible to query using the DatasetSample API...
List<Row> rows = df.filter(df.col("age").gt(30)).filter(df.col("age").lt(40)).collectAsList();

// ... or execute SQL queries
df.createOrReplaceTempView("user");
String query = "SELECT first(r.name) as name, first(r.age) as age FROM user u GROUP BY r.age";
List<Row> results = sparkSession.sql(query).collectAsList();
```

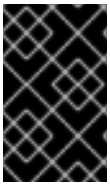
Dataset API Example (Scala)

```
import org.apache.spark._
import org.apache.spark.sql._
import org.infinispan.protostream.annotations.{ProtoField, ProtoMessage}
import org.infinispan.spark.config.ConnectorConfiguration

import scala.annotation.meta.beanGetter
import scala.beans.BeanProperty
```

```
  // Entities can be annotated in order to automatically generate protobuf schemas.
  // Also, they should be valid java beans. From Scala this can be achieved as:

@ProtoMessage(name = "user")
class User(@(ProtoField@beanGetter)(number = 1, required = true) @BeanProperty var name:
String,
          @(ProtoField@beanGetter)(number = 2, required = true) @BeanProperty var age: Int) {
  def this() = {
      this(name = "", age = -1)
  }
}

// Configure the connector using the ConnectorConfiguration: register entities annotated with
Protobuf,
// and turn on automatic registration of schemas
val infinispanConfig: ConnectorConfiguration = new ConnectorConfiguration()
    .setServerList("server1:11222,server2:11222")
    .addProtoAnnotatedClass(classOf[User])
    .setAutoRegisterProto()

// Create the SparkSession
val sparkSession = SparkSession.builder().config(new
SparkConf().setMaster("masterHost")).getOrCreate()

// Load the "infinispan" datasource into a DataFame, using the infinispan config
val df = sparkSession.read.format("infinispan").options(infinispanConfig.toStringsMap).load()

// From here it's possible to query using the DatasetSample API...
val rows: Array[Row] = df.filter(df("age").gt(30)).filter(df("age").lt(40)).collect()

// ... or execute SQL queries
df.createOrReplaceTempView("user")
val query = "SELECT first(r.name) as name, first(r.age) as age FROM user u GROUP BY r.age"
val rowsFromSQL: Array[Row] = sparkSession.sql(query).collect()
```

34.5.8. Using the Infinispan Query DSL with Spark

The Infinispan Query DSL can be used as a filter for the InfinispanRDD, allowing data to be pre-filtered at the source rather than at RDD level.



IMPORTANT

Data in the cache must have been encoded with protobuf for the querying DSL to function correctly. Instructions on using protobuf encoding are found in Protobuf Encoding.

34.5.9. Filtering with a pre-built Query Object

Filtering with a pre-built Query Object (Java)

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
```

```
import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.Search;
import org.infinispan.query.dsl.Query;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.rdd.InfinispanJavaRDD;

JavaSparkContext jsc = new JavaSparkContext();

ConnectorConfiguration conf = new ConnectorConfiguration();

InfinispanJavaRDD<String, MyEntity> infinispanRDD = InfinispanJavaRDD.createInfinispanRDD(jsc,
conf);

RemoteCache<String, MyEntity> remoteCache = new RemoteCacheManager().getCache();

// Assuming MyEntity is already stored in the cache with protobuf encoding, and has protobuf
annotations.
Query query =
Search.getQueryFactory(remoteCache).from(MyEntity.class).having("field").equal("value").build();

JavaPairRDD<String, MyEntity> filtered = infinispanRDD.filterByQuery(query);
```

Filtering with a pre-built Query Object (Scala)

```
import org.infinispan.client.hotrod.{RemoteCache, Search}
import org.infinispan.spark.rdd.InfinispanRDD

val rdd: InfinispanRDD[String, MyEntity] = // Initalize your InfinispanRDD here
val cache: RemoteCache[String, MyEntity] = // Initalize your RemoteCache here

// Assuming MyEntity is already stored in the cache with protobuf encoding, and has protobuf
annotations.
val query =
Search.getQueryFactory(cache).from(classOf[MyEntity]).having("field").equal("value").build()

val filteredRDD = rdd.filterByQuery(query)
```

34.5.10. Filtering with an Ickle Query

Filtering with an Ickle Query (Java)

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.rdd.InfinispanJavaRDD;

JavaSparkContext jsc = new JavaSparkContext();
ConnectorConfiguration conf = new ConnectorConfiguration();

InfinispanJavaRDD<String, MyEntity> infinispanRDD = InfinispanJavaRDD.createInfinispanRDD(jsc,
conf);
```

```
JavaPairRDD<String, MyEntity> filtered = infinispanRDD.filterByQuery("From myEntity where field =
'value'");
```

Filtering with an Ickle Query (Scala)

```
import org.infinispan.spark.rdd.InfinispanRDD

val rdd: InfinispanRDD[String, MyEntity] = // Initialize your InfinispanRDD here
val filteredRDD = rdd.filterByQuery("FROM MyEntity e where e.field BETWEEN 10 AND 20")
```

34.5.11. Filtering on the Server

Filtering on the Server (Java)

```
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.infinispan.spark.config.ConnectorConfiguration;
import org.infinispan.spark.rdd.InfinispanJavaRDD;

JavaSparkContext jsc = new JavaSparkContext();

ConnectorConfiguration conf = new ConnectorConfiguration();

InfinispanJavaRDD<String, MyEntity> infinispanRDD = InfinispanJavaRDD.createInfinispanRDD(jsc,
conf);

JavaPairRDD<String, MyEntity> filtered = infinispanRDD.filterByCustom("my-filter", "param1",
"param2");
```

Filtering on the Server (Scala)

```
import org.infinispan.spark.rdd.InfinispanRDD

val rdd: InfinispanRDD[String, MyEntity] = // Initalize your InfinispanRDD here
// "my-filter-factory" filter and converts MyEntity to a Double, and has two parameters
val filteredRDD = rdd.filterByCustom[Double]("my-filter-factory", "param1", "param2")
```

34.6. SPARK PERFORMANCE CONSIDERATIONS

The Data Grid Spark connector creates by default two partitions per each Data Grid node, each partition specifies a subset of the data in that particular node.

Those partitions are then sent to the Spark workers that will process them in parallel. If the number of Spark workers is less than the number of Data Grid nodes, some delay can occur since each worker has a maximum capacity of executing tasks in parallel. For this reason it is recommended to have at least the same number of Spark workers as Data Grid nodes to take advantage of the parallelism.

In addition, if a Spark worker is co-located in the same node as the Data Grid node, the connector will distribute tasks so that each worker only processes data found in the local Data Grid node.

CHAPTER 35. INTEGRATION WITH APACHE HADOOP

35.1. INTEGRATION WITH APACHE HADOOP

The JBoss Data Grid connector allows the JBoss Data Grid to be a Hadoop compliant data source. It accomplishes this integration by providing implementations of Hadoop's **InputFormat** and **OutputFormat**, allowing applications to read and write data to a JBoss Data Grid server with best data locality. While JBoss Data Grid's implementation of the **InputFormat** and **OutputFormat** allow one to run traditional Hadoop Map/Reduce jobs, they may also be used with any tool or utility that supports Hadoop's **InputFormat** data source.

35.2. HADOOP DEPENDENCIES

The JBoss Data Grid implementations of Hadoop's formats are found in the following Maven dependency:

```
<dependency>
   <groupId>org.infinispan.hadoop</groupId>
   <artifactId>infinispan-hadoop-core</artifactId>
   <version>0.3.0.Final-redhat-9</version>
</dependency>
```

35.3. SUPPORTED HADOOP CONFIGURATION PARAMETERS

The following parameters are supported:

Table 35.1. Supported Hadoop Configuration Parameters

Parameter Name	Description	Default Value
hadoop.ispn.input.filter.factory	The name of the filter factory deployed on the server to pre-filter data before reading.	null (no filtering enabled)
hadoop.ispn.input.cache.name	The name of cache where data will be read.	___defaultcache
hadoop.ispn.input.remote.cache.servers	List of servers of the input cache, in the format: `host1:port;host2:port2`	localhost:11222
hadoop.ispn.output.cache.name	The name of cache where data will be written.	default
hadoop.ispn.output.remote.cache.servers	List of servers of the output cache, in the format: `host1:port;host2:port2`	null (no output cache)

Parameter Name	Description	Default Value
hadoop.ispn.input.read.batch	Batch size when reading from the cache.	5000
hadoop.ispn.output.write.batch	Batch size when writing to the cache.	500
hadoop.ispn.input.converter	Class name with an implementation of **org.infinispan.hadoop.KeyValueConverter**, applied after reading from the cache.	null (no converting enabled).
hadoop.ispn.output.converter	Class name with an implementation of **org.infinispan.hadoop.KeyValueConverter** , applied before writing.	null (no converting enabled).

35.4. USING THE HADOOP CONNECTOR

InfinispanInputFormat and InfinispanOutputFormat

In Hadoop, the **InputFormat** interface indicates how a specific data source is partitioned, along with how to read data from each of the partitions, while the **OutputFormat** interface specifies how to write data.

There are two methods of importance defined in the **InputFormat** interface:

1. The **getSplits** method defines a data partitioner, returning one or more **InputSplit** instances that contain information regarding a certain section of the data.

```
List<InputSplit> getSplits(JobContext context);
```

2. The **InputSplit** can then be used to obtain a **RecordReader** which will be used to iterate over the resulting dataset.

```
RecordReader<K,V> createRecordReader(InputSplit split,TaskAttemptContext context);
```

These two operations allow for parallelization of data processing across multiple nodes, resulting in Hadoop's high throughput over large datasets.

In regards to JBoss Data Grid, partitions are generated based on segment ownership, meaning that each partition is a set of segments on a certain server. By default there will be as many partitions as servers in the cluster, and each partition will contain all segments associated with that specific server.

Running a Hadoop Map Reduce job on JBoss Data Grid

Example of configuring a Map Reduce job targeting a JBoss Data Grid cluster:

```
import org.infinispan.hadoop.*;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.mapreduce.Job;

[...]
Configuration configuration = new Configuration();
configuration.set(InfinispanConfiguration.INPUT_REMOTE_CACHE_SERVER_LIST,
"localhost:11222");
configuration.set(InfinispanConfiguration.INPUT_REMOTE_CACHE_NAME, "map-reduce-in");
configuration.set(InfinispanConfiguration.OUTPUT_REMOTE_CACHE_SERVER_LIST,
"localhost:11222");
configuration.set(InfinispanConfiguration.OUTPUT_REMOTE_CACHE_NAME, "map-reduce-out");

Job job = Job.getInstance(configuration, "Infinispan Integration");
[...]
```

In order to target the JBoss Data Grid, the job needs to be configured with the **InfinispanInputFormat** and **InfinispanOutputFormat** classes:

```
[...]
// Define the Map and Reduce classes
job.setMapperClass(MapClass.class);
job.setReducerClass(ReduceClass.class);

