Red Hat Data Grid 8.3

Data Grid Developer Guide

Use Data Grid APIs to customize, configure, and extend your deployment

Red Hat Data Grid 8.3 Data Grid Developer Guide

Use Data Grid APIs to customize, configure, and extend your deployment

Legal Notice

Abstract

Use Data Grid APIs to customize, configure, and extend your deployment.

Table of Contents

RED HAT DATA GRID

Data Grid is a high-performance, distributed in-memory data store.

Schemaless data structure

Flexibility to store different objects as key-value pairs.

Grid-based data storage

Designed to distribute and replicate data across clusters.

Elastic scaling

Dynamically adjust the number of nodes to meet demand without service disruption.

Data interoperability

Store, retrieve, and query data in the grid from different endpoints.

DATA GRID DOCUMENTATION

Documentation for Data Grid is available on the Red Hat customer portal.

- Data Grid 8.3 Documentation

- Data Grid 8.3 Component Details

- Supported Configurations for Data Grid 8.3

- Data Grid 8 Feature Support

- Data Grid Deprecated Features and Functionality

DATA GRID DOWNLOADS

Access the Data Grid Software Downloads on the Red Hat customer portal.

NOTE

You must have a Red Hat account to access and download Data Grid software.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message .

CHAPTER 1. CONFIGURING THE DATA GRID MAVEN REPOSITORY

Data Grid Java distributions are available from Maven.

You can download the Data Grid Maven repository from the customer portal or pull Data Grid dependencies from the public Red Hat Enterprise Maven repository.

1.1. DOWNLOADING THE DATA GRID MAVEN REPOSITORY

Download and install the Data Grid Maven repository to a local file system, Apache HTTP server, or Maven repository manager if you do not want to use the public Red Hat Enterprise Maven repository.

Procedure

1. Log in to the Red Hat customer portal.

2. Navigate to the Software Downloads for Data Grid.

3. Download the Red Hat Data Grid 8.3 Maven Repository.

4. Extract the archived Maven repository to your local file system.

5. Open the **README.md** file and follow the appropriate installation instructions.

1.2. ADDING RED HAT MAVEN REPOSITORIES

Include the Red Hat GA repository in your Maven build environment to get Data Grid artifacts and dependencies.

Procedure

- Add the Red Hat GA repository to your Maven settings file, typically **~/.m2/settings.xml**, or directly in the **pom.xml** file of your project.

```xml
<repositories>
  <repository>
    <id>redhat-ga-repository</id>
    <name>Red Hat GA Repository</name>
    <url>https://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>redhat-ga-repository</id>
    <name>Red Hat GA Repository</name>
    <url>https://maven.repository.redhat.com/ga/</url>
  </pluginRepository>
</pluginRepositories>
```

Reference

- Red Hat Enterprise Maven Repository

1.3. CONFIGURING YOUR DATA GRID POM

Maven uses configuration files called Project Object Model (POM) files to define projects and manage builds. POM files are in XML format and describe the module and component dependencies, build order, and targets for the resulting project packaging and output.

Procedure

1. Open your project **pom.xml** for editing.

2. Define the **version.infinispan** property with the correct Data Grid version.

3. Include the **infinispan-bom** in a **dependencyManagement** section.
   The Bill Of Materials (BOM) controls dependency versions, which avoids version conflicts and means you do not need to set the version for each Data Grid artifact you add as a dependency to your project.

4. Save and close **pom.xml**.

The following example shows the Data Grid version and BOM:

```xml
<properties>
  <version.infinispan>13.0.10.Final-redhat-00001</version.infinispan>
</properties>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-bom</artifactId>
      <version>${version.infinispan}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Next Steps

Add Data Grid artifacts as dependencies to your **pom.xml** as required.

CHAPTER 2. CACHE MANAGERS

The main entry point to Data Grid is the **CacheManager** interface that lets you:

- Configure and obtain caches.

- Manage and monitor clustered Data Grid nodes.

- Execute code across your cluster.

If you embed Data Grid in your application, then you use an **EmbeddedCacheManager**. If you run Data Grid as a remote server, then you use a **RemoteCacheManager**.

Cache Managers are heavyweight objects so you should instantiate only one **CacheManager** instance per JVM in most cases.

```
EmbeddedCacheManager manager = new DefaultCacheManager(); ❶
```

❶ Starts a local, non-clustered, Cache Manager with no caches.

Cache Managers have lifecycles and the default constructors also call the **start()** method. Overloaded versions of the constructors are available, but they do not start the **CacheManager**. However, you must always start the **CacheManager** before you can create caches.

Likewise, you must call **stop()** when you no longer require a running **CacheManager** so that it releases resources. This also ensures that the Cache Manager safely stops any caches that it controls.

2.1. OBTAINING CACHES

After you configure the **CacheManager**, you can obtain and control caches.

Invoke the **getCache(String)** method to obtain caches, as follows:

```
Cache<String, String> myCache = manager.getCache("myCache");
```

The preceding operation creates a cache named **myCache**, if it does not already exist, and returns it.

Using the **getCache()** method creates the cache only on the node where you invoke the method. In other words, it performs a local operation that must be invoked on each node across the cluster. Typically, applications deployed across multiple nodes obtain caches during initialization to ensure that caches are *symmetric* and exist on each node.

Invoke the **createCache()** method to create caches dynamically across the entire cluster, as follows:

```
Cache<String, String> myCache = manager.administration().createCache("myCache",
"myTemplate");
```

The preceding operation also automatically creates caches on any nodes that subsequently join the cluster.

Caches that you create with the **createCache()** method are ephemeral by default. If the entire cluster shuts down, the cache is not automatically created again when it restarts.

Use the PERMANENT flag to ensure that caches can survive restarts, as follows:

```
Cache<String, String> myCache =
manager.administration().withFlags(AdminFlag.PERMANENT).createCache("myCache",
"myTemplate");
```

For the PERMANENT flag to take effect, you must enable global state and set a configuration storage provider.

For more information about configuration storage providers, see GlobalStateConfigurationBuilder#configurationStorage().

2.2. CLUSTERING INFORMATION

The **EmbeddedCacheManager** has quite a few methods to provide information as to how the cluster is operating. The following methods only really make sense when being used in a clustered environment (that is when a Transport is configured).

2.3. MEMBER INFORMATION

When you are using a cluster it is very important to be able to find information about membership in the cluster including who is the owner of the cluster.

getMembers()

The getMembers() method returns all of the nodes in the current cluster.

getCoordinator()

The getCoordinator() method will tell you which one of the members is the coordinator of the cluster. For most intents you shouldn't need to care who the coordinator is. You can use isCoordinator() method directly to see if the local node is the coordinator as well.

CHAPTER 3. CACHE INTERFACE

Data Grid provides a Cache interface that exposes simple methods for adding, retrieving and removing entries, including atomic mechanisms exposed by the JDK's ConcurrentMap interface. Based on the cache mode used, invoking these methods will trigger a number of things to happen, potentially even including replicating an entry to a remote node or looking up an entry from a remote node, or potentially a cache store.

3.1. CACHE API

For simple usage, using the Cache API should be no different from using the JDK Map API, and hence migrating from simple in-memory caches based on a Map to Data Grid's Cache should be trivial.

3.1.1. Performance Concerns of Certain Map Methods

Certain methods exposed in Map have certain performance consequences when used with Data Grid, such as size() , values() , keySet() and entrySet() . Specific methods on the **keySet**, **values** and **entrySet** are fine for use please see their Javadoc for further details.

Attempting to perform these operations globally would have large performance impact as well as become a scalability bottleneck. As such, these methods should only be used for informational or debugging purposes only.

It should be noted that using certain flags with the withFlags() method can mitigate some of these concerns, please check each method's documentation for more details.

3.1.2. Mortal and Immortal Data

Further to simply storing entries, Data Grid's cache API allows you to attach mortality information to data. For example, simply using put(key, value) would create an *immortal* entry, i.e., an entry that lives in the cache forever, until it is removed (or evicted from memory to prevent running out of memory). If, however, you put data in the cache using put(key, value, lifespan, timeunit) , this creates a *mortal* entry, i.e., an entry that has a fixed lifespan and expires after that lifespan.

In addition to *lifespan* , Data Grid also supports *maxIdle* as an additional metric with which to determine expiration. Any combination of lifespans or maxIdles can be used.

3.1.3. putForExternalRead operation

Data Grid's Cache class contains a different 'put' operation called putForExternalRead . This operation is particularly useful when Data Grid is used as a temporary cache for data that is persisted elsewhere. Under heavy read scenarios, contention in the cache should not delay the real transactions at hand, since caching should just be an optimization and not something that gets in the way.

To achieve this, **putForExternalRead()** acts as a put call that only operates if the key is not present in the cache, and fails fast and silently if another thread is trying to store the same key at the same time. In this particular scenario, caching data is a way to optimise the system and it's not desirable that a failure in caching affects the on-going transaction, hence why failure is handled differently. **putForExternalRead()** is considered to be a fast operation because regardless of whether it's successful or not, it doesn't wait for any locks, and so returns to the caller promptly.

To understand how to use this operation, let's look at basic example. Imagine a cache of Person instances, each keyed by a PersonId , whose data originates in a separate data store. The following code shows the most common pattern of using putForExternalRead within the context of this example:

–

```
// Id of the person to look up, provided by the application
PersonId id = ...;

// Get a reference to the cache where person instances will be stored
Cache<PersonId, Person> cache = ...;