// Define the JBoss Data Grid implementations
job.setInputFormatClass(InfinispanInputFormat.class);
job.setOutputFormatClass(InfinispanOutputFormat.class);
[...]
```

CHAPTER 36. INTEGRATION WITH EAP

36.1. INTEGRATION WITH EAP

While EAP includes Infinispan modules, they are intended for internal EAP use, and are not supported with JBoss Data Grid. To use JDG within EAP, use the JDG provided EAP modules. Using these modules will avoid any conflict with EAP's internal modules because the slot will be different. Using them will also allow for deployment of an application without packaging JDG within the deployments (WARs, EARs, etc.), thus minimizing their size.

36.2. INSTALLATION OF EAP MODULES

The modules for EAP can be downloaded from the Red Hat Customer Portal:

Procedure: Download EAP Modules

1. Log into the Customer Portal at https://access.redhat.com.
2. Click the **Downloads** button near the top of the page.
3. In the **Product Downloads** page, click **Red Hat JBoss Data Grid**.
4. Select the appropriate JBoss Data Grid version from the **Version:** drop down menu.
5. Locate the **Red Hat JBoss Data Grid 7.2 Library Module for JBoss EAP** entry and click the corresponding **Download** link.

The zip file should be extracted to **EAP_HOME/modules**. If the files were extracted correctly the infinispan core module would be under **EAP_HOME/modules/org/infinispan/core**.

36.3. EAP DEPENDENCIES

To configure the modules using Maven, mark the JDG dependencies as *provided* and configure the artifact archiver to generate a WAR file with the proper **MANIFEST.MF** using the following **pom.xml**:

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-core</artifactId>
    <version>${infinispan.version}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-cachestore-jdbc</artifactId>
    <version>${infinispan.version}</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
```

```
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <configuration>
        <archive>
          <manifestEntries>
            <Dependencies>org.infinispan.core:jdg-7.2 services, org.infinispan.cachestore.jdbc:jdg-7.2
services</Dependencies>
          </manifestEntries>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

36.4. DEPENDENCIES FOR SPECIFIC JDG COMPONENTS

Various example **MANIFEST.MF** configuration files to enable specific features of JDG are provided below.

36.4.1. Core Dependencies

To expose only JDG core dependencies to an application, add the following to the manifest:

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan:jdg-7.2 services
```

36.4.2. Remote/Hot Rod Dependencies

To connect to remote JDG servers via Hot Rod, including for execution of remote queries, use the module **org.infinispan.remote**. This exposes all needed dependencies automatically:

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan.remote:jdg-7.2 services
```

36.4.3. Embedded Querying Dependencies

For embedded querying, including the Infinispan Query DSL, Lucene, and Hibernate Search Queries, add the following to the manifest:

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan:jdg-7.2 services, org.infinispan.query:jdg-7.2 services
```

36.4.4. Lucene Directory Dependencies

To use JDG as a directory for Lucene using *org.apache.lucene.store.Directory*, the query module isn't needed, the following is sufficient

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan.lucene-directory:jdg-7.2 services
```

36.4.5. Hibernate Search Directory Provider Dependencies

The Hibernate Search directory provider for JDG is also contained within the JBoss Data Grid 7.2 Library Module for JBoss EAP zip file. It is not necessary to add an entry to the manifest file since the Hibernate Search module already has an optional dependency to it. When deciding what JDG module zip to use, start by checking which Hibernate Search is in use.

36.4.6. Using EAP's Internal Hibernate Search Modules

The Hibernate Search module present in EAP 7.1 has version 5.5.x, and has an optional dependency to module **org.infinispan.hibernate-search.directory-provider**, with slot **for-hibernatesearch-5.5**. This dependency is available once the Infinispan modules are installed.

36.4.7. Usage with Other Hibernate Search Modules

The module **org.hibernate.search:jdg-7.2** distributed with JDG is to be used together with Infinispan Query only (querying data from caches), and should not be used by Hibernate ORM applications. To use a Hibernate Search with a different version that is present in EAP, consult the Hibernate Search documentation.

Make sure the chosen Hibernate Search optional slot for **org.infinispan.hibernate-search.directory-provider** matches the one distributed with JBoss Data Grid.

36.5. USAGE OF EAP MODULES

An application can use JDG within EAP either in Library (embedded) Mode, or in EAP Subsystem Mode.

36.5.1. Library Mode

When using JDG within EAP in Library Mode, all **CacheManager** and cache instances are created in application logic. As such, the lifecycle of the **EmbeddedCacheManager** is tightly coupled with the application's lifecycle, resulting in any manager instances created by the application being destroyed when the application is destroyed.

36.5.2. EAP Subsystem Mode

In EAP Subsystem Mode, where JDG is a subsystem to EAP, it's possible for cache containers and caches to be created before runtime as part of EAP's **domain/configuration/domain.xml** configuration. This allows cache instances to be shared across multiple applications, with the lifecycle of the underlying cache container being independent of the deployed application.

36.6. CONFIGURATION FOR EAP SUBSYSTEM MODE

To enable EAP Subsystem Mode, add the following to the EAP configuration in **domain/configuration/domain.xml**.

NOTE

Only the first two steps are required for local cache instances.

1. Add the infinispan extensions to the **<extensions>** section

```
<extensions>
  <extension module="org.infinispan.extension:jdg-7.2"/>
  <extension module="org.jgroups.extension:jdg-7.2"/>

  <!--Other EAP extensions-->
</extensions>
```

2. Configure the Infinispan subsystem, along with all required containers and caches, in the server profile which requires Infinispan.

NOTE

Ensure the module attribute is defined or else the correct Infinispan classes won't be loaded.

```
<subsystem xmlns="urn:infinispan:server:core:8.5">
    <cache-container name="jdg-container" default-cache="default"
module="org.infinispan.extension:jdg-7.2">
        <transport channel="jdg-cluster"/>
        <global-state/>
        <distributed-cache-configuration name="default"/>
 <distributed-cache name="default"/>
    </cache-container>
</subsystem>
```

3. Define the EAP interface and socket bindings required by JGroup subsystems.

Interface definition:

```
<interfaces>
    <interface name="jdg">
        <inet-address value="${jdg.bind.address:127.0.0.1}"/>
    </interface>
</interfaces>
```

Socket bindings definition:

```
<socket-binding-group name="full-sockets" default-interface="public">
<socket-binding name="jdg-jgroups-udp" interface="jdg" port="55200" multicast-
address="${jdg.default.multicast.address:230.0.0.4}" multicast-port="45688"/>
<socket-binding name="jdg-jgroups-udp-fd" interface="jdg" port="54200"/>
</socket-binding-group>
```

For more information on EAP interface and socket bindings see Network and Port Configuration in the EAP Configuration Guide.

4. Define JGroups transport, ensuring the model attribute, for all protocols specified, is defined.

```
<subsystem xmlns="urn:infinispan:server:jgroups:8.0">
    <channels>
 <channel name="jdg-cluster" stack="udp"/>
    </channels>
    <stacks>
 <stack name="udp">
        <transport type="UDP" socket-binding="jdg-jgroups-udp" module="org.jgroups:jdg-7.2"/>
     <protocol type="PING" module="org.jgroups:jdg-7.2"/>
        <protocol type="MERGE3" module="org.jgroups:jdg-7.2"/>
        <protocol type="FD_SOCK" socket-binding="jdg-jgroups-udp-fd" module="org.jgroups:jdg-7.2"/>
        <protocol type="FD_ALL" module="org.jgroups:jdg-7.2"/>
        <protocol type="VERIFY_SUSPECT" module="org.jgroups:jdg-7.2"/>
        <protocol type="pbcast.NAKACK2" module="org.jgroups:jdg-7.2"/>
        <protocol type="UNICAST3" module="org.jgroups:jdg-7.2"/>
        <protocol type="pbcast.STABLE" module="org.jgroups:jdg-7.2"/>
        <protocol type="pbcast.GMS" module="org.jgroups:jdg-7.2"/>
        <protocol type="UFC" module="org.jgroups:jdg-7.2"/>
        <protocol type="MFC" module="org.jgroups:jdg-7.2"/>
        <protocol type="FRAG3" module="org.jgroups:jdg-7.2"/>
 </stack>
    </stacks>
</subsystem>
```

A command line script is also available to configure server mode:

```
# adding the necessary modules to the EAP configuration
# remember to add the datagrid library modules of JDG 7.2 before !
/extension=org.infinispan.extension\:jdg-7.2:add
/extension=org.jgroups.extension\:jdg-7.2:add

batch
/profile=full/subsystem=datagrid-infinispan:add
/profile=full/subsystem=datagrid-infinispan/cache-container=jdg-
container:add(module="org.infinispan.extension:jdg-7.2", default-cache="default"
/profile=full/subsystem=datagrid-infinispan/cache-container=jdg-
container/transport=TRANSPORT:add(channel=jdg-cluster)
/profile=full/subsystem=datagrid-infinispan/cache-container=jdg-container/global-
state=GLOBAL_STATE:add

# add an interface for JDG cluster communication, can be skipped if the same as JGroups or public is
used
/interface=jdg:add(inet-address="${jdg.bind.address:127.0.0.1}"

# add the port numbers for JDG JGroups
/socket-binding-group=full-sockets/socket-binding=jdg-jgroups-udp:add(interface="jdg", port=55200,
multicast-address="${jdg.default.multicast.address:230.0.0.4}", multicast-port="45688"
/socket-binding-group=full-sockets/socket-binding=jdg-jgroups-udp-fd:add(port=54200,
interface="jdg")

# adding the datagrid JGroups subsystem with UDP stack
/profile=full/subsystem=datagrid-jgroups:add(default-channel=jdg-cluster)
/profile=full/subsystem=datagrid-jgroups/channel=jdg-cluster:add(stack=udp)
/profile=full/subsystem=datagrid-jgroups/stack=udp:add()
/profile=full/subsystem=datagrid-jgroups/stack=udp/transport=UDP:add(socket-binding=jdg-jgroups-
udp, module="org.jgroups:jdg-7.2")
```

```
/profile=full/subsystem=datagrid-jgroups/stack=udp/protocol=PING:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-jgroups/stack=udp/protocol=MERGE3:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-jgroups/stack=udp/protocol=FD_SOCK:add(module="org.jgroups:jdg-7.2", socket-binding=jdg-jgroups-udp-fd)
/profile=full/subsystem=datagrid-jgroups/stack=udp/protocol=FD_ALL:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-jgroups/stack=udp/protocol=VERIFY_SUSPECT:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-jgroups/stack=udp/protocol=pbcast.NAKACK2:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-jgroups/stack=udp/protocol=UNICAST3:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-jgroups/stack=udp/protocol=pbcast.STABLE:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-jgroups/stack=udp/protocol=pbcast.GMS:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-jgroups/stack=udp/protocol=UFC:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-jgroups/stack=udp/protocol=MFC:add(module="org.jgroups:jdg-7.2")
/profile=full/subsystem=datagrid-jgroups/stack=udp/protocol=FRAG3:add(module="org.jgroups:jdg-7.2")

# add a configuration as this is needed if the CLI is used to add a cache
/profile=full/subsystem=datagrid-infinispan/cache-container=jdg-container/configurations=CONFIGURATIONS:add
/profile=full/subsystem=datagrid-infinispan/cache-container=jdg-container/configurations=CONFIGURATIONS/distributed-cache-configuration=default:add

run-batch
```

To add the cache use the following command:

```
# add a simple cache
/profile=full/subsystem=datagrid-infinispan/cache-container=jdg-container/distributed-cache=default:add(configuration=default)
```

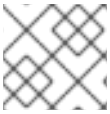
36.7. ACCESSING CONTAINERS AND CACHES REMOTELY

Once a container has been defined in the server's configuration, it is possible to inject an instance of a **CacheContainer** or **Cache** into the application using the **@Resource** JNDI lookup. A container is accessed using the string **java:jboss/datagrid-infinispan/container/<container_name>**, and similarly, a cache is accessed via **java:jboss/datagrid-infinispan/container/<container_name>/cache/<cache_name>**.

The example below shows how to inject the **CacheContainer** called "jdg-container" and the distributed cache "default" into an application.

```
public class ExampleApplication {
   @Resource(lookup = "java:jboss/datagrid-infinispan/container/jdg-container")
   CacheContainer container;
```

```
    @Resource(lookup = "java:jboss/datagrid-infinispan/container/jdg-container/cache/default")
    Cache cache;
}
```



NOTE

This example code has a dependency on the **jdg-7.2** module.

36.8. TROUBLESHOOTING EAP AND JDG IN EAP SUBSYSTEM MODE

36.8.1. Enable logging

Enabling trace on **org.jboss.modules** can be useful to debug issues like **LinkageError** and **ClassNotFoundException**. To enable trace logging at runtime use the EAP CLI:

```
bin/jboss-cli.sh -c '/subsystem=logging/logger=org.jboss.modules:add'
bin/jboss-cli.sh -c '/subsystem=logging/logger=org.jboss.modules:write-
attribute(name=level,value=TRACE)'
```

36.8.2. Print Dependency Tree

The following command can be used to print all dependencies for a certain module. For example, to obtain the tree for the module **org.infinispan:jdg-7.2**, execute the following from **EAP_HOME**:

```
java -jar jboss-modules.jar -deptree -mp modules/ "org.infinispan:jdg-7.2"
```

CHAPTER 37. HIGH AVAILABILITY USING SERVER HINTING

37.1. SERVER HINTING

Server Hinting helps you achieve high availability with your Red Hat JBoss Data Grid deployment.

To use Server Hinting, you provide information about the physical topology with attributes that identify servers, racks, or data centers to achieve more resilience with your data in the event that all the nodes in a given physical location become unavailable.

When you configure Server Hinting, JBoss Data Grid uses the location information you provided to distribute data across the cluster so that backup copies of data are stored on as many servers, racks, and data centers as possible.

In some cases JBoss Data Grid stores copies of data on nodes that share the same physical location. For example, if the number of owners for segments is greater than the number of distinct sites, then JBoss Data Grid assigns more than one owner for a given segment in the same site.

NOTE

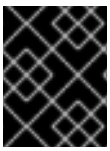
Server Hinting does not apply to total replication, which requires complete copies of data on every node.

Consistent Hashing controls how data is distributed across nodes. JBoss Data Grid uses **TopologyAwareSyncConsistentHashFactory** if you enable Server Hinting. For details see ConsistentHashFactories.

37.2. CONSISTENTHASHFACTORIES

37.2.1. ConsistentHashFactories

Red Hat JBoss Data Grid offers a pluggable mechanism for selecting the consistent hashing algorithm and provides different implementations. You can also use a custom implementation.



IMPORTANT

You can configure **ConsistentHashFactory** implementations in Library Mode only. In Remote Client/Server Mode, this configuration is not valid and results in a runtime error.

ConsistentHashFactory implementations

- **SyncConsistentHashFactory** guarantees that the key mapping is the same for each cache, provided the current membership is the same. This has a drawback in that a node joining the cache can cause the existing nodes to also exchange segments, resulting in either additional state transfer traffic, the distribution of the data becoming less even, or both. This is the default consistent hashing implementation without Server Hinting.
- **TopologyAwareSyncConsistentHashFactory** is equivalent to **SyncConsistentHashFactory** but used with Server Hinting to distribute data across the topology so that backed up copies of data are stored on different nodes in the topology than the primary owners. This is the default consistent hashing implementation with Server Hinting.

- **DefaultConsistentHashFactory** keeps segments balanced evenly across all nodes, however the key mapping is not guaranteed to be same across caches as this depends on the history of each cache.
- **TopologyAwareConsistentHashFactory** is equivalent to **DefaultConsistentHashFactory** but used with Server Hinting to distribute data across the topology so that backed up copies of data are stored on different nodes in the topology than the primary owners.

You configure JBoss Data Grid to use a consistent hash implementation with the **consistent-hash-factory** attribute, as in the following example:

```
<distributed-cache consistent-hash-
factory="org.infinispan.distribution.ch.impl.SyncConsistentHashFactory">
```

This configuration guarantees caches with the same members have the same consistent hash, and if the **machineId**, **rackId**, or **siteId** attributes are specified in the transport configuration it also spreads backup copies across physical machines/racks/data centers.

It has a potential drawback in that it can move a greater number of segments than necessary during re-balancing. This can be mitigated by using a larger number of segments.

Another potential drawback is that the segments are not distributed as evenly as possible, and actually using a very large number of segments can make the distribution of segments worse.

Despite the above potential drawbacks the **SyncConsistentHashFactory** and **TopologyAwareSyncConsistentHashFactory** both tend to reduce overhead in clustered environments, as neither of these calculate the hash based on the order that nodes have joined the cluster. In addition, both of these classes are typically faster than the default algorithms as both of these classes allow larger differences in the number of segments allocated to each node.

37.2.2. Implementing a ConsistentHashFactory

A custom **ConsistentHashFactory** must implement the **org.infinispan.distribution.ch.ConsistenHashFactory** interface with the following methods (all of which return an implementation of **org.infinispan.distribution.ch.ConsistentHash**):

ConsistentHashFactory Methods

```
create(Hash hashFunction, int numOwners, int numSegments, List<Address>
members,Map<Address, Float> capacityFactors)
updateMembers(ConsistentHash baseCH, List<Address> newMembers, Map<Address,
Float> capacityFactors)
rebalance(ConsistentHash baseCH)
union(ConsistentHash ch1, ConsistentHash ch2)
```

37.3. KEY AFFINITY SERVICE

37.3.1. Key Affinity Service

The key affinity service allows a value to be placed in a certain node in a distributed Red Hat JBoss Data Grid cluster. The service returns a key that is hashed to a particular node based on a supplied cluster address identifying it.

The keys returned by the key affinity service cannot hold any meaning, such as a username. These are only random identifiers that are used throughout the application for this record. The provided key generators do not guarantee that the keys returned by this service are unique. For custom key format, you can pass your own implementation of KeyGenerator.

The following is an example of how to obtain and use a reference to this service.

Key Affinity Service

```
EmbeddedCacheManager cacheManager = getCacheManager();
Cache cache = cacheManager.getCache();
KeyAffinityService keyAffinityService =
      KeyAffinityServiceFactory.newLocalKeyAffinityService(
         cache,
         new RndKeyGenerator(),
         Executors.newSingleThreadExecutor(),
         100);
Object localKey = keyAffinityService.getKeyForAddress(cacheManager.getAddress());
cache.put(localKey, "yourValue");
```

The following procedure is an explanation of the provided example.

Using the Key Affinity Service

1. Obtain a reference to a cache manager and cache.
2. This starts the service, then uses the supplied **Executor** to generate and queue keys.
3. Obtain a key from the service which will be mapped to the local node (**cacheManager.getAddress()** returns the local address).
4. The entry with a key obtained from the **KeyAffinityService** is always stored on the node with the provided address. In this case, it is the local node.

37.3.2. Lifecycle

KeyAffinityService extends **Lifecycle**, which allows the key affinity service to be stopped, started, and restarted.

Key Affinity Service Lifecycle Parameter

```
public interface Lifecycle {
    void start();
    void stop();
}
```

The service is instantiated through the **KeyAffinityServiceFactory**. All factory methods have an **Executor**, that is used for asynchronous key generation, so that this does not occur in the caller's thread. The user controls the shutting down of this **Executor**.

The **KeyAffinityService** must be explicitly stopped when it is no longer required. This stops the background key generation, and releases other held resources. The **KeyAffinityServce** will only stop itself when the cache manager with which it is registered is shut down.

37.3.3. Topology Changes

KeyAffinityService key ownership may change when a topology change occurs. The key affinity service monitors topology changes and updates so that it doesn't return stale keys, or keys that would map to a different node than the one specified. However, this does not guarantee that a node affinity hasn't changed when a key is used. For example:

1. Thread (**T1**) reads a key (**K1**) that maps to a node (**A**).
2. A topology change occurs, resulting in **K1** mapping to node **B**.
3. **T1** uses **K1** to add something to the cache. At this point, **K1** maps to **B**, a different node to the one requested at the time of read.

The above scenario is a not ideal, however it is a supported behavior for the application, as the keys that are already in use may be moved over during cluster change. The **KeyAffinityService** provides an access proximity optimization for stable clusters, which does not apply during the instability of topology changes.

CHAPTER 38. DISTRIBUTED EXECUTION

38.1. 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**.

38.2. DISTRIBUTED EXECUTOR SERVICE

A **DistributedExecutorService** controls the execution of **DistributedCallable**, and other **Callable** and **Runnable**, classes on the cluster. These instances are tied to a specific cache that is passed in upon instantiation:

```
DistributedExecutorService des = new DefaultExecutorService(cache);
```

It is only possible to execute a **DistributedTask** against a subset of keys if **DistributedCallable** is extended, as discussed in DistributedCallableAPI. If a task is submitted in this manner to a single node, then JBoss Data Grid will locate the nodes containing the indicated keys, migrate the **DistributedCallable** to this node, and return a **CompletableFuture**. Alternatively, if a task is submitted to all available nodes in this manner then only the nodes containing the indicated keys will receive the task.

Once a **DistributedTask** has been created it may be submitted to the cluster using any of the below methods:

- The task can be submitted to all available nodes and key/value pairs on the cluster using the **submitEverywhere** method:

  ```
  des.submitEverywhere(task)
  ```

- The **submitEverywhere** method can also take a set of keys as an argument. Passing in keys in this manner will submit the task only to available nodes that contain the indicated keys:

  ```
  des.submitEverywhere(task, $KEY)
  ```

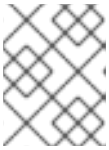
- If a key is specified, then the task will be executed on a single node that contains at least one of the specified keys. Any keys not present locally will be retrieved from the cluster. This version of

the **submit** method accepts one or more keys to be operated on, as seen in the following examples:

```
des.submit(task, $KEY)
des.submit(task, $KEY1, $KEY2, $KEY3)
```

- A specific node can be instructed to execute the task by passing the node's **Address** to the **submit** method. The below will only be executed on the cluster's **Coordinator**:

```
des.submit(cache.getCacheManager().getCoordinator(), task)
```



NOTE

By default tasks are automatically balanced, and there is typically no need to indicate a specific node to execute against.

38.3. 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 the **Callable** interface must extend **DistributedCallable** if access to the cache or the set of passed in keys is required.

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);

}
```

38.4. 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.

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;
    }
}
```

38.5. 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:

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.

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();
}
```

38.6. 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.