// First, check whether the cache contains the person instance
// associated with with the given id
Person cachedPerson = cache.get(id);

if (cachedPerson == null) {
   // The person is not cached yet, so query the data store with the id
   Person person = dataStore.lookup(id);

   // Cache the person along with the id so that future requests can
   // retrieve it from memory rather than going to the data store
   cache.putForExternalRead(id, person);
} else {
   // The person was found in the cache, so return it to the application
   return cachedPerson;
}
```

Note that putForExternalRead should never be used as a mechanism to update the cache with a new Person instance originating from application execution (i.e. from a transaction that modifies a Person's address). When updating cached values, please use the standard put operation, otherwise the possibility of caching corrupt data is likely.

3.2. ADVANCEDCACHE API

In addition to the simple Cache interface, Data Grid offers an AdvancedCache interface, geared towards extension authors. The AdvancedCache offers the ability to access certain internal components and to apply flags to alter the default behavior of certain cache methods. The following code snippet depicts how an AdvancedCache can be obtained:

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

3.2.1. Flags

Flags are applied to regular cache methods to alter the behavior of certain methods. For a list of all available flags, and their effects, see the Flag enumeration. Flags are applied using AdvancedCache.withFlags() . This builder method can be used to apply any number of flags to a cache invocation, for example:

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

3.3. LISTENERS AND NOTIFICATIONS

Data Grid offers a listener API, where clients can register for and get notified when events take place. This annotation-driven API applies to 2 different levels: cache level events and cache manager level events.

Events trigger a notification which is dispatched to listeners. Listeners are simple POJOs annotated with @Listener and registered using the methods defined in the Listenable interface.

> **NOTE**
>
> Both Cache and CacheManager implement Listenable, which means you can attach listeners to either a cache or a cache manager, to receive either cache-level or cache manager-level notifications.

For example, the following class defines a listener to print out some information every time a new entry is added to the cache, in a non blocking fashion:

```
@Listener
public class PrintWhenAdded {
  Queue<CacheEntryCreatedEvent> events = new ConcurrentLinkedQueue<>();

  @CacheEntryCreated
  public CompletionStage<Void> print(CacheEntryCreatedEvent event) {
    events.add(event);
    return null;
  }

}
```

For more comprehensive examples, please see the Javadocs for @Listener.

3.3.1. Cache-level notifications

Cache-level events occur on a per-cache basis, and by default are only raised on nodes where the events occur. Note in a distributed cache these events are only raised on the owners of data being affected. Examples of cache-level events are entries being added, removed, modified, etc. These events trigger notifications to listeners registered to a specific cache.

Please see the Javadocs on the org.infinispan.notifications.cachelistener.annotation package for a comprehensive list of all cache-level notifications, and their respective method-level annotations.

> **NOTE**
>
> Please refer to the Javadocs on the org.infinispan.notifications.cachelistener.annotation package for the list of cache-level notifications available in Data Grid.

3.3.1.1. Cluster Listeners

The cluster listeners should be used when it is desirable to listen to the cache events on a single node.

To do so all that is required is set to annotate your listener as being clustered.

```
@Listener (clustered = true)
public class MyClusterListener { .... }
```

There are some limitations to cluster listeners from a non clustered listener.

1. A cluster listener can only listen to **@CacheEntryModified**, **@CacheEntryCreated**, **@CacheEntryRemoved** and **@CacheEntryExpired** events. Note this means any other type of event will not be listened to for this listener.

2. Only the post event is sent to a cluster listener, the pre event is ignored.

3.3.1.2. Event filtering and conversion

All applicable events on the node where the listener is installed will be raised to the listener. It is possible to dynamically filter what events are raised by using a KeyFilter (only allows filtering on keys) or CacheEventFilter (used to filter for keys, old value, old metadata, new value, new metadata, whether command was retried, if the event is before the event (ie. isPre) and also the command type).

The example here shows a simple **KeyFilter** that will only allow events to be raised when an event modified the entry for the key **Only Me**.

```
public class SpecificKeyFilter implements KeyFilter<String> {
   private final String keyToAccept;

   public SpecificKeyFilter(String keyToAccept) {
     if (keyToAccept == null) {
       throw new NullPointerException();
     }
     this.keyToAccept = keyToAccept;
   }

   public boolean accept(String key) {
     return keyToAccept.equals(key);
   }
}

...
cache.addListener(listener, new SpecificKeyFilter("Only Me"));
...
```

This can be useful when you want to limit what events you receive in a more efficient manner.

There is also a CacheEventConverter that can be supplied that allows for converting a value to another before raising the event. This can be nice to modularize any code that does value conversions.

> **NOTE**
>
> The mentioned filters and converters are especially beneficial when used in conjunction with a Cluster Listener. This is because the filtering and conversion is done on the node where the event originated and not on the node where event is listened to. This can provide benefits of not having to replicate events across the cluster (filter) or even have reduced payloads (converter).

3.3.1.3. Initial State Events

When a listener is installed it will only be notified of events after it is fully installed.

It may be desirable to get the current state of the cache contents upon first registration of listener by having an event generated of type **@CacheEntryCreated** for each element in the cache. Any additionally generated events during this initial phase will be queued until appropriate events have been

raised.

> **NOTE**
>
> This only works for clustered listeners at this time. ISPN-4608 covers adding this for non clustered listeners.

3.3.1.4. Duplicate Events

It is possible in a non transactional cache to receive duplicate events. This is possible when the primary owner of a key goes down while trying to perform a write operation such as a put.

Data Grid internally will rectify the put operation by sending it to the new primary owner for the given key automatically, however there are no guarantees in regards to if the write was first replicated to backups. Thus more than 1 of the following write events (**CacheEntryCreatedEvent**, **CacheEntryModifiedEvent** & **CacheEntryRemovedEvent**) may be sent on a single operation.

If more than one event is generated Data Grid will mark the event that it was generated by a retried command to help the user to know when this occurs without having to pay attention to view changes.

```
@Listener
public class MyRetryListener {
  @CacheEntryModified
  public void entryModified(CacheEntryModifiedEvent event) {
    if (event.isCommandRetried()) {
      // Do something
    }
  }
}
```

Also when using a **CacheEventFilter** or **CacheEventConverter** the EventType contains a method **isRetry** to tell if the event was generated due to retry.

3.3.2. Cache manager-level notifications

Cache manager-level events occur on a cache manager. These too are global and cluster-wide, but involve events that affect all caches created by a single cache manager. Examples of cache manager-level events are nodes joining or leaving a cluster, or caches starting or stopping.

See the org.infinispan.notifications.cachemanagerlistener.annotation package for a comprehensive list of all cache manager-level notifications, and their respective method-level annotations.

3.3.3. Synchronicity of events

By default, all async notifications are dispatched in the notification thread pool. Sync notifications will delay the operation from continuing until the listener method completes or the CompletionStage completes (the former causing the thread to block). Alternatively, you could annotate your listener as *asynchronous* in which case the operation will continue immediately, while the notification is completed asynchronously on the notification thread pool. To do this, simply annotate your listener such:

Asynchronous Listener

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

```
    @CacheEntryCreated
    void listen(CacheEntryCreatedEvent event) { }
  }
```

Blocking Synchronous Listener

```
  @Listener
  public class MySyncListener {
    @CacheEntryCreated
    void listen(CacheEntryCreatedEvent event) { }
  }
```

Non-Blocking Listener

```
  @Listener
  public class MyNonBlockingListener {
    @CacheEntryCreated
    CompletionStage<Void> listen(CacheEntryCreatedEvent event) { }
  }
```

3.3.3.1. Asynchronous thread pool

To tune the thread pool used to dispatch such asynchronous notifications, use the **<listener-executor />** XML element in your configuration file.

3.4. ASYNCHRONOUS API

In addition to synchronous API methods like Cache.put() , Cache.remove() , etc., Data Grid also has an asynchronous, non-blocking API where you can achieve the same results in a non-blocking fashion.

These methods are named in a similar fashion to their blocking counterparts, with "Async" appended. E.g., Cache.putAsync() , Cache.removeAsync() , etc. These asynchronous counterparts return a CompletableFuture that contains the actual result of the operation.

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

3.4.1. Why use such an API?

Non-blocking APIs are powerful in that they provide all of the guarantees of synchronous communications - with the ability to handle communication failures and exceptions - with the ease of not having to block until a call completes. This allows you to better harness parallelism in your system. For example:

```
  Set<CompletableFuture<?>> futures = new HashSet<>();
  futures.add(cache.putAsync(key1, value1)); // does not block
  futures.add(cache.putAsync(key2, value2)); // does not block
  futures.add(cache.putAsync(key3, value3)); // does not block

  // the remote calls for the 3 puts will effectively be executed
  // in parallel, particularly useful if running in distributed mode
  // and the 3 keys would typically be pushed to 3 different nodes
  // in the cluster
```

```
// check that the puts completed successfully
for (CompletableFuture<?> f: futures) f.get();
```

3.4.2. Which processes actually happen asynchronously?

There are 4 things in Data Grid that can be considered to be on the critical path of a typical write operation. These are, in order of cost:

- network calls

- marshalling

- writing to a cache store (optional)

- locking

Using the async methods will take the network calls and marshalling off the critical path. For various technical reasons, writing to a cache store and acquiring locks, however, still happens in the caller's thread.

CHAPTER 4. CLUSTERED LOCKS

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

4.1. LOCK API

Data Grid provides a **ClusteredLock** API that lets you concurrently execute code on a cluster when using Data Grid in embedded mode.

The API consists of the following:

- **ClusteredLock** exposes methods to implement clustered locks.

- **ClusteredLockManager** exposes methods to define, configure, retrieve, and remove clustered locks.

- **EmbeddedClusteredLockManagerFactory** initializes **ClusteredLockManager** implementations.

Ownership

Data Grid supports **NODE** ownership so that all nodes in a cluster can use a lock.

Reentrancy

Data Grid clustered locks are non-reentrant so any node in the cluster can acquire a lock but only the node that creates the lock can release it.

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.

Reference

- EmbeddedClusteredLockManagerFactory

- ClusteredLockManager

- ClusteredLock

4.2. USING CLUSTERED LOCKS

Learn how to use clustered locks with Data Grid embedded in your application.

Prerequisites

- Add the **infinispan-clustered-lock** dependency to your **pom.xml**:

```xml
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-lock</artifactId>
</dependency>
```

Procedure

1. Initialize the **ClusteredLockManager** interface from a Cache Manager. This interface is the entry point for defining, retrieving, and removing clustered locks.

2. Give a unique name for each clustered lock.

3. Acquire locks with the **lock.tryLock(1, TimeUnit.SECONDS)** method.