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();
```

38.7. DISTRIBUTED EXECUTION AND LOCALITY

In a Distributed Environment ownership, in regards to the **DistributionManager** and **ConsistentHash**, is theoretical; neither of these classes have any knowledge if data is actively in the cache. Instead, these classes are used to determine which node should store the specified key.

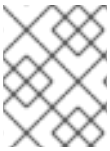
To examine the locality of a given key use either of the following options:

- **Option 1:** Confirm that the key is both found in the cache and the **DistributionManager** indicates it is local, as seen in the following example:

```
(cache.getAdvancedCache().withFlags(SKIP_REMOTE_LOOKUP).containsKey(key)
&& cache.getAdvancedCache().getDistributionManager().getLocality(key).isLocal())
```

- **Option 2:** Query the **DataContainer** directly:

```
cache.getAdvancedCache().getDataContainer().containsKey(key)
```



NOTE

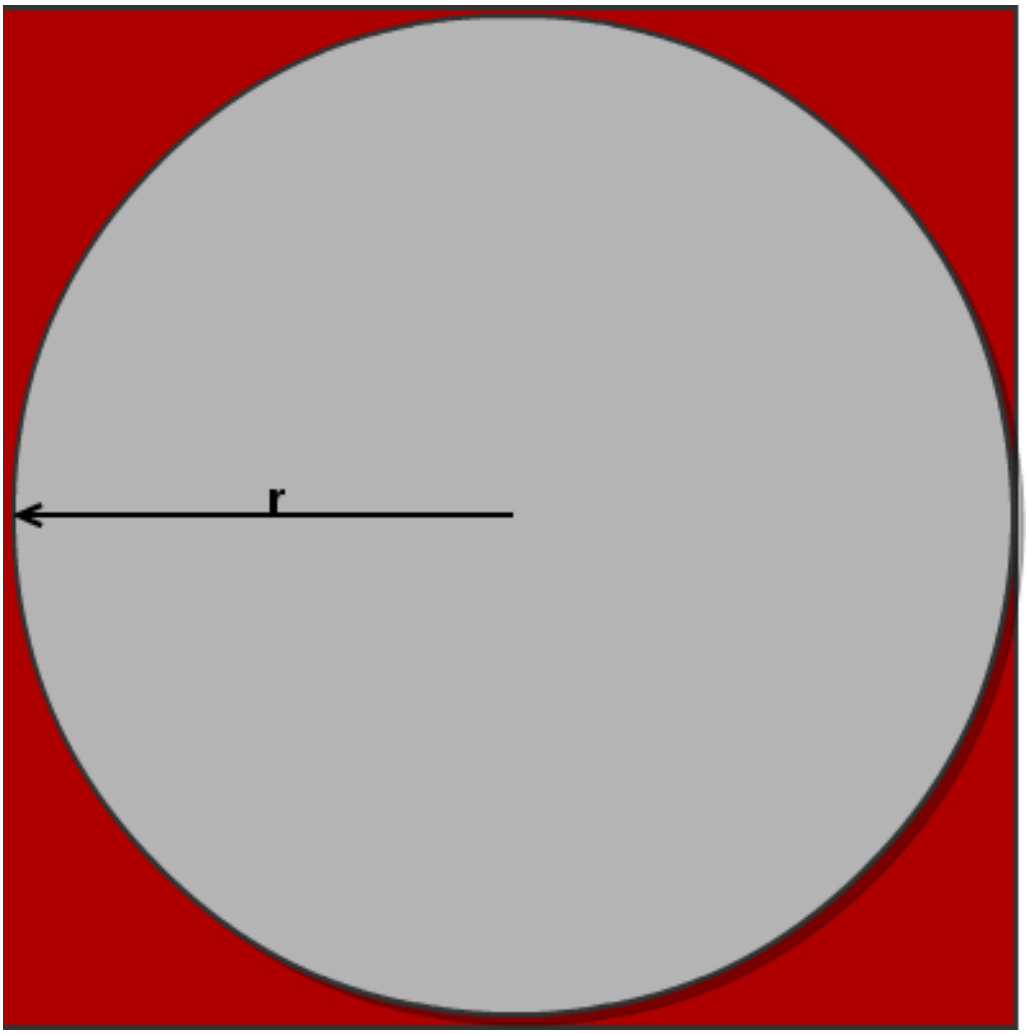
If the entry is passivated then the **DataContainer** will return **False**, regardless of the key's presence.

38.7.1. Distributed Execution Example

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

1. As shown below, the area of a square is:
   Area of a Square (S) = $4r^2$
2. The following is an equation for the area of a circle:
   Area of a Circle (C) = $\pi \times r^2$
3. Isolate r from the first equation:
   $r^2 = S/4$
4. Inject this value of r into the second equation to find a value for Pi:
   $C = S\pi/4$
5. Isolating in the equation results in:
   $C = S\pi/4$
   $4C = S\pi$
   $4C/S = \pi$

Figure 38.1. 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 is previously worked out to 4C/S. We can use this to derive the approximate value of . By maximizing the amount of darts thrown, we can derive an improved approximation of .

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

Distributed Execution Example

```
public class PiAppx {

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

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

    DistributedExecutorService des = new DefaultExecutorService(cache);
    long start = System.currentTimeMillis();
    CircleTest ct = new CircleTest(numberPerWorker);
    List<CompletableFuture<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);
      }
    }
  }
```

CHAPTER 39. STREAMS

39.1. STREAMS

Streams were introduced in Java 8, allowing an efficient way of performing operations on very large data sets, including the entirety of a given cache. These operations are performed on collections instead of procedurally iterating over the entire dataset.

In addition, if the cache is distributed then operations may be performed even more efficiently as these may be performed across the cluster concurrently.

A **Stream** may be obtained by invoking the **stream()**, for a single-threaded stream, or **parallelStream()**, for a multi-threaded stream, methods on a given **Map**. Parallel streams are discussed in more detail at Parallelism.

39.2. USING STREAMS ON A LOCAL/INVALIDATION/REPLICATION CACHE

A stream used with a local, invalidation, or replication cache can be used identical to a stream on a regular collection.

For example, consider a cache that contains a number of Books. To create a Map that contains all entries with JBoss in the title the following could be used:

```
Map<Object, String> jbossBooks = cache.entrySet().stream()
   .filter(e -> e.getValue().getTitle().contains("JBoss"))
   .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

39.3. USING STREAMS WITH A DISTRIBUTION CACHE

When a Stream operation is performed on a distribution cache it will send the intermediate and terminal operations to each node, and then the resulting data will be sent back to the originating node. This behavior allows operations to be performed on the remote nodes and only the end results returned, resulting in much better performance as the intermediate values are not returned.

Rehash Aware

Once the stream has been created the data will be segmented so that each node will only perform operations upon the data that it owns as the primary owner. Assuming the segments are granular enough to provide an even distribution of data per node, this allows for even processing of data across the segments.

This process can be volatile if new nodes are added or old nodes leave the cluster, as the data is redistributed between nodes. This may cause issues where data can be processed a second time; however, Distributed Streams handles the redistribution of data automatically so that manual monitoring of nodes does not need to occur.

39.4. SETTING TIMEOUTS

It is possible to configure a timeout value for the operation request; this is only used for remote requests and is configured per request. This means that local requests will never timeout, and if a failover occurs then each subsequent request will have a new timeout.

If no timeout is specified then the replication timeout will be used by default. This may be manually configured by using the **timeout(long timeout, TimeUnit unit)** method of the stream, as seen in the following example:

```
CacheStream<Map.Entry<Object, String>> stream = cache.entrySet().stream();
stream.timeout(1, TimeUnit.MINUTES);
```

39.5. DISTRIBUTED STREAMS

39.5.1. Distributed Streams

Distributed Streams work similarly to map reduce; however, with Distributed Streams there are zero to many intermediate operations followed by a single terminal operation that is sent to each node where work is performed. The following steps are used for this behavior:

1. The desired segments are grouped by which node is the primary owner of each given segment.
2. A request is generated for each remote node. This request contains the intermediate and terminal operations, along with the segments to process.
    - The thread where the terminal operation was initiated will perform the local operation directly.
    - Each remote node will receive the generated request, run the operations on a remote thread, and then send the response back.
3. Once all requests complete the user thread will gather all responses and perform any reductions specified by the operations.
4. The final response is returned to the user.

39.5.2. Marshallability

When using distributed or replicated caches the keys and values must be marshallable; in addition, operations executed on Distributed Streams must also be marshallable, as these operations are sent to the other nodes in the cluster. This is most commonly accomplished by using a new class that is either **Serializable** or has an **Externalizer** registered; however, as the **FunctionalInterface** implements **Serializable** all lambdas are instantly serialized and thus no additional cast is required.

NOTE

Intermediate values in distributed streams do not need to be marshallable; only the final value sent back, typically the terminal operation, must be marshallable.

If a lambda function is in use this may be serialized by casting the parameter as an instance of **Serializable**. For instance, consider a cache that stores Book entries; the following would create a collection of Book instances that match a specific author:

```
List<Book> books = cache.keySet().stream()
   .filter(e -> e.getAuthor().equals("authorname"))
   .collect(toList());
```

Additionally, not all produced **Collectors** are marshallable by default. JBoss Data Grid has included **org.infinispan.stream.CacheCollectors** as a convenient way to utilize any combination of **Collectors** that function properly when marshalling is required.

39.5.3. Parallelism

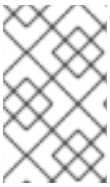
There are two different methods to parallelize streams:

- Parallel Streams - causing each operation to be executed in parallel on a single node
- Parallel Distribution - parallelizing the request so that it involves multiple nodes

By default, Distributed Streams enable parallel distribution; however, this may be further coupled with a parallel **Stream**, allowing concurrent operations executing across multiple nodes, with multiple threads on each node.

To mark a **Stream** as parallel it may either be obtained with **parallelStream()**, or it may be enabled after obtaining the **Stream** by invoking **parallel()**. The following example shows both methods:

```
// Obtain a parallel Stream initially
List<Book> books = cache.keySet().parallelStream()
   [...]

// Create the initial stream and then invoke parallel
List<Book> books = cache.keySet().stream()
   .parallel()
   [...]
```



NOTE

Some operations, such as rehash aware iterator or forEach operations, have a sequential stream forced locally. Using parallel streams on these operations is not possible at this time.

39.5.4. Distributed Operators

39.5.4.1. Terminal Operator Distributed Result Reductions

Below each terminal operator is discussed, along with how the distributed reduction works for each one.

- **allMatch**
  This operator is run on each node and then all results are combined using a logical **AND** operation locally to obtain the final value. If a normal stream operation returns early then these methods will complete early as well.
- **noneMatch anyMatch**
  These operators are run on each node and then all results are combined using a logical **OR** operation locally to obtain the final value. If a normal stream operation returns early then these methods will complete early as well.
- **collect**
  The collect method can perform a few extra steps. Similar to other methods the remote node will perform everything as expected; however, instead of performing the final finisher operator it sends back the fully combined results. The local thread will then combine all local and remote

results into a value which then performs the finisher operator. In addition, the final value does not need to be serializable, but the values produced from the supplier and combiner methods must be serialized.

- **count**
  The count method simply adds the numbers received from each node.

- **findAny findFirst**
  The findAny method will return the first value found, regardless if it was from a remote or local node. This operation supports early termination, as once an initial value has been found no others will be processed. The findFirst method behaves similarly, but requires a sorted intermediate operation which is described in Intermediate_Operation_Exceptions.

- **max min**
  The max and min methods find the respective value on each node before a final reduction is performed locally to determine the true max or min across all nodes.

- **reduce**
  The various reduce methods seralize the result as much as possible before accumulating the local and remote results together locally, combining if enabled. Due to this behavior a value returned from the combiner does not need to be serializable.

39.5.4.2. Key Based Rehash Aware Operators

Unlike the other terminal operators each of the following operators require a special type of rehash awareness to keep track of which keys per segment have been processed. This guarantees each key will be processed exactly once, for **iterator** and **spliterator** operators, or at least once, for **forEach**, even if cluster membership changes.

- **iterator spliterator**
  These operators return batches of entries when run on a remote node, where the next batch is only sent after the previous is fully consumed. This behavior is to limit how many entries are retained in memory at any given time. The user node will keep track of which keys have been processed, and once a segment has completed those keys will be released from memory. Because of this behavior it is preferable to use sequential processing, allowing only a subset of segment keys to be held in memory instead of having keys from all nodes retained.

- **forEach**
  While **forEach** returns batches it only returns a batch after it has finished processing at least a batch worth of keys. This way the originating node knows which keys have been processed already, which reduces the possibility of processing the same entry again; however, it is possible to have the same set processed repeatedly if a node goes down unexpectedly. In this case the node could have been processing an uncompleted batch when it went down, resulting in the same batch to be ran again when the rehash failure operation occurs. Adding a node will not cause this issue, as the rehash failover does not occur until all responses are received.

The operations' batch sizes are controlled by the same value, **distributedBatchSize**, on the **CacheStream**. If no value is set then it will default to the **chunkSize** configured in state transfer. While larger values will allow for larger batches, resulting in fewer returns, this results in increased memory usage, and testing should be performed to determine an appropriate size for each application.

39.5.4.3. Intermediate Operation Exceptions

The following intermediate operations have special exceptions. All of these methods have some sort of artificial iterator implanted in the stream processing to guarantee correctness, and due to this using any of the following may cause severe performance degradation.

- **Skip**
  An artificial iterator is implanted up to the skip operation, and then results are brought locally so that the appropriate number of elements may be skipped.

- **Peek**
  An artificial iterator is implanted up to the peek operation. Only up to a number of peeked elements are returned to a remote node, and then results are brought locally so that it may peek at only the desired amount.

- **Sorted**
  An artificial iterator is implanted up to the sorted operation, and then all results are locally sorted.

WARNING

This operation requires having all entries in memory on the local node.

- **Distinct**
  Distinct is performed on each remote node and then an artificial iterator returns those distinct values, before all of those results have a distinct operation performed upon them.

WARNING

This operation requires having all entries in memory on the local node.

39.5.5. Distributed Stream Examples

A classic example of Map/Reduce is word count. Assuming we have a cache with **String** for keys and values, and we need to count the occurrence of all words in all sentences, this could be implemented using the following:

```
Map<String, Long> wordCountMap = cache.entrySet().parallelStream()
   .map(e -> e.getValue().split("\\s"))
   .flatMap(Arrays::stream)
   .collect(CacheCollectors.serializableCollector(() -> Collectors.groupingBy(Function.identity(),
Collectors.counting())));
```

If we wanted to revise the example to find the most frequent word we would need to have all words available and counted locally first. The following snippet extends our previous example to perform this search:

```
String mostFrequentWord = cache.entrySet().parallelStream()
   .map((Serializable & Function<Map.Entry<String, String>, String[]>) e -> e.getValue().split("\\s"))
   .flatMap((Function<String[], Stream<String>>) Arrays::stream)
   .collect(CacheCollectors.serializableCollector(() -> Collectors.collectingAndThen(
```

```
      Collectors.groupingBy(Function.identity(), Collectors.counting()),
         wordCountMap -> {
            String mostFrequent = null;
            long maxCount = 0;
            for (Map.Entry<String, Long> e : wordCountMap.entrySet()) {
               int count = e.getValue().intValue();
               if (count > maxCount) {
                  maxCount = count;
                  mostFrequent = e.getKey();
               }
            }
            return mostFrequent;
         }));
```

At present, this last step will be executed in a single thread. We can further optimize this operation by using a parallel stream locally to perform the final operation:

```
Map<String, Long> wordCount = cache.entrySet().parallelStream()
   .map((Function<Map.Entry<String, String>, String[]>) e -> e.getValue().split("\\s"))
   .flatMap((Function<String[], Stream<String>>) Arrays::stream)
   .collect(CacheCollectors.serializableCollector(() -> Collectors.groupingBy(Function.identity(),
Collectors.counting())));
Optional<Map.Entry<String, Long>> mostFrequent = wordCount.entrySet().parallelStream()
   .reduce((e1, e2) -> e1.getValue() > e2.getValue() ? e1 : e2);
```

CHAPTER 40. SCRIPTING

40.1. SCRIPTING

JBoss Data Grid includes a method of storing scripts on servers, allowing remote clients to execute scripts locally with the JDK's **javax.script.ScriptEngines**. By default the JDK comes with Nashorn, capable of running JavaScript; however, this may be extended to run any JVM language that offers their own **ScriptEngine**.

40.2. ACCESSING THE SCRIPT CACHE

Scripts are stored in a special, protected cache entitled **___script_cache**. As this is a protected cache only loopback requests or connections with authorization enabled will be allowed to access the cache.

The following requirements must be met to connect to the **___script_cache** remotely:

- A user has been defined with the **___script_manager** role.
- The client has a secure connection to the server; this may be attained by following the instructions in Securing Interfaces.
- Authorization has been enabled on the cache-container.

Configuring the Server for Access the Script Cache

The following example covers configuring the server to access the script cache, using the **DIGEST-MD5** method of securing the Hot Rod connector.

1. Add a user to the server as follows:
   a. Execute the *$JDG_HOME/bin/add-user.sh* (Linux) or *$JDG_HOME\bin\add-user.bat* (Windows) script.
   b. Enter **b** at the first prompt to create an **ApplicationRealm** user.

```
What type of user do you wish to add?
 a) Management User (mgmt-users.properties)
 b) Application User (application-users.properties)
(a): b
```

   c. Follow the prompts to define the desired username and password for the user.
   d. When prompted for the groups enter **___script_manager** for this user:

```
What groups do you want this user to belong to? (Please enter a comma separated list, or leave blank for none)[  ]: ___script_manager
```

2. Secure the communication between the client and server. As this example is using **DIGEST-MD5** the instructions in Configure Hot Rod Authentication (MD5) will be followed. The following snippet demonstrates the necessary xml configuration:

```
<cache-container name="local" default-cache="default" statistics="true">
  <security>
    <authorization>
      <identity-role-mapper />
```

```
        <role name="admin" permissions="ALL" />
        <role name="reader" permissions="READ" />
        <role name="writer" permissions="WRITE" />
        <role name="supervisor" permissions="READ WRITE EXEC" />
    </authorization>
  </security>
  [...]
<cache-container>
[...]
<hotrod-connector socket-binding="hotrod" cache-container="local">
  <authentication security-realm="ApplicationRealm">
    <sasl server-name="scriptserver" mechanisms="DIGEST-MD5" qop="auth" />
  </authentication>
</hotrod-connector>
```

3. Create the cache manager using the secured connection, as seen in the following code snippet:

```
Configuration config = new ConfigurationBuilder()
    .addServer()
        .host("localhost")
        .port(11222)
    .security()
        .authentication()
        .enable()
        .saslMechanism("DIGEST-MD5")
        .serverName("scriptserver")
        .callbackHandler(new MyCallbackHandler("user", "ApplicationRealm",
"password".toCharArray()))
    .build();

cacheManager = new RemoteCacheManager(config);
```

40.3. INSTALLING SCRIPTS

A script may be added to the **___script_cache** by putting the script into the cache itself with the name of the script as the key, and the content of the script as the value. If the name of the script contains a filename extension, such as *sample.js* , then the extension will determine the engine used to execute the script. This behavior may be overridden by specifying metadata inside the script itself.

As the contents of the script should be stored in the value of the **___script_cache** this may either be loaded from a pre-existing file, or manually entered. The following examples demonstrate both of these options:

Loading a Script From a File

Assuming the script is stored within a file the following code sample may be used to read the contents of the file and store it into the scripting cache:

```
private static final String SCRIPT_CACHE = "___script_cache";
private RemoteCache<String, String> scriptingCache;
[...]
    scriptingCache = cacheManager.getCache(SCRIPT_CACHE);
[...]

    private void loadScript(String filename) throws IOException{
```

```
        StringBuilder sb = new StringBuilder();
        BufferedReader reader = new BufferedReader(new FileReader(filename));
        for (String line = reader.readLine(); line != null; line = reader.readLine()) {
            sb.append(line);
            sb.append("\n");
        }
        System.out.println(sb.toString());
        scriptingCache.put(filename,sb.toString());
    }
```

Defining the Contents of the Script

Instead of loading a script from a file the script may be manually defined and placed into the scripting cache:

```
RemoteCache<String, String> scriptCache = cacheManager.getCache("___script_cache");
scriptCache.put("multiplication.js",
  "// mode=local,language=javascript\n" +
  "// parameters=[multiplicand,multiplier]" +
  "multiplicand * multiplier\n");
```

40.4. SCRIPTING METADATA

Metadata may be stored in the script to provide additional information to the server on how the script is executed. This metadata is contained in a specially formatted comment on the first lines of the script.

Properties are defined as key=value pairs separated by commas, with the comment styles, such as //, **;;**, or **\#**, depending on the scripting language in use. This information may be split over multiple lines if necessary, and single or double quotes may be used to delimit the values.

The following is an example of a valid metadata comment:

```
// name=test, language=javascript
// mode=local, parameters=[a,b,c]
```

Metadata Properties

The following metadata properties are available:

- **mode**: defines the mode of execution of a script. Can be one of the following values:
  - **local**: the script will be executed only by the node handling the request. The script itself however can invoke clustered operations.
  - **distributed**: runs the script using the Distributed Executor Service.
- **language**: defines the script engine that will be used to execute the script, e.g. Javascript.
- **extension**: an alternative method of specifying the script engine that will be used to execute the script, e.g. js.
- **role**: a specific role which is required to execute the script.
- **parameters**: an array of valid parameter names for this script. Invocations which specify parameter names not included in this list will cause an exception.

As the execution mode is a characteristic of the script there is no additional configuration required on the client to invoke scripts in different modes.

40.5. SCRIPT BINDINGS

The script engine exposes several internal objects as pre-defined bindings when the script is executed. These are:

- **cache**: the cache against which the script is being executed.
- **cacheManager**: the cacheManager for the cache.
- **marshaller**: the marshaller to use for marshalling/unmarshalling data to the cache.
- **scriptingManager**: the instance of the script manager which is being used to run the script. This can be used to run other scripts from a script.

40.6. SCRIPT PARAMETERS

In addition to the standard bindings, a script may have a set of named parameters passed in which also appear as bindings. Parameters are passed in as a map of name, value pairs where the name is a string, and the value is any value understood by the marshaller in use.

Consider the following script which takes two parameters, **multiplicand** and **multiplier**:

```
// mode=local,language=javascript
// parameters=[multiplicand,multiplier]
multiplicand * multiplier
```

As the last operation is an evaluation its result will be returned to the script invoker. Passing in values changes depending on how the script is executed, and will be covered under each execution method.

40.7. SCRIPT EXECUTION USING THE HOT ROD JAVA CLIENT

If authorization is disabled on the server then anyone may execute scripts once they have been installed. Otherwise, only users with **EXEC** permissions will be allowed to run previously installed scripts.

Scripts may be executed in Hot Rod by calling **execute(scriptName, parameters)** on the cache where the script should be executed. In this case the **scriptName** corresponds with the name of the script stored in the **___script_cache**, and **parameters** is a **Map<String,Object>** of named parameters.