```
// Set up a clustered Cache Manager.
GlobalConfigurationBuilder global = GlobalConfigurationBuilder.defaultClusteredBuilder();

// Configure the cache mode, in this case it is 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.
ClusteredLockManager clm1 = EmbeddedClusteredLockManagerFactory.from(cm);

// Define a clustered lock named 'lock'.
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();
});
```

4.3. CONFIGURING INTERNAL CACHES FOR LOCKS

Clustered Lock Managers include an internal cache that stores lock state. You can configure the internal cache either declaratively or programmatically.

Procedure

1. Define the number of nodes in the cluster that store the state of clustered locks. The default value is **-1**, which replicates the value to all nodes.

2. Specify one of the following values for the cache reliability, which controls how clustered locks behave when clusters split into partitions or multiple nodes leave:

   - **AVAILABLE**: Nodes in any partition can concurrently operate on locks.

   - **CONSISTENT**: Only nodes that belong to the majority partition can operate on locks. This is the default value.

   - Programmatic configuration

     ```
     import org.infinispan.lock.configuration.ClusteredLockManagerConfiguration;
     import org.infinispan.lock.configuration.ClusteredLockManagerConfigurationBuilder;
     import org.infinispan.lock.configuration.Reliability;
     ...

     GlobalConfigurationBuilder global =
     GlobalConfigurationBuilder.defaultClusteredBuilder();

     final ClusteredLockManagerConfiguration config =
     global.addModule(ClusteredLockManagerConfigurationBuilder.class).numOwner(2).relia
     bility(Reliability.AVAILABLE).create();

     DefaultCacheManager cm = new DefaultCacheManager(global.build());

     ClusteredLockManager clm1 = EmbeddedClusteredLockManagerFactory.from(cm);

     clm1.defineLock("lock");
     ```

   - Declarative configuration

     ```
     <infinispan
     ```

```
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:13.0
https://infinispan.org/schemas/infinispan-config-13.0.xsd"
      xmlns="urn:infinispan:config:13.0">

  <cache-container default-cache="default">
    <transport/>
    <local-cache name="default">
       <locking concurrency-level="100" acquire-timeout="1000"/>
    </local-cache>
    <clustered-locks xmlns="urn:infinispan:config:clustered-locks:13.0"
              num-owners = "3"
              reliability="AVAILABLE">
       <clustered-lock name="lock1" />
       <clustered-lock name="lock2" />
    </clustered-locks>
  </cache-container>
  <!-- Cache configuration goes here. -->
</infinispan>
```

Reference

- ClusteredLockManagerConfiguration

- Clustered Locks Configuration Schema

CHAPTER 5. CLUSTERED COUNTERS

Clustered counters are counters which are distributed and shared among all nodes in the Data Grid cluster. Counters can have different consistency levels: strong and weak.

Although a strong/weak consistent counter has separate interfaces, both support updating its value, return the current value and they provide events when its value is updated. Details are provided below in this document to help you choose which one fits best your uses-case.

5.1. INSTALLATION AND CONFIGURATION

In order to start using the counters, you needs to add the dependency in your Maven **pom.xml** file:

pom.xml

```xml
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-counter</artifactId>
</dependency>
```

The counters can be configured Data Grid configuration file or on-demand via the **CounterManager** interface detailed later in this document. A counters configured in Data Grid configuration file is created at boot time when the **EmbeddedCacheManager** is starting. These counters are started eagerly and they are available in all the cluster's nodes.

configuration.xml

```xml
<infinispan>
   <cache-container ...>
      <!-- To persist counters, you need to configure the global state. -->
      <global-state>
      <!-- Global state configuration goes here. -->
      </global-state>
      <!-- Cache configuration goes here. -->
       <counters xmlns="urn:infinispan:config:counters:13.0" num-owners="3"
reliability="CONSISTENT">
            <strong-counter name="c1" initial-value="1" storage="PERSISTENT"/>
            <strong-counter name="c2" initial-value="2" storage="VOLATILE" lower-bound="0"/>
            <strong-counter name="c3" initial-value="3" storage="PERSISTENT" upper-bound="5"/>
            <strong-counter name="c4" initial-value="4" storage="VOLATILE" lower-bound="0" upper-
bound="10"/>
            <strong-counter name="c5" initial-value="0" upper-bound="100" lifespan="60000"/>
            <weak-counter name="c6" initial-value="5" storage="PERSISTENT" concurrency-level="1"/>
        </counters>
    </cache-container>
</infinispan>
```

or programmatically, in the **GlobalConfigurationBuilder**:

```java
GlobalConfigurationBuilder globalConfigurationBuilder = ...;
CounterManagerConfigurationBuilder builder =
globalConfigurationBuilder.addModule(CounterManagerConfigurationBuilder.class);
builder.numOwner(3).reliability(Reliability.CONSISTENT);
builder.addStrongCounter().name("c1").initialValue(1).storage(Storage.PERSISTENT);
```

```
builder.addStrongCounter().name("c2").initialValue(2).lowerBound(0).storage(Storage.VOLATILE);
builder.addStrongCounter().name("c3").initialValue(3).upperBound(5).storage(Storage.PERSISTENT)
;
builder.addStrongCounter().name("c4").initialValue(4).lowerBound(0).upperBound(10).storage(Storag
e.VOLATILE);
builder.addStrongCounter().name("c5").initialValue(0).upperBound(100).lifespan(60000);
builder.addWeakCounter().name("c6").initialValue(5).concurrencyLevel(1).storage(Storage.PERSIST
ENT);
```

On other hand, the counters can be configured on-demand, at any time after the
EmbeddedCacheManager is initialized.

```
CounterManager manager = ...;
manager.defineCounter("c1",
CounterConfiguration.builder(CounterType.UNBOUNDED_STRONG).initialValue(1).storage(Storage.
PERSISTENT).build());
manager.defineCounter("c2",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(2).lowerBound(0).stora
ge(Storage.VOLATILE).build());
manager.defineCounter("c3",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(3).upperBound(5).stor
age(Storage.PERSISTENT).build());
manager.defineCounter("c4",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(4).lowerBound(0).uppe
rBound(10).storage(Storage.VOLATILE).build());
manager.defineCounter("c4",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(0).upperBound(100).lif
espan(60000).build());
manager.defineCounter("c6",
CounterConfiguration.builder(CounterType.WEAK).initialValue(5).concurrencyLevel(1).storage(Storag
e.PERSISTENT).build());
```

> **NOTE**
>
> **CounterConfiguration** is immutable and can be reused.

The method **defineCounter()** will return **true** if the counter is successful configured or **false** otherwise.
However, if the configuration is invalid, the method will throw a **CounterConfigurationException**. To
find out if a counter is already defined, use the method **isDefined()**.

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

Additional resources

- [Data Grid configuration schema reference](#)

5.1.1. List counter names

To list all the counters defined, the method **CounterManager.getCounterNames()** returns a collection
of all counter names created cluster-wide.

5.2. COUNTERMANAGER INTERFACE

The **CounterManager** interface is the entry point to define, retrieve and remove counters.

Embedded deployments

CounterManager automatically listen to the creation of **EmbeddedCacheManager** and proceeds with the registration of an instance of it per **EmbeddedCacheManager**. It starts the caches needed to store the counter state and configures the default counters.

Retrieving the **CounterManager** is as simple as invoke the **EmbeddedCounterManagerFactory.asCounterManager(EmbeddedCacheManager)** as shown in the example below:

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

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

Server deployments

For Hot Rod clients, the **CounterManager** is registered in the RemoteCacheManager and can be retrieved as follows:

```
// create or obtain your RemoteCacheManager
RemoteCacheManager manager = ...;

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

5.2.1. Remove a counter via CounterManager

There is a difference between remove a counter via the **Strong/WeakCounter** interfaces and the **CounterManager**. The **CounterManager.remove(String)** removes the counter value from the cluster and removes all the listeners registered in the counter in the local counter instance. In addition, the counter instance is no longer reusable and it may return an invalid results.

On the other side, the **Strong/WeakCounter** removal only removes the counter value. The instance can still be reused and the listeners still works.

> **NOTE**
>
> The counter is re-created if it is accessed after a removal.

5.3. THE COUNTER

A counter can be strong (**StrongCounter**) or weakly consistent (**WeakCounter**) and both is identified by a name. They have a specific interface but they share some logic, namely, both of them are asynchronous (a **CompletableFuture** is returned by each operation), provide an update event and can be reset to its initial value.

If you don't want to use the async API, it is possible to return a synchronous counter via **sync()** method. The API is the same but without the **CompletableFuture** return value.