The following example demonstrates executing the above **multiplication.js** script through Hot Rod:

```
RemoteCache<String, Integer> cache = cacheManager.getCache();
// Create the parameters for script execution
Map<String, Object> params = new HashMap<>();
params.put("multiplicand", 10);
params.put("multiplier", 20);
// Run the script on the server, passing in the parameters
Object result = cache.execute("multiplication.js", params);
```

40.8. SCRIPT EXAMPLES

The following examples demonstrate various tasks to assist in the reader's understanding of the scripting syntax, and to get ideas on what tasks may be suitable for scripts in each environment.

Distributed Execution

The following is a script that runs within a Distributed Executor. Each node will return its address, and all nodes will be collected in a **List** to be returned to the client:

```
// mode:distributed,language=javascript
cacheManager.getAddress().toString();
```

Word Count Stream

The following is a script that runs on the local cache, counting the occurrences of each word in the result set, and then returning the words and their occurrences in a key, value pairing:

```
// mode=local,language=javascript
var Function = Java.type("java.util.function.Function")
var Collectors = Java.type("java.util.stream.Collectors")
var Arrays = Java.type("org.infinispan.scripting.utils.JSArrays")
cache
    .entrySet().stream()
    .map(function(e) e.getValue())
    .map(function(v) v.toLowerCase())
    .map(function(v) v.split(/[\W]+/))
    .flatMap(function(f) Arrays.stream(f))
    .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
```

40.9. LIMITATIONS WHEN EXECUTING STORED SCRIPTS

Java Streams throw an error when used with clusters in DIST mode

It is not possible to use scripts that create a **Stream** in JavaScript when the cluster is in **DIST** mode. Any attempts to execute these scripts will result in a **NotSerializableException**, as the lambdas fail when attempting to be serialized. To workaround this issue it is recommended to manually iterate over data using an **Iterator**, or to execute lambdas after the data has been transferred from the script to the originator node.

There are no issues using streams in clusters with other modes.

CHAPTER 41. REMOTE TASK EXECUTION

41.1. REMOTE TASK EXECUTION

Tasks, or business logic, can run directly on JBoss Data Grid servers, which means task execution is close to the data and uses the resources of all nodes in the cluster.

You can bundle tasks in Java executable files and deploy them to server instances where you can run the executables programmatically.

41.2. CREATING REMOTE TASKS

To create a task for remote execution, you must create a **.jar** file that contains a class that implements the **org.infinispan.tasks.ServerTask** interface.

The following methods are required in the implementation:

- **void setTaskContext(TaskContext taskContext)** Sets the task context. Use this method to access caches and other necessary resources.
- **String getName()** Provides a unique name for the task. This name is used for execution by **TaskManager**.

The following methods are optional in the implementation:

- **TaskExecutionMethod getExecutionMode()** Determines if the task is executed on one node, as in **TaskExecutionMode.ONE_NODE**, or on all nodes, as in **TaskExecutionMode.ALL_NODES**. Execution on one node is the default.
- **Optional<String> getAllowedRole()** Sets a role that users must have to run a task. No additional user role is set by default. For more information, see Running Remote Tasks.
- **Set<String> getParameters()** Specifies named parameters for use with the task.

41.3. REMOTE TASK EXAMPLE

The following provides an example class that implements the **org.infinispan.tasks.ServerTask** interface:

```
public class HelloTask implements ServerTask<String> {

  private TaskContext ctx;

  //Set the task context.
  @Override
  public void setTaskContext(TaskContext ctx) {
    this.ctx = ctx;
  }

  //Take the name of a person as a parameter.
  //Return a greeting with that person's name.
  @Override
  public String call() throws Exception {
    String name = (String) ctx.getParameters().get().get("name");
```

```
    return "Hello " + name;
  }

  //Return a unique name that clients can use to invoke the task.
  @Override
  public String getName() {
    return "hello-task";
  }

}
```

41.4. INSTALLING REMOTE TASKS

After you create a remote task and bundle it into a **.jar** file, you can deploy it to a JBoss Data Grid server instance with one of the following options:

Option 1: Copy to the Deployments Directory

1. Copy the **.jar** file to the *deployments/* directory.

```
$] cp /path/to/sample_task.jar $JDG_HOME/standalone/deployments/
```

Option 2: Deploy with the CLI

1. Connect to the JBoss Data Grid server.

```
[$JDG_HOME] $ bin/cli.sh --connect --controller=$IP:$PORT
```

2. Deploy the **.jar** file.

```
deploy /path/to/sample_task.jar
```

NOTE

If JBoss Data Grid is in domain mode, you must specify the server groups with either the **--all-server-groups** or **--server-groups** parameter.

41.5. REMOVING REMOTE TASKS

You can remove remote tasks from the running instances of JBoss Data Grid as follows:

1. Connect to the JBoss Data Grid server.

```
[$JDG_HOME] $ bin/cli.sh --connect --controller=$IP:$PORT
```

2. Run the **undeploy** command to remove the **.jar** file.

```
undeploy /path/to/sample_task.jar
```

NOTE

If JBoss Data Grid is in domain mode, you must specify the server groups with either the **--all-server-groups** or **--server-groups** parameter.

41.6. RUNNING REMOTE TASKS

If authorization is enabled on the JBoss Data Grid server, only users with **EXEC** permissions can run remote tasks. If authorization is not enabled, any user can run remote tasks.

Remote tasks can have additional user roles specified with the **getAllowedRole** method. In this case, users must belong to the role to run remote tasks.

To execute a previously deployed task call **execute(String taskName, Map parameters)** on the desired cache.

The following example shows how to run a task named **sampleTask**:

```
import org.infinispan.client.hotrod.*;
import java.util.*;
[...]
String TASK_NAME = "sampleTask";

RemoteCacheManager rcm = new RemoteCacheManager();
RemoteCache remoteCache = rcm.getCache();

// Assume the task takes a single parameter, and will return a result
Map<String, String> params = new HashMap<>();
params.put("name", "James");

String result = (String) remoteCache.execute(TASK_NAME, params);
```

CHAPTER 42. CONFIGURING MEDIA TYPES

Red Hat JBoss Data Grid lets you configure the media type for data in the cache. In other words, you can set a media type that defines the storage format for data in the cache.

JBoss Data Grid allows clients to write and read data in different storage formats and automatically converts between formats when necessary.

42.1. DEFAULT MEDIA TYPE

The default media type is **application/octet-stream** for both keys and values with the following exceptions:

- Indexed caches have a default media type of **application/x-protostream**.
- Caches that use compatibility mode have a default media type of **application/x-java-object**.

42.2. SUPPORTED MEDIA TYPES

Red Hat JBoss Data Grid lets clients read and write data in different formats and automatically converts between formats.

JBoss Data Grid supports several data formats that are interchangeable with one another, as follows:

- **application/x-java-object**
- **application/octet-stream**
- **application/x-www-form-urlencoded**
- **text/plain**

JBoss Data Grid also supports data formats that it must convert to and from the data formats in the preceding list, as follows:

- **application/xml**
- **application/json**
- **application/x-jboss-marshalling**
- **application/x-java-serialized**
- **application/x-protostream**

In addition, JBoss Data Grid supports conversion between **application/x-protostream** and **application/json**.

42.3. DECLARATIVELY CONFIGURING MEDIA TYPES

The following is an example configuration that defines the media type for keys and values:

```
<cache>
  <encoding>
    <key media-type="application/x-java-object; type=java.lang.Integer"/>
```

```
    <value media-type="application/xml; charset=UTF-8"/>
  </encoding>
</cache>
```

42.4. PROGRAMMATICALLY CONFIGURING MEDIA TYPES

Use the **ConfigurationBuilder** interface to programmatically configure the media type, as in the following example:

```
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg.encoding().key().mediaType("text/plain");
cfg.encoding().value().mediaType("application/json");
```

42.5. OVERRIDING MEDIA TYPES

You can programmatically override the media type that is configured for the cache, as in the following example:

```
DefaultCacheManager cacheManager = new DefaultCacheManager();

// The cache will store POJO for keys and values
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.encoding().key().mediaType("application/x-java-object");
cfg.encoding().key().mediaType("application/x-java-object");

cacheManager.defineConfiguration("mycache", cfg.build());

Cache<Integer, Person> cache = cacheManager.getCache("mycache");

cache.put(1, new Person("John","Doe"));

// Wraps cache using 'application/x-java-object' for keys but JSON for values.
cache.getAdvancedCache().withMediaType("application/x-java-object", "application/json");

byte[] json = cache.get(1);
```

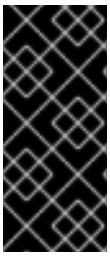
This configuration returns the value in JSON format, as follows:

```
{
  "_type":"org.infinispan.sample.Person",
  "name":"John",
  "surname":"Doe"
}
```

CHAPTER 43. CONFIGURING COMPATIBILITY MODE

Compatibility mode provides a mechanism for accessing data in the cache from multiple endpoints.

Compatibility mode configures JBoss Data Grid to use a marshaller that serializes and deserializes raw bytes into strings and primitives. For this reason, compatibility mode supports only strings and primitives and does not support objects.



IMPORTANT

Compatibility mode is not an efficient method for achieving interoperability between remote endpoints. It is a legacy feature that is not recommended for new deployments.

Instead of using compatibility mode, you should configure the format in which the cache stores data by defined the media type. See Endpoint Interoperability.

43.1. ENABLING COMPATIBILITY MODE

To enable compatibility mode, add **enabled=true** to the **compatibility** element as follows:

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

43.2. MARSHALLERS IN COMPATIBILITY MODE

JBoss Data Grid does not support custom marshallers. You can use the following marshallers in compatibility mode:

Marshaller	Description
GenericJBossMarshaller	Uses the JBoss marshaller to serialize and deserialize strings and primitives as byte arrays. This is the default marshaller in compatibility mode.
ProtoStreamCompatibilityMarshaller	Uses the ProtoStream library to serialize and deserialize strings and primitives as byte arrays.
UTF8StringMarshaller	Serializes and deserializes strings and primitives as UTF8 byte arrays.

43.3. SPECIFYING THE MARSHALLER

Specify marshallers with the **marshaller** attribute, as in the following example:

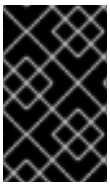
```
<cache-container name="local" default-cache="default" statistics="true">
   <local-cache name="default" statistics="true">
    <compatibility enabled="true"
```

```
marshaller="org.infinispan.commons.marshall.UTF8StringMarshaller"/>
    </local-cache>
</cache-container>
```

43.3.1. Memcached Marshaller

When using memcached in compatibility mode, you must explicitly set the default marshaller, **GenericJBossMarshaller**, in the configuration. For example:

```
<cache-container name="local" default-cache="default" statistics="true">
    <local-cache name="default" statistics="true">
        <compatibility enabled="true"
marshaller="org.infinispan.commons.marshall.jboss.GenericJBossMarshaller"/>
    </local-cache>
</cache-container>
```



IMPORTANT

Java clients must use a transcoder to perform read and write operations in compatibility mode. The transcoder enables memcached clients written in Java to convert between byte arrays and strings or primitives.

CHAPTER 44. ENDPOINT INTEROPERABILITY

Clients exchange data with Red Hat JBoss Data Grid through endpoints such as REST or Hot Rod.

Each endpoint uses a different protocol so that clients can read and write data in the cache in a suitable storage format. Because JBoss Data Grid can interoperate with multiple clients at the same time, it must convert data between client formats and the storage formats.

To configure JBoss Data Grid endpoint interoperability, you should define the media type that sets the format for data stored in the cache.

44.1. CONSIDERATIONS WITH MEDIA TYPES AND ENDPOINT INTEROPERABILITY

Configuring Red Hat JBoss Data Grid to store data with a specific media type affects client interoperability.

REST clients generally handle text formats such as JSON, XML, or plain text better than binary formats. Although the JBoss Data Grid REST API does handle binary formats represented as String, encoded in hexadecimal or base64 format.

Java Hot Rod clients are suitable for handling Java objects that represent entities that reside in the cache. Java Hot Rod clients use marshalling operations to serialize and deserialize those objects into byte arrays.

Similarly, non-Java Hot Rod clients, such as the C++, C#, and Javascript clients, are suitable for handling objects in the respective languages. However, non-Java Hot Rod clients can interoperate with Java Hot Rod clients using platform independent data formats.

44.1.1. REST and Hot Rod Interoperability with Text-Based Storage

You can configure key and values with a text-based storage format.

For example, specify **text/plain; charset=UTF-8**, or any other character set, to set plain text as the media type. You can also specify a media type for other text-based formats such as JSON (**application/json**) or XML (**application/xml**) with an optional character set.

The following example configures the cache to store entries with the **text/plain; charset=UTF-8** media type:

```
<cache>
  <encoding>
    <key media-type="text/plain; charset=UTF-8"/>
    <value media-type="text/plain; charset=UTF-8"/>
  </encoding>
</cache>
```

To handle the exchange of data in a text-based format, you must configure Hot Rod clients with the **org.infinispan.commons.marshall.StringMarshaller** marshaller.

REST clients must also send the correct headers when writing and reading from the cache, as follows:

- Write: **Content-Type: text/plain; charset=UTF-8**
- Read: **Accept: text/plain; charset=UTF-8**

This configuration is compatible with...	
REST clients	Yes
Java Hot Rod clients	Yes
Non-Java Hot Rod clients	No
Querying and Indexing	No
Custom Java objects	No

44.1.2. Java and Non-Java Client Interoperability with Protobuf

Storing data in the cache as Protobuf encoded entries provides a platform independent configuration that enables Java and Non-Java clients to access and query the cache from any endpoint.

If indexing is configured for the cache, JBoss Data Grid automatically stores keys and values with the **application/x-protostream** media type.

If indexing is not configured for the cache, you can configure it to store entries with the **application/x-protostream** media type as follows:

```
<distributed-cache name="my-cache">
  <encoding>
    <key media-type="application/x-protostream"/>
    <value media-type="application/x-protostream"/>
  </encoding>
</distributed-cache>
```

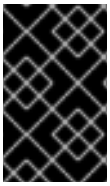
JBoss Data Grid converts between **application/x-protostream** and **application/json**, which allows REST clients to read and write JSON formatted data. However REST clients must send the correct headers, as follows:

Read Header

```
Read: Accept: application/json
```

Write Header

```
Write: Content-Type: application/json
```



IMPORTANT

The **application/x-protostream** media type uses Protobuf encoding, which requires you to register a Protocol Buffers schema definition that describes the entities and marshallers that the clients use. See Protobuf Encoding.

This configuration is compatible with...	
REST clients	Yes
Java Hot Rod clients	Yes
Non-Java Hot Rod clients	Yes
Querying and Indexing	Yes
Custom Java objects	Yes

CHAPTER 45. SET UP CROSS-DATACENTER REPLICATION

45.1. CROSS-DATACENTER REPLICATION

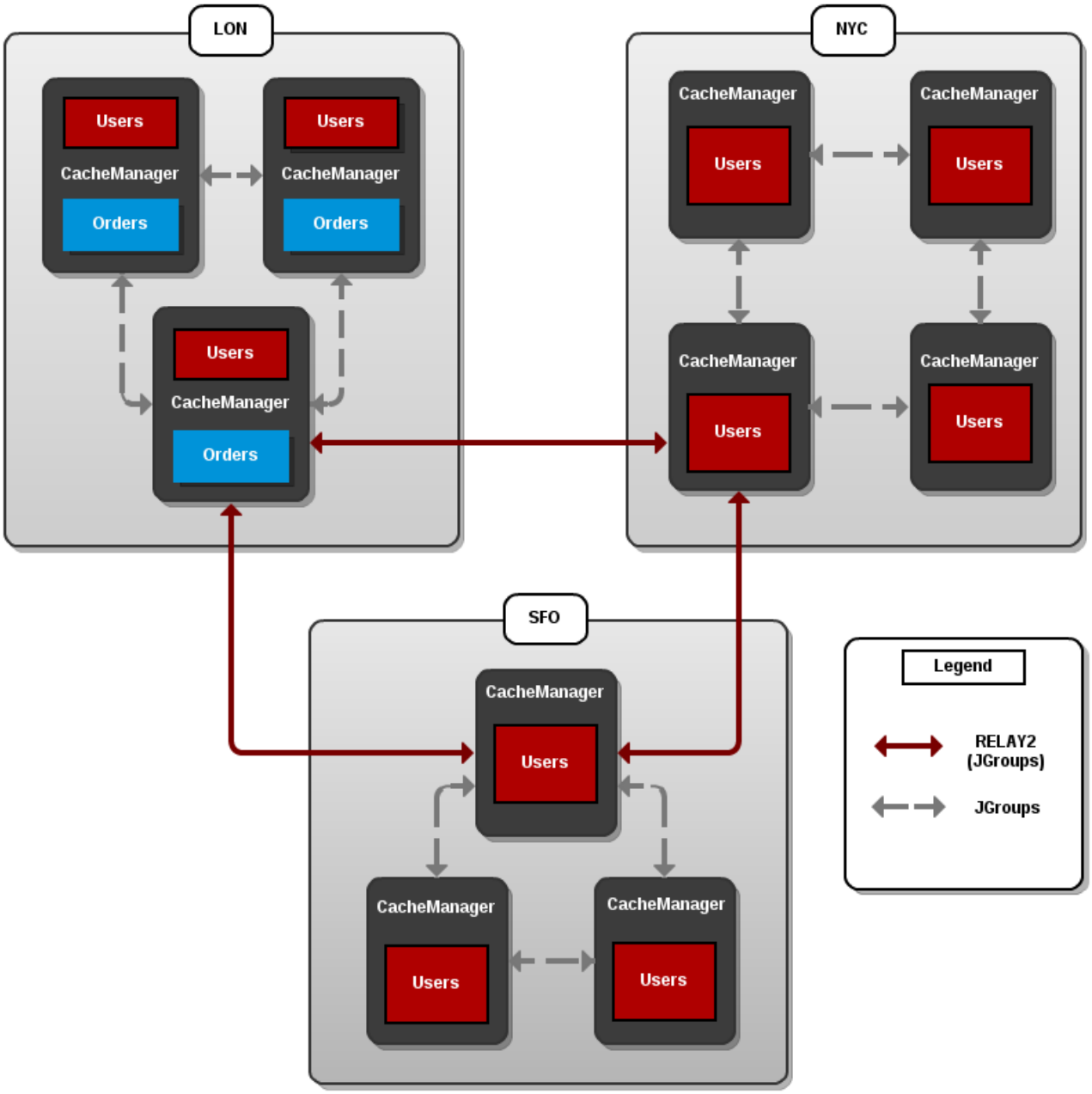
In Red Hat JBoss Data Grid, Cross-Datacenter Replication allows the administrator to create data backups in multiple clusters. These clusters can be at the same physical location or different ones. JBoss Data Grid's Cross-Site Replication implementation is based on JGroups' *RELAY2* protocol.

Cross-Datacenter Replication ensures data redundancy across clusters. In addition to creating backups for data restoration, these datasets may also be used in an active-active mode. When configured in this manner systems in separate environments are able to handle sessions should one cluster fail. Ideally, each of these clusters should be in a different physical location than the others.

45.2. CROSS-DATACENTER REPLICATION OPERATIONS

Red Hat JBoss Data Grid's Cross-Datacenter Replication operation is explained through the use of an example, as follows:

Figure 45.1. Cross-Datacenter Replication Example



Three sites are configured in this example: **LON**, **NYC** and **SFO**. Each site hosts a running JBoss Data Grid cluster made up of three to four physical nodes.

The **Users** cache is active in all three sites - **LON**, **NYC** and **SFO**. Changes to the **Users** cache at the any one of these sites will be replicated to the other two as long as the cache defines the other two sites as its backups through configuration. The **Orders** cache, however, is only available locally at the **LON** site because it is not replicated to the other sites.

The **Users** cache can use different replication mechanisms each site. For example, it can back up data synchronously to **SFO** and asynchronously to **NYC** and **LON**.

The **Users** cache can also have a different configuration from one site to another. For example, it can be configured as a distributed cache with **owners** set to **2** in the **LON** site, as a replicated cache in the **NYC** site and as a distributed cache with **owners** set to **1** in the **SFO** site.

JGroups is used for communication within each site as well as inter-site communication. Specifically, a JGroups protocol called *RELAY2* facilitates communication between sites. For more information, refer to the *RELAY2* section in the JBoss Data Grid *Administration Guide* .

45.3. CONFIGURE CROSS-DATACENTER REPLICATION PROGRAMMATICALLY

The programmatic method to configure cross-datacenter replication in Red Hat JBoss Data Grid is as follows:

Configure Cross-Datacenter Replication Programmatically

1. Identify the Node Location
   Declare the site the node resides in:

```
globalConfiguration.site().localSite("LON");
```

2. Configure JGroups
   Configure JGroups to use the *RELAY* protocol:

```
globalConfiguration.transport().addProperty("configurationFile","jgroups-with-relay.xml");
```

3. Set Up the Remote Site
   Set up JBoss Data Grid caches to replicate to the remote site:

```
ConfigurationBuilder lon = new ConfigurationBuilder();
lon.sites().addBackup()
      .site("NYC")
      .backupFailurePolicy(BackupFailurePolicy.WARN)
      .strategy(BackupConfiguration.BackupStrategy.SYNC)
      .replicationTimeout(12000)
      .sites().addInUseBackupSite("NYC")
    .sites().addBackup()
      .site("SFO")
      .backupFailurePolicy(BackupFailurePolicy.IGNORE)
      .strategy(BackupConfiguration.BackupStrategy.ASYNC)
      .sites().addInUseBackupSite("SFO")
```

4. Optional: Configure the Backup Caches
   JBoss Data Grid implicitly replicates data to a cache with same name as the remote site. If a backup cache on the remote site has a different name, users must specify a **backupFor** cache to ensure data is replicated to the correct cache.

NOTE

This step is optional and only required if the remote site's caches are named differently from the original caches.

   a. Configure the cache in site **NYC** to receive backup data from **LON**:

```
ConfigurationBuilder NYCbackupOfLon = new ConfigurationBuilder();
NYCbackupOfLon.sites().backupFor().remoteCache("lon").remoteSite("LON");
```

b. Configure the cache in site **SFO** to receive backup data from **LON**:

```
ConfigurationBuilder SFObackupOfLon = new ConfigurationBuilder();
SFObackupOfLon.sites().backupFor().remoteCache("lon").remoteSite("LON");
```

5. Add the Contents of the Configuration File
As a default, Red Hat JBoss Data Grid includes JGroups configuration files such as *default-configs/default-jgroups-tcp.xml* and *default-configs/default-jgroups-udp.xml* in the *infinispan-embedded-{VERSION}.jar* package.