The following methods are common to both interfaces:

```
String getName();
CompletableFuture<Long> getValue();
CompletableFuture<Void> reset();
<T extends CounterListener> Handle<T> addListener(T listener);
CounterConfiguration getConfiguration();
CompletableFuture<Void> remove();
SyncStrongCounter sync(); //SyncWeakCounter for WeakCounter
```

- **getName()** returns the counter name (identifier).

- **getValue()** returns the current counter's value.

- **reset()** allows to reset the counter's value to its initial value.

- **addListener()** register a listener to receive update events. More details about it in the Notification and Events section.

- **getConfiguration()** returns the configuration used by the counter.

- **remove()** removes the counter value from the cluster. The instance can still be used and the listeners are kept.

- **sync()** creates a synchronous counter.

> **NOTE**
>
> The counter is re-created if it is accessed after a removal.

5.3.1. The `StrongCounter` interface: when the consistency or bounds matters.

The strong counter provides uses a single key stored in Data Grid cache to provide the consistency needed. All the updates are performed under the key lock to updates its values. On other hand, the reads don't acquire any locks and reads the current value. Also, with this scheme, it allows to bound the counter value and provide atomic operations like compare-and-set/swap.

A **StrongCounter** can be retrieved from the **CounterManager** by using the **getStrongCounter()** method. As an example:

```
CounterManager counterManager = ...
StrongCounter aCounter = counterManager.getStrongCounter("my-counter");
```

> **WARNING**
>
> Since every operation will hit a single key, the **StrongCounter** has a higher contention rate.

The **StrongCounter** interface adds the following method:

```
default CompletableFuture<Long> incrementAndGet() {
   return addAndGet(1L);
}

default CompletableFuture<Long> decrementAndGet() {
   return addAndGet(-1L);
}

CompletableFuture<Long> addAndGet(long delta);

CompletableFuture<Boolean> compareAndSet(long expect, long update);

CompletableFuture<Long> compareAndSwap(long expect, long update);
```

- **incrementAndGet()** increments the counter by one and returns the new value.

- **decrementAndGet()** decrements the counter by one and returns the new value.

- **addAndGet()** adds a delta to the counter's value and returns the new value.

- **compareAndSet()** and **compareAndSwap()** atomically set the counter's value if the current value is the expected.
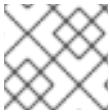
> **NOTE**
>
> A operation is considered completed when the **CompletableFuture** is completed.

> **NOTE**
>
> The difference between compare-and-set and compare-and-swap is that the former returns true if the operation succeeds while the later returns the previous value. The compare-and-swap is successful if the return value is the same as the expected.

5.3.1.1. Bounded **StrongCounter**

When bounded, all the update method above will throw a **CounterOutOfBoundsException** when they reached the lower or upper bound. The exception has the following methods to check which side bound has been reached:

```
public boolean isUpperBoundReached();
public boolean isLowerBoundReached();
```

5.3.1.2. Uses cases

The strong counter fits better in the following uses cases:

- When counter's value is needed after each update (example, cluster-wise ids generator or sequences)

- When a bounded counter is needed (example, rate limiter)

5.3.1.3. Usage Examples

```java
StrongCounter counter = counterManager.getStrongCounter("unbounded_counter");

// incrementing the counter
System.out.println("new value is " + counter.incrementAndGet().get());

// decrement the counter's value by 100 using the functional API
counter.addAndGet(-100).thenApply(v -> {
  System.out.println("new value is " + v);
  return null;
}).get();

// alternative, you can do some work while the counter is updated
CompletableFuture<Long> f = counter.addAndGet(10);
// ... do some work ...
System.out.println("new value is " + f.get());

// and then, check the current value
System.out.println("current value is " + counter.getValue().get());

// finally, reset to initial value
counter.reset().get();
System.out.println("current value is " + counter.getValue().get());

// or set to a new value if zero
System.out.println("compare and set succeeded? " + counter.compareAndSet(0, 1));
```

And below, there is another example using a bounded counter:

```java
StrongCounter counter = counterManager.getStrongCounter("bounded_counter");

// incrementing the counter
try {
  System.out.println("new value is " + counter.addAndGet(100).get());
} catch (ExecutionException e) {
  Throwable cause = e.getCause();
  if (cause instanceof CounterOutOfBoundsException) {
    if (((CounterOutOfBoundsException) cause).isUpperBoundReached()) {
      System.out.println("ops, upper bound reached.");
    } else if (((CounterOutOfBoundsException) cause).isLowerBoundReached()) {
      System.out.println("ops, lower bound reached.");
    }
  }
}

// now using the functional API
counter.addAndGet(-100).handle((v, throwable) -> {
  if (throwable != null) {
    Throwable cause = throwable.getCause();
    if (cause instanceof CounterOutOfBoundsException) {
      if (((CounterOutOfBoundsException) cause).isUpperBoundReached()) {
        System.out.println("ops, upper bound reached.");
      } else if (((CounterOutOfBoundsException) cause).isLowerBoundReached()) {
        System.out.println("ops, lower bound reached.");
      }
    }
```

```
      return null;
   }
   System.out.println("new value is " + v);
   return null;
}).get();
```

Compare-and-set vs Compare-and-swap examples:

```
StrongCounter counter = counterManager.getStrongCounter("my-counter");
long oldValue, newValue;
do {
   oldValue = counter.getValue().get();
   newValue = someLogic(oldValue);
} while (!counter.compareAndSet(oldValue, newValue).get());
```

With compare-and-swap, it saves one invocation counter invocation (**counter.getValue()**)

```
StrongCounter counter = counterManager.getStrongCounter("my-counter");
long oldValue = counter.getValue().get();
long currentValue, newValue;
do {
   currentValue = oldValue;
   newValue = someLogic(oldValue);
} while ((oldValue = counter.compareAndSwap(oldValue, newValue).get()) != currentValue);
```

To use a strong counter as a rate limiter, configure **upper-bound** and **lifespan** parameters as follows:

```
// 5 request per minute
CounterConfiguration configuration =
CounterConfiguration.builder(CounterType.BOUNDED_STRONG)
      .upperBound(5)
      .lifespan(60000)
      .build();
counterManager.defineCounter("rate_limiter", configuration);
StrongCounter counter = counterManager.getStrongCounter("rate_limiter");

// on each operation, invoke
try {
   counter.incrementAndGet().get();
   // continue with operation
} catch (InterruptedException e) {
   Thread.currentThread().interrupt();
} catch (ExecutionException e) {
   if (e.getCause() instanceof CounterOutOfBoundsException) {
      // maximum rate. discard operation
      return;
   } else {
      // unexpected error, handling property
   }
}
```

NOTE

The **lifespan** parameter is an experimental capability and may be removed in a future version.

5.3.2. The `WeakCounter` interface: when speed is needed

The **WeakCounter** stores the counter's value in multiple keys in Data Grid cache. The number of keys created is configured by the **concurrency-level** attribute. Each key stores a partial state of the counter's value and it can be updated concurrently. It main advantage over the **StrongCounter** is the lower contention in the cache. On other hand, the read of its value is more expensive and bounds are not allowed.

WARNING

The reset operation should be handled with caution. It is **not** atomic and it produces intermediates values. These value may be seen by a read operation and by any listener registered.

A **WeakCounter** can be retrieved from the **CounterManager** by using the **getWeakCounter()** method. As an example:

```
CounterManager counterManager = ...
StrongCounter aCounter = counterManager.getWeakCounter("my-counter);
```

5.3.2.1. Weak Counter Interface

The **WeakCounter** adds the following methods:

```
default CompletableFuture<Void> increment() {
   return add(1L);
}

default CompletableFuture<Void> decrement() {
   return add(-1L);
}

CompletableFuture<Void> add(long delta);
```

They are similar to the `StrongCounter's methods but they don't return the new value.

5.3.2.2. Uses cases

The weak counter fits best in uses cases where the result of the update operation is not needed or the counter's value is not required too often. Collecting statistics is a good example of such an use case.

5.3.2.3. Examples

Below, there is an example of the weak counter usage.

```
WeakCounter counter = counterManager.getWeakCounter("my_counter");

// increment the counter and check its result
counter.increment().get();
System.out.println("current value is " + counter.getValue());

CompletableFuture<Void> f = counter.add(-100);
//do some work
f.get(); //wait until finished
System.out.println("current value is " + counter.getValue().get());

//using the functional API
counter.reset().whenComplete((aVoid, throwable) -> System.out.println("Reset done " + (throwable
== null ? "successfully" : "unsuccessfully"))).get();
System.out.println("current value is " + counter.getValue().get());
```

5.4. NOTIFICATIONS AND EVENTS

Both strong and weak counter supports a listener to receive its updates events. The listener must implement **CounterListener** and it can be registered by the following method:

```
<T extends CounterListener> Handle<T> addListener(T listener);
```

The **CounterListener** has the following interface:

```
public interface CounterListener {
   void onUpdate(CounterEvent entry);
}
```

The **Handle** object returned has the main goal to remove the **CounterListener** when it is not longer needed. Also, it allows to have access to the **CounterListener** instance that is it handling. It has the following interface:

```
public interface Handle<T extends CounterListener> {
   T getCounterListener();
   void remove();
}
```

Finally, the **CounterEvent** has the previous and current value and state. It has the following interface:

```
public interface CounterEvent {
   long getOldValue();
   State getOldState();
   long getNewValue();
   State getNewState();
}
```

> **NOTE**
>
> The state is always **State.VALID** for unbounded strong counter and weak counter. **State.LOWER_BOUND_REACHED** and **State.UPPER_BOUND_REACHED** are only valid for bounded strong counters.

> **WARNING**
>
> The weak counter **reset()** operation will trigger multiple notification with intermediate values.

CHAPTER 6. USING THE CDI EXTENSION

Data Grid provides an extension that integrates with the CDI (Contexts and Dependency Injection) programming model and allows you to:

- Configure and inject caches into CDI Beans and Java EE components.

- Configure cache managers.

- Receive cache and cache manager level events.

- Control data storage and retrieval using JCache annotations.

6.1. CDI DEPENDENCIES

Update your **pom.xml** with one of the following dependencies to include the Data Grid CDI extension in your project:

Embedded (Library) Mode

```xml
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cdi-embedded</artifactId>
</dependency>
```

Server Mode

```xml
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cdi-remote</artifactId>
</dependency>
```

6.2. INJECTING EMBEDDED CACHES

Set up CDI beans to inject embedded caches.

Procedure

1. Create a cache qualifier annotation.

   ```java
   ...
   import javax.inject.Qualifier;

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

   ❶   Creates a **@GreetingCache** qualifier.

2. Add a producer method that defines the cache configuration.

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

public class Config {

    @ConfigureCache("mygreetingcache") 1
    @GreetingCache 2
    @Produces
    public Configuration greetingCacheConfiguration() {
        return new ConfigurationBuilder()
                .memory()
                    .size(1000)
                .build();
    }
}
```

1 Names the cache to inject.

2 Adds the cache qualifier.

3. Add a producer method that creates a clustered cache manager, if required

```
...
package org.infinispan.configuration.global.GlobalConfigurationBuilder;

public class Config {

    @GreetingCache 1
    @Produces
    @ApplicationScoped 2
    public EmbeddedCacheManager defaultClusteredCacheManager() { 3
        return new DefaultCacheManager(
            new GlobalConfigurationBuilder().transport().defaultTransport().build();
    }
}
```

1 Adds the cache qualifier.

2 Creates the bean once for the application. Producers that create cache managers should always include the **@ApplicationScoped** annotation to avoid creating multiple cache managers.

3 Creates a new **DefaultCacheManager** instance that is bound to the **@GreetingCache** qualifier.
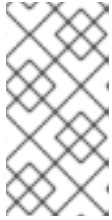
NOTE

Cache managers are heavy weight objects. Having more than one cache manager running in your application can degrade performance. When injecting multiple caches, either add the qualifier of each cache to the cache manager producer method or do not add any qualifier.

4. Add the **@GreetingCache** qualifier to your cache injection point.

```
...
import javax.inject.Inject;

public class GreetingService {

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

    public String greet(String user) {
        String cachedValue = cache.get(user);
        if (cachedValue == null) {
            cachedValue = "Hello " + user;
            cache.put(user, cachedValue);
        }
        return cachedValue;
    }
}
```

6.3. INJECTING REMOTE CACHES

Set up CDI beans to inject remote caches.

Procedure

1. Create a cache qualifier annotation.

```
@Remote("mygreetingcache") 1
@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RemoteGreetingCache { 2
}
```

1   names the cache to inject.

2   creates a **@RemoteGreetingCache** qualifier.

2. Add the **@RemoteGreetingCache** qualifier to your cache injection point.

```
public class GreetingService {

    @Inject @RemoteGreetingCache
    private RemoteCache<String, String> cache;
```

```java
    public String greet(String user) {
        String cachedValue = cache.get(user);
        if (cachedValue == null) {
            cachedValue = "Hello " + user;
            cache.put(user, cachedValue);
        }
        return cachedValue;
    }
}
```

Tips for injecting remote caches

- You can inject remote caches without using qualifiers.

  ```java
  ...
  @Inject
  @Remote("greetingCache")
  private RemoteCache<String, String> cache;
  ```

- If you have more than one Data Grid cluster, you can create separate remote cache manager producers for each cluster.

  ```java
  ...
  import javax.enterprise.context.ApplicationScoped;

  public class Config {

      @RemoteGreetingCache
      @Produces
      @ApplicationScoped ❶
      public ConfigurationBuilder builder = new ConfigurationBuilder(); ❷
          builder.addServer().host("localhost").port(11222);
          return new RemoteCacheManager(builder.build());
      }
  }
  ```

❶ creates the bean once for the application. Producers that create cache managers should always include the **@ApplicationScoped** annotation to avoid creating multiple cache managers, which are heavy weight objects.

❷ creates a new **RemoteCacheManager** instance that is bound to the **@RemoteGreetingCache** qualifier.

6.4. JCACHE CACHING ANNOTATIONS

You can use the following JCache caching annotations with CDI managed beans when JCache artifacts are on the classpath:

@CacheResult
    caches the results of method calls.

@CachePut

caches method parameters.

@CacheRemoveEntry

removes entries from a cache.

@CacheRemoveAll

removes all entries from a cache.
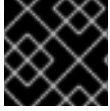
> **IMPORTANT**
>
> **Target type:** You can use these JCache caching annotations on methods only.

To use JCache caching annotations, declare interceptors in the **beans.xml** file for your application.

Managed Environments (Application Server)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
   version="1.2" bean-discovery-mode="annotated">

  <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>
```

Non-managed Environments (Standalone)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
   version="1.2" bean-discovery-mode="annotated">

  <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>
```

JCache Caching Annotation Examples

The following example shows how the **@CacheResult** annotation caches the results of the **GreetingService.greet()** method:

```
import javax.cache.interceptor.CacheResult;
```

```
public class GreetingService {

  @CacheResult
  public String greet(String user) {
    return "Hello" + user;
  }
}
```

With JCache annotations, the default cache uses the fully qualified name of the annotated method with its parameter types, for example:
org.infinispan.example.GreetingService.greet(java.lang.String)

To use caches other than the default, use the **cacheName** attribute to specify the cache name as in the following example:

```
@CacheResult(cacheName = "greeting-cache")
```

6.5. RECEIVING CACHE AND CACHE MANAGER EVENTS

You can use CDI Events to receive Cache and cache manager level events.

- Use the **@Observes** annotation as in the following example:

```
import javax.enterprise.event.Observes;
import org.infinispan.notifications.cachemanagerlistener.event.CacheStartedEvent;
import org.infinispan.notifications.cachelistener.event.*;

public class GreetingService {

  // Cache level events
  private void entryRemovedFromCache(@Observes CacheEntryCreatedEvent event) {
    ...
  }

  // Cache manager level events
  private void cacheStarted(@Observes CacheStartedEvent event) {
    ...
  }
}
```

CHAPTER 7. LOCKING AND CONCURRENCY

Data Grid makes use of multi-versioned concurrency control (MVCC) - a concurrency scheme popular with relational databases and other data stores. MVCC offers many advantages over coarse-grained Java synchronization and even JDK Locks for access to shared data, including:

- allowing concurrent readers and writers

- readers and writers do not block one another

- write skews can be detected and handled

- internal locks can be striped

7.1. LOCKING IMPLEMENTATION DETAILS

Data Grid's MVCC implementation makes use of minimal locks and synchronizations, leaning heavily towards lock-free techniques such as compare-and-swap and lock-free data structures wherever possible, which helps optimize for multi-CPU and multi-core environments.

In particular, Data Grid's MVCC implementation is heavily optimized for readers. Reader threads do not acquire explicit locks for entries, and instead directly read the entry in question.

Writers, on the other hand, need to acquire a write lock. This ensures only one concurrent writer per entry, causing concurrent writers to queue up to change an entry.

To allow concurrent reads, writers make a copy of the entry they intend to modify, by wrapping the entry in an **MVCCEntry**. This copy isolates concurrent readers from seeing partially modified state. Once a write has completed, **MVCCEntry.commit()** will flush changes to the data container and subsequent readers will see the changes written.

7.1.1. Clustered caches and locks

In Data Grid clusters, primary owner nodes are responsible for locking keys.

For non-transactional caches, Data Grid forwards the write operation to the primary owner of the key so it can attempt to lock it. Data Grid either then forwards the write operation to the other owners or throws an exception if it cannot lock the key.

> **NOTE**
>
> If the operation is conditional and fails on the primary owner, Data Grid does not forward it to the other owners.

For transactional caches, primary owners can lock keys with optimistic and pessimistic locking modes. Data Grid also supports different isolation levels to control concurrent reads between transactions.

7.1.2. The LockManager

The **LockManager** is a component that is responsible for locking an entry for writing. The **LockManager** makes use of a **LockContainer** to locate/hold/create locks. **LockContainers** come in two broad flavours, with support for lock striping and with support for one lock per entry.

7.1.3. Lock striping

Lock striping entails the use of a fixed-size, shared collection of locks for the entire cache, with locks being allocated to entries based on the entry's key's hash code. Similar to the way the JDK's **ConcurrentHashMap** allocates locks, this allows for a highly scalable, fixed-overhead locking mechanism in exchange for potentially unrelated entries being blocked by the same lock.

The alternative is to disable lock striping - which would mean a *new* lock is created per entry. This approach *may* give you greater concurrent throughput, but it will be at the cost of additional memory usage, garbage collection churn, etc.

DEFAULT LOCK STRIPING SETTINGS

lock striping is disabled by default, due to potential deadlocks that can happen if locks for different keys end up in the same lock stripe.

The size of the shared lock collection used by lock striping can be tuned using the **concurrencyLevel** attribute of the **<locking />** configuration element.