Copy the JGroups configuration to a new file (in this example, it is named *jgroups-with-relay.xml*) and add the provided configuration information to this file. Note that the *relay.RELAY2* protocol configuration must be the last protocol in the configuration stack.

```
<config>
   <!-- Additional configuration information here -->
   <relay.RELAY2 site="LON"
            config="relay.xml"
            relay_multicasts="false" />
</config>
```

6. Configure the relay.xml File
Set up the *relay.RELAY2* configuration in the *relay.xml* file. This file describes the global cluster configuration.

```
<RelayConfiguration>
   <sites>
      <site name="LON"
           id="0">
         <bridges>
            <bridge config="jgroups-global.xml"
                 name="global"/>
            </bridges>
      </site>
      <site name="NYC"
           id="1">
         <bridges>
            <bridge config="jgroups-global.xml"
                 name="global"/>
            </bridges>
      </site>
      <site name="SFO"
           id="2">
         <bridges>
            <bridge config="jgroups-global.xml"
                 name="global"/>
         </bridges>
      </site>
   </sites>
</RelayConfiguration>
```

7. Configure the Global Cluster
The file *jgroups-global.xml* referenced in *relay.xml* contains another JGroups configuration which is used for the global cluster: communication between sites.

The global cluster configuration is usually *TCP* -based and uses the *TCPPING* protocol (instead of *PING* or *MPING*) to discover members. Copy the contents of *default-configs/default-jgroups-tcp.xml* into *jgroups-global.xml* and add the following configuration in order to configure *TCPPING* :

```
<config>
   <TCP bind_port="7800" <!-- Additional configuration information here --> />
   <TCPPING initial_hosts="lon.hostname[7800],nyc.hostname[7800],sfo.hostname[7800]"
            ergonomics="false" />
        <!-- Rest of the protocols -->
</config>
```

Replace the hostnames (or IP addresses) in **TCPPING.initial_hosts** with those used for your site masters. The ports (**7800** in this example) must match the **TCP.bind_port**.

For more information about the *TCPPING* protocol, refer to the JBoss Data Grid *Administration and Configuration Guide* .

45.4. TAKING A SITE OFFLINE

In Red Hat JBoss Data Grid's Cross-datacenter replication configuration, if backing up to one site fails a certain number of times during a time interval, that site can be marked as offline automatically. This feature removes the need for manual intervention by an administrator to mark the site as offline.

To configure taking a Cross-datacenter replication site offline automatically in Red Hat JBoss Data Grid programmatically:

Taking a Site Offline Programmatically

```
lon.sites().addBackup()
      .site("NYC")
      .backupFailurePolicy(BackupFailurePolicy.FAIL)
      .strategy(BackupConfiguration.BackupStrategy.SYNC)
      .takeOffline()
        .afterFailures(500)
        .minTimeToWait(10000);
```

45.5. HOT ROD CROSS SITE CLUSTER FAILOVER

Besides in-cluster failover, Hot Rod clients can failover to different clusters each representing independent sites. Hot Rod Cross Site cluster failover is available in both automatic and manual modes.

Automatic Cross Site Cluster Failover

If the main/primary cluster nodes are unavailable, the client application checks for alternatively defined clusters and will attempt to failover to them. Upon successful failover, the client will remain connected to the alternative cluster until it becomes unavailable. After that, the client will try to failover to other defined clusters and finally switch over to the main/primary cluster with the original server settings if the connectivity is restored.

To configure an alternative cluster in the Hot Rod client, provide details of at least one host/port pair for each of the clusters configured as shown in the following example.

Configure Alternate Cluster

-

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
    = new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.addCluster("remote-cluster").addClusterNode("remote-cluster-host", 11222);
RemoteCacheManager rcm = new RemoteCacheManager(cb.build());
```

NOTE

Regardless of the cluster definitions, the initial server(s) configuration must be provided unless the initial servers can be resolved using the default server host and port details.

Manual Cross Site Cluster Failover

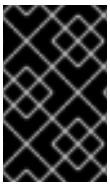
For manual site cluster switchover, call RemoteCacheManager's **switchToCluster(clusterName)** or **switchToDefaultCluster()**.

Using **switchToCluster(clusterName)**, users can force a client to switch to one of the clusters predefined in the Hot Rod client configuration. To switch to the default cluster use **switchToDefaultCluster()** instead.

CHAPTER 46. NEAR CACHING

46.1. NEAR CACHING

Near caches are optional caches for Hot Rod Java client implementations that keep recently accessed data close to the user, providing faster access to data that is accessed frequently. This cache acts as a local Hot Rod client cache that is updated whenever a remote entry is retrieved via **get** or **getVersioned** operations.



IMPORTANT

Near Caching for Library mode, or non-Hot Rod interfaces, is achieved by configuring L1 Caches. Configuring L1 Caches are documented in the JBoss Data Grid *Administration and Configuration Guide* .

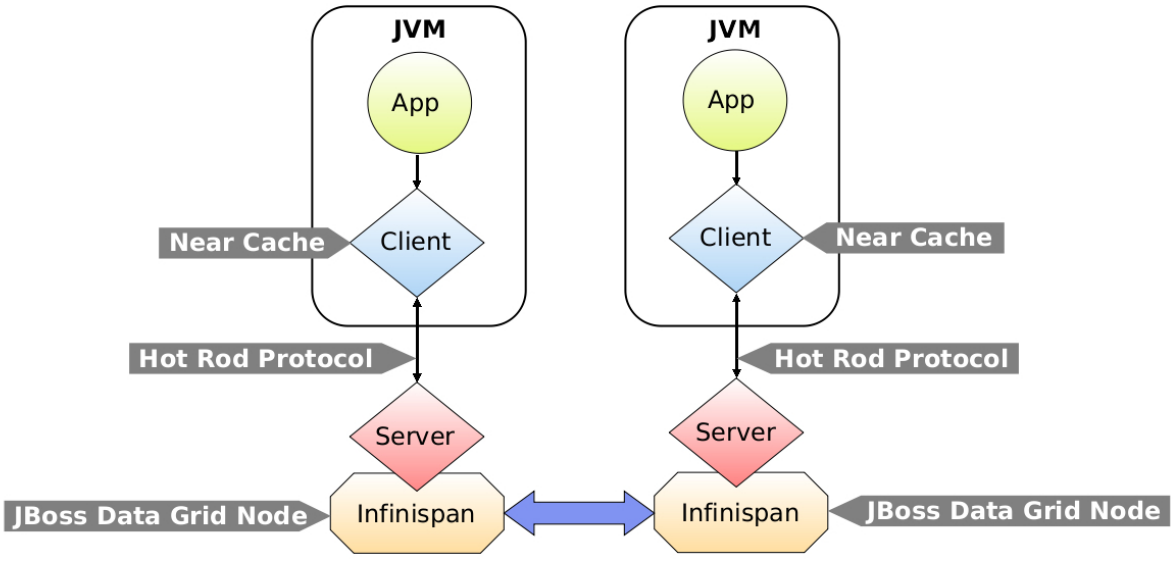
In Red Hat JBoss Data Grid, near cache consistency is achieved by using remote events, which send notifications to clients when entries are modified or removed (refer to Remote Event Listeners (Hot Rod)). With Near Caching, local cache remains consistent with remote cache. Local entry is updated or invalidated whenever remote entry on the server is updated or removed. At the client level, near caching is configurable as either of the following:

- **DISABLED** - the default mode, indicating that Near Caching is not enabled.
- **INVALIDATED** - enables near caching, keeping it in sync with the remote cache via invalidation messages.

NOTE

Near caching is disabled for Hot Rod clients by default.

Figure 46.1. Near Caching Architecture



46.2. CONFIGURING NEAR CACHES

Near caching can be enabled and disabled via configuration without making any changes to the Hot Rod Client application. To enable near caching, configure the near caching mode as **INVALIDATED** on the client, and optionally specify the number of entries to be kept in the cache.

Near cache mode is configured using the **NearCacheMode** enumeration.

The following example demonstrates how to configure near caching:

Enabling a Near Cache

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.configuration.NearCacheMode;
…

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.nearCache().mode(NearCacheMode.INVALIDATED).maxEntries(100);
```

A maximum size for the near cache must be provided, using the **maxEntries(int maxEntries)** method. In the above example this is defined to 100. When the maximum size is reached, near cached entries are evicted using a least-recently-used (LRU) algorithm. To define an unlimited near cache, a 0 or negative value may be passed in.

46.3. NEAR CACHES IN A CLUSTERED ENVIRONMENT

Near caches are implemented using Hot Rod Remote Events, and utilize clustered listeners for receiving events from across the cluster. Clustered listeners are installed on a single node within the cluster, with the remaining nodes sending events to the node on which the listeners are installed. It is therefore possible for a node running the near cache-backing clustered listener to fail. In this situation, another node takes over the clustered listener.

When the node running the clustered listener fails, a client failover event callback can be defined and invoked. For near caches, this callback and its implementation will clear the near cache, as during a failover events may be missed.

Refer to Clustered Listeners for more information.

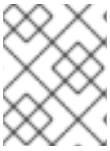
CHAPTER 47. CONFLICT MANAGER USAGE

47.1. FIND AND RESOLVE CACHE CONFLICTS

The Conflict Manager is often used with Partition Handling. A split-brain occurs when nodes in a cluster are separated into two or more groups (partitions) that can't communicate with each other. In some split-brain situations, nodes can have different data written to them. In this case, JBoss Data Grid's Partition Handling, combined with its Conflict Manager, can be used to automatically resolve differences in the same **CacheEntries** across nodes. The Conflict Manager can also be used to manually search for and resolve conflicts.

The code below shows how to retrieve an **EmbeddedCacheManager**'s **ConflictManager**, how to retrieve all versions of a given key, and how to check for conflicts across a given cache.

```
EmbeddedCacheManager manager = new DefaultCacheManager("example-config.xml");
Cache<Integer, String> cache = manager.getCache("testCache");
ConflictManager<Integer, String> crm = ConflictManagerFactory.get(cache.getAdvancedCache());

// Get All Versions of Key
Map<Address, InternalCacheValue<String>> versions = crm.getAllVersions(1);

// Process conflicts stream and perform some operation on the cache
Stream<Map<Address, InternalCacheEntry<Integer, String>>> stream = crm.getConflicts();
stream.forEach(map -> {
   CacheEntry<Object, Object> entry = map.values().iterator().next();
   Object conflictKey = entry.getKey();
   cache.remove(conflictKey);
});

// Detect and then resolve conflicts using the configured EntryMergePolicy
crm.resolveConflicts();

// Detect and then resolve conflicts using the passed EntryMergePolicy instance
crm.resolveConflicts((preferredEntry, otherEntries) -> preferredEntry);
```



NOTE

Although the **ConflictManager::getConflicts** stream is processed per entry, the underlying spliterator is in fact lazily-loading cache entries on a per segment basis.

APPENDIX A. REFERENCES

A.1. THE EXTERNALIZER

A.1.1. About Externalizer

An **Externalizer** is a class that can:

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

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

A.1.2. Internal Externalizer Implementation Access

Externalizable objects should not access Red Hat JBoss Data Grids Externalizer implementations. The following is an example of incorrect usage:

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

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

End user externalizers do not need to interact with internal externalizer classes. The following is an example of correct usage:

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

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

```
    }

    <!-- Additional configuration information here -->
}
```

A.2. HASH SPACE ALLOCATION

A.2.1. About Hash Space Allocation

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

A.2.2. Locating a Key in the Hash Space

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