Configuration example:

```
<locking striping="false|true"/>
```

Or

```
new ConfigurationBuilder().locking().useLockStriping(false|true);
```

7.1.4. Concurrency levels

In addition to determining the size of the striped lock container, this concurrency level is also used to tune any JDK **ConcurrentHashMap** based collections where related, such as internal to **DataContainer**s. Please refer to the JDK **ConcurrentHashMap** Javadocs for a detailed discussion of concurrency levels, as this parameter is used in exactly the same way in Data Grid.

Configuration example:

```
<locking concurrency-level="32"/>
```

Or

```
new ConfigurationBuilder().locking().concurrencyLevel(32);
```

7.1.5. Lock timeout

The lock timeout specifies the amount of time, in milliseconds, to wait for a contented lock.

Configuration example:

```
<locking acquire-timeout="10000"/>
```

Or

```
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10000);
//alternatively
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10, TimeUnit.SECONDS);
```

7.1.6. Consistency

The fact that a single owner is locked (as opposed to all owners being locked) does not break the following consistency guarantee: if key **K** is hashed to nodes **{A, B}** and transaction **TX1** acquires a lock for **K**, let's say on **A**. If another transaction, **TX2**, is started on **B** (or any other node) and **TX2** tries to lock **K** then it will fail with a timeout as the lock is already held by **TX1**. The reason for this is the that the lock for a key **K** is always, deterministically, acquired on the same node of the cluster, regardless of where the transaction originates.

7.2. DATA VERSIONING

Data Grid supports two forms of data versioning: simple and external. The simple versioning is used in transactional caches for write skew check.

The external versioning is used to encapsulate an external source of data versioning within Data Grid, such as when using Data Grid with Hibernate which in turn gets its data version information directly from a database.

In this scheme, a mechanism to pass in the version becomes necessary, and overloaded versions of **put()** and **putForExternalRead()** will be provided in **AdvancedCache** to take in an external data version. This is then stored on the **InvocationContext** and applied to the entry at commit time.
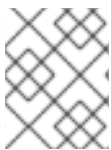
> **NOTE**
>
> Write skew checks cannot and will not be performed in the case of external data versioning.

CHAPTER 8. TRANSACTIONS

Data Grid can be configured to use and to participate in JTA compliant transactions.

Alternatively, if transaction support is disabled, it is equivalent to using autocommit in JDBC calls, where modifications are potentially replicated after every change (if replication is enabled).

On every cache operation Data Grid does the following:

1. Retrieves the current Transaction associated with the thread

2. If not already done, registers XAResource with the transaction manager to be notified when a transaction commits or is rolled back.

In order to do this, the cache has to be provided with a reference to the environment's TransactionManager. This is usually done by configuring the cache with the class name of an implementation of the TransactionManagerLookup interface. When the cache starts, it will create an instance of this class and invoke its **getTransactionManager()** method, which returns a reference to the **TransactionManager**.

Data Grid ships with several transaction manager lookup classes:

Transaction manager lookup implementations

- EmbeddedTransactionManagerLookup: This provides with a basic transaction manager which should only be used for embedded mode when no other implementation is available. This implementation has some severe limitations to do with concurrent transactions and recovery.

- JBossStandaloneJTAManagerLookup: If you're running Data Grid in a standalone environment, or in JBoss AS 7 and earlier, and WildFly 8, 9, and 10, this should be your default choice for transaction manager. It's a fully fledged transaction manager based on JBoss Transactions which overcomes all the deficiencies of the **EmbeddedTransactionManager**.

- WildflyTransactionManagerLookup: If you're running Data Grid in WildFly 11 or later, this should be your default choice for transaction manager.

- GenericTransactionManagerLookup: This is a lookup class that locate transaction managers in the most popular Java EE application servers. If no transaction manager can be found, it defaults on the **EmbeddedTransactionManager**.

WARN: **DummyTransactionManagerLookup** has been deprecated in 9.0 and it will be removed in the future. Use **EmbeddedTransactionManagerLookup** instead.

Once initialized, the **TransactionManager** can also be obtained from the **Cache** itself:

```
//the cache must have a transactionManagerLookupClass defined
Cache cache = cacheManager.getCache();

//equivalent with calling TransactionManagerLookup.getTransactionManager();
TransactionManager tm = cache.getAdvancedCache().getTransactionManager();
```

8.1. CONFIGURING TRANSACTIONS

Transactions are configured at cache level. Below is the configuration that affects a transaction behaviour and a small description of each configuration attribute.

```
<locking
  isolation="READ_COMMITTED"/>
<transaction
  locking="OPTIMISTIC"
  auto-commit="true"
  complete-timeout="60000"
  mode="NONE"
  notifications="true"
  reaper-interval="30000"
  recovery-cache="__recoveryInfoCacheName__"
  stop-timeout="30000"
  transaction-manager-
lookup="org.infinispan.transaction.lookup.GenericTransactionManagerLookup"/>
```

or programmatically:

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.locking()
    .isolationLevel(IsolationLevel.READ_COMMITTED);
builder.transaction()
    .lockingMode(LockingMode.OPTIMISTIC)
    .autoCommit(true)
    .completedTxTimeout(60000)
    .transactionMode(TransactionMode.NON_TRANSACTIONAL)
    .useSynchronization(false)
    .notifications(true)
    .reaperWakeUpInterval(30000)
    .cacheStopTimeout(30000)
    .transactionManagerLookup(new GenericTransactionManagerLookup())
    .recovery()
    .enabled(false)
    .recoveryInfoCacheName("__recoveryInfoCacheName__");
```

- **isolation** – configures the isolation level. Check section Isolation Levels for more details. Default is **REPEATABLE_READ**.

- **locking** – configures whether the cache uses optimistic or pessimistic locking. Check section Transaction Locking for more details. Default is **OPTIMISTIC**.

- **auto-commit** – if enable, the user does not need to start a transaction manually for a single operation. The transaction is automatically started and committed. Default is **true**.

- **complete-timeout** – the duration in milliseconds to keep information about completed transactions. Default is **60000**.

- **mode** – configures whether the cache is transactional or not. Default is **NONE**. The available options are:

  - **NONE** – non transactional cache

  - **FULL_XA** – XA transactional cache with recovery enabled. Check section Transaction recovery for more details about recovery.

  - **NON_DURABLE_XA** – XA transactional cache with recovery disabled.

- **NON_XA** - transactional cache with integration via Synchronization instead of XA. Check section Enlisting Synchronizations for details.

- **BATCH**- transactional cache using batch to group operations. Check section Batching for details.

- **notifications** - enables/disables triggering transactional events in cache listeners. Default is **true**.

- **reaper-interval** - the time interval in millisecond at which the thread that cleans up transaction completion information kicks in. Defaults is **30000**.

- **recovery-cache** - configures the cache name to store the recovery information. Check section Transaction recovery for more details about recovery. Default is ***recoveryInfoCacheName***.

- **stop-timeout** - the time in millisecond to wait for ongoing transaction when the cache is stopping. Default is **30000**.

- **transaction-manager-lookup** - configures the fully qualified class name of a class that looks up a reference to a **javax.transaction.TransactionManager**. Default is **org.infinispan.transaction.lookup.GenericTransactionManagerLookup**.

For more details on how Two-Phase-Commit (2PC) is implemented in Data Grid and how locks are being acquired see the section below. More details about the configuration settings are available in Configuration reference.

8.2. ISOLATION LEVELS

Data Grid offers two isolation levels - READ_COMMITTED and REPEATABLE_READ.

These isolation levels determine when readers see a concurrent write, and are internally implemented using different subclasses of **MVCCEntry**, which have different behaviour in how state is committed back to the data container.

Here's a more detailed example that should help understand the difference between **READ_COMMITTED** and **REPEATABLE_READ** in the context of Data Grid. With **READ_COMMITTED**, if between two consecutive read calls on the same key, the key has been updated by another transaction, the second read may return the new updated value:

```
Thread1: tx1.begin()
Thread1: cache.get(k) // returns v
Thread2:                         tx2.begin()
Thread2:                         cache.get(k) // returns v
Thread2:                         cache.put(k, v2)
Thread2:                         tx2.commit()
Thread1: cache.get(k) // returns v2!
Thread1: tx1.commit()
```

With **REPEATABLE_READ**, the final get will still return **v**. So, if you're going to retrieve the same key multiple times within a transaction, you should use **REPEATABLE_READ**.

However, as read-locks are not acquired even for **REPEATABLE_READ**, this phenomena can occur:

```
cache.get("A") // returns 1
cache.get("B") // returns 1
```

```
Thread1: tx1.begin()
Thread1: cache.put("A", 2)
Thread1: cache.put("B", 2)
Thread2:                      tx2.begin()
Thread2:                      cache.get("A") // returns 1
Thread1: tx1.commit()
Thread2:                      cache.get("B") // returns 2
Thread2:                      tx2.commit()
```

8.3. TRANSACTION LOCKING

8.3.1. Pessimistic transactional cache

From a lock acquisition perspective, pessimistic transactions obtain locks on keys at the time the key is written.

1. A lock request is sent to the primary owner (can be an explicit lock request or an operation)

2. The primary owner tries to acquire the lock:

   a. If it succeed, it sends back a positive reply;

   b. Otherwise, a negative reply is sent and the transaction is rollback.

As an example:

```
transactionManager.begin();
cache.put(k1,v1); //k1 is locked.
cache.remove(k2); //k2 is locked when this returns
transactionManager.commit();
```

When **cache.put(k1,v1)** returns, **k1** is locked and no other transaction running anywhere in the cluster can write to it. Reading **k1** is still possible. The lock on **k1** is released when the transaction completes (commits or rollbacks).

> **NOTE**
>
> For conditional operations, the validation is performed in the originator.

8.3.2. Optimistic transactional cache

With optimistic transactions locks are being acquired at transaction prepare time and are only being held up to the point the transaction commits (or rollbacks). This is different from the 5.0 default locking model where local locks are being acquire on writes and cluster locks are being acquired during prepare time.

1. The prepare is sent to all the owners.

2. The primary owners try to acquire the locks needed:

   a. If locking succeeds, it performs the write skew check.

   b. If the write skew check succeeds (or is disabled), send a positive reply.

   c. Otherwise, a negative reply is sent and the transaction is rolled back.

As an example:

```
transactionManager.begin();
cache.put(k1,v1);
cache.remove(k2);
transactionManager.commit(); //at prepare time, K1 and K2 is locked until committed/rolled back.
```
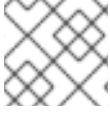
> **NOTE**
>
> For conditional commands, the validation still happens on the originator.

8.3.3. What do I need - pessimistic or optimistic transactions?

From a use case perspective, optimistic transactions should be used when there is *not* a lot of contention between multiple transactions running at the same time. That is because the optimistic transactions rollback if data has changed between the time it was read and the time it was committed (with write skew check enabled).

On the other hand, pessimistic transactions might be a better fit when there is high contention on the keys and transaction rollbacks are less desirable. Pessimistic transactions are more costly by their nature: each write operation potentially involves a RPC for lock acquisition.

8.4. WRITE SKEWS

Write skews occur when two transactions independently and simultaneously read and write to the same key. The result of a write skew is that both transactions successfully commit updates to the same key but with different values.

Data Grid automatically performs write skew checks to ensure data consistency for **REPEATABLE_READ** isolation levels in optimistic transactions. This allows Data Grid to detect and roll back one of the transactions.

When operating in **LOCAL** mode, write skew checks rely on Java object references to compare differences, which provides a reliable technique for checking for write skews.

8.4.1. Forcing write locks on keys in pessimitic transactions

To avoid write skews with pessimistic transactions, lock keys at read-time with **Flag.FORCE_WRITE_LOCK**.

> **NOTE**
>
> - In non-transactional caches, **Flag.FORCE_WRITE_LOCK** does not work. The **get()** call reads the key value but does not acquire locks remotely.
>
> - You should use **Flag.FORCE_WRITE_LOCK** with transactions in which the entity is updated later in the same transaction.

Compare the following code snippets for an example of **Flag.FORCE_WRITE_LOCK**:

```
// begin the transaction
if (!cache.getAdvancedCache().lock(key)) {
   // abort the transaction because the key was not locked
```

```
  } else {
    cache.get(key);
    cache.put(key, value);
    // commit the transaction
  }
```

```
  // begin the transaction
  try {
    // throws an exception if the key is not locked.
    cache.getAdvancedCache().withFlags(Flag.FORCE_WRITE_LOCK).get(key);
    cache.put(key, value);
  } catch (CacheException e) {
    // mark the transaction rollback-only
  }
  // commit or rollback the transaction
```

8.5. DEALING WITH EXCEPTIONS

If a CacheException (or a subclass of it) is thrown by a cache method within the scope of a JTA transaction, then the transaction is automatically marked for rollback.

8.6. ENLISTING SYNCHRONIZATIONS

By default Data Grid registers itself as a first class participant in distributed transactions through XAResource. There are situations where Data Grid is not required to be a participant in the transaction, but only to be notified by its lifecycle (prepare, complete): e.g. in the case Data Grid is used as a 2nd level cache in Hibernate.

Data Grid allows transaction enlistment through Synchronization. To enable it just use **NON_XA** transaction mode.

Synchronizations have the advantage that they allow **TransactionManager** to optimize 2PC with a 1PC where only one other resource is enlisted with that transaction (last resource commit optimization). E.g. Hibernate second level cache: if Data Grid registers itself with the **TransactionManager** as a **XAResource** than at commit time, the **TransactionManager** sees two **XAResource** (cache and database) and does not make this optimization. Having to coordinate between two resources it needs to write the tx log to disk. On the other hand, registering Data Grid as a **Synchronization** makes the **TransactionManager** skip writing the log to the disk (performance improvement).

8.7. BATCHING

Batching allows atomicity and some characteristics of a transaction, but not full-blown JTA or XA capabilities. Batching is often a lot lighter and cheaper than a full-blown transaction.

TIP

Generally speaking, one should use batching API whenever the only participant in the transaction is an Data Grid cluster. On the other hand, JTA transactions (involving **TransactionManager**) should be used whenever the transactions involves multiple systems. E.g. considering the "Hello world!" of transactions: transferring money from one bank account to the other. If both accounts are stored within Data Grid, then batching can be used. If one account is in a database and the other is Data Grid, then distributed transactions are required.
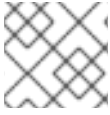
> **NOTE**
>
> You *do not* have to have a transaction manager defined to use batching.

8.7.1. API

Once you have configured your cache to use batching, you use it by calling **startBatch()** and **endBatch()** on **Cache**. E.g.,

```
Cache cache = cacheManager.getCache();
// not using a batch
cache.put("key", "value"); // will replicate immediately

// using a batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k2", "value");
cache.endBatch(true); // This will now replicate the modifications since the batch was started.

// a new batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(false); // This will "discard" changes made in the batch
```

8.7.2. Batching and JTA

Behind the scenes, the batching functionality starts a JTA transaction, and all the invocations in that scope are associated with it. For this it uses a very simple (e.g. no recovery) internal **TransactionManager** implementation. With batching, you get:

1. Locks you acquire during an invocation are held until the batch completes

2. Changes are all replicated around the cluster in a batch as part of the batch completion process. Reduces replication chatter for each update in the batch.

3. If synchronous replication or invalidation are used, a failure in replication/invalidation will cause the batch to roll back.

4. All the transaction related configurations apply for batching as well.

8.8. TRANSACTION RECOVERY

Recovery is a feature of XA transactions, which deal with the eventuality of a resource or possibly even the transaction manager failing, and recovering accordingly from such a situation.

8.8.1. When to use recovery

Consider a distributed transaction in which money is transferred from an account stored in an external database to an account stored in Data Grid. When **TransactionManager.commit()** is invoked, both resources prepare successfully (1st phase). During the commit (2nd) phase, the database successfully applies the changes whilst Data Grid fails before receiving the commit request from the transaction

manager. At this point the system is in an inconsistent state: money is taken from the account in the external database but not visible yet in Data Grid (since locks are only released during 2nd phase of a two-phase commit protocol). Recovery deals with this situation to make sure data in both the database and Data Grid ends up in a consistent state.

8.8.2. How does it work

Recovery is coordinated by the transaction manager. The transaction manager works with Data Grid to determine the list of in-doubt transactions that require manual intervention and informs the system administrator (via email, log alerts, etc). This process is transaction manager specific, but generally requires some configuration on the transaction manager.

Knowing the in-doubt transaction ids, the system administrator can now connect to the Data Grid cluster and replay the commit of transactions or force the rollback. Data Grid provides JMX tooling for this - this is explained extensively in the Transaction recovery and reconciliation section.

8.8.3. Configuring recovery

Recovery is *not* enabled by default in Data Grid. If disabled, the **TransactionManager** won't be able to work with Data Grid to determine the in-doubt transactions. The Transaction configuration section shows how to enable it.

NOTE: **recovery-cache** attribute is not mandatory and it is configured per-cache.

> **NOTE**
>
> For recovery to work, **mode** must be set to **FULL_XA**, since full-blown XA transactions are needed.

8.8.3.1. Enable JMX support

In order to be able to use JMX for managing recovery JMX support must be explicitly enabled.

8.8.4. Recovery cache

In order to track in-doubt transactions and be able to reply them, Data Grid caches all transaction state for future use. This state is held only for in-doubt transaction, being removed for successfully completed transactions after when the commit/rollback phase completed.

This in-doubt transaction data is held within a local cache: this allows one to configure swapping this info to disk through cache loader in the case it gets too big. This cache can be specified through the **recovery-cache** configuration attribute. If not specified Data Grid will configure a local cache for you.

It is possible (though not mandated) to share same recovery cache between all the Data Grid caches that have recovery enabled. If the default recovery cache is overridden, then the specified recovery cache must use a TransactionManagerLookup that returns a different transaction manager than the one used by the cache itself.

8.8.5. Integration with the transaction manager

Even though this is transaction manager specific, generally a transaction manager would need a reference to a **XAResource** implementation in order to invoke **XAResource.recover()** on it. In order to obtain a reference to an Data Grid **XAResource** following API can be used:

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

It is a common practice to run the recovery in a different process from the one running the transaction.

8.8.6. Reconciliation

The transaction manager informs the system administrator on in-doubt transaction in a proprietary way. At this stage it is assumed that the system administrator knows transaction's XID (a byte array).

A normal recovery flow is:

- **STEP 1**: The system administrator connects to an Data Grid server through JMX, and lists the in doubt transactions. The image below demonstrates JConsole connecting to an Data Grid node that has an in doubt transaction.

Figure 8.1. Show in-doubt transactions
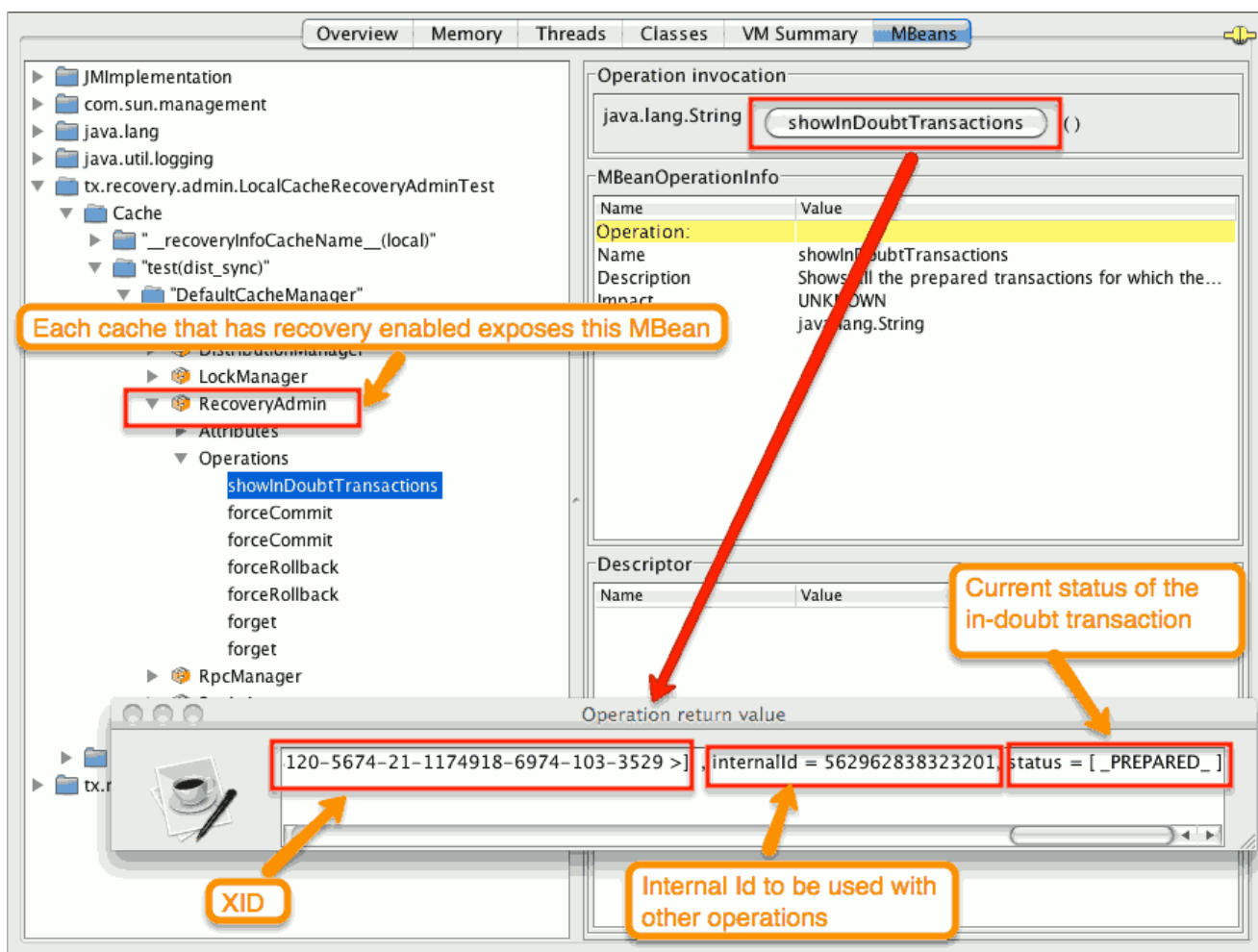
The status of each in-doubt transaction is displayed(in this example " *PREPARED* "). There might be multiple elements in the status field, e.g. "PREPARED" and "COMMITTED" in the case the transaction committed on certain nodes but not on all of them.

- **STEP 2**: The system administrator visually maps the XID received from the transaction manager to an Data Grid internal id, represented as a number. This step is needed because the XID, a byte array, cannot conveniently be passed to the JMX tool (e.g. JConsole) and then re-assembled on Data Grid's side.

- **STEP 3**: The system administrator forces the transaction's commit/rollback through the corresponding jmx operation, based on the internal id. The image below is obtained by forcing the commit of the transaction based on its internal id.

Figure 8.2. Force commit
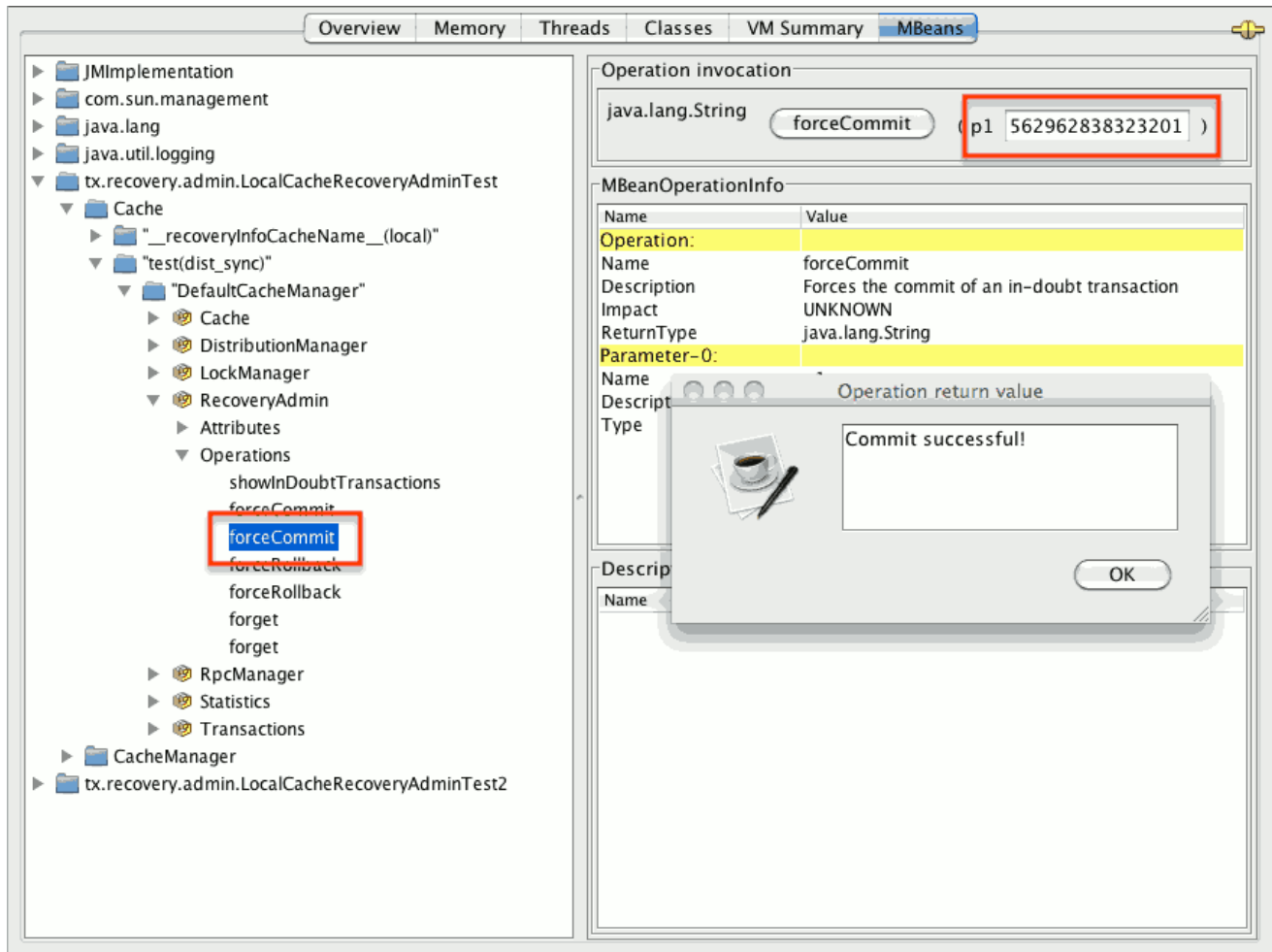
TIP

All JMX operations described above can be executed on any node, regardless of where the transaction originated.

8.8.6.1. Force commit/rollback based on XID

XID-based JMX operations for forcing in-doubt transactions' commit/rollback are available as well: these methods receive byte[] arrays describing the XID instead of the number associated with the transactions (as previously described at step 2). These can be useful e.g. if one wants to set up an automatic completion job for certain in-doubt transactions. This process is plugged into transaction manager's recovery and has access to the transaction manager's XID objects.

CHAPTER 9. EXECUTING CODE IN THE GRID

The main benefit of a Cache is the ability to very quickly lookup a value by its key, even across machines. In fact this use alone is probably the reason many users use Data Grid. However Data Grid can provide many more benefits that aren't immediately apparent. Since Data Grid is usually used in a cluster of machines we also have features available that can help utilize the entire cluster for performing the user's desired workload.

> **NOTE**
>
> This section covers only executing code in the grid using an embedded cache, if you are using a remote cache you should review details about executing code in the remote grid.

9.1. CLUSTER EXECUTOR

Since you have a group of machines, it makes sense to leverage their combined computing power for executing code on all of them them. The cache manager comes with a nice utility that allows you to execute arbitrary code in the cluster. Note this feature requires no Cache to be used. This Cluster Executor can be retrieved by calling executor() on the **EmbeddedCacheManager**. This executor is retrievable in both clustered and non clustered configurations.
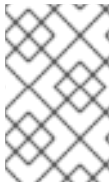
> **NOTE**
>
> The ClusterExecutor is specifically designed for executing code where the code is not reliant upon the data in a cache and is used instead as a way to help users to execute code easily in the cluster.

This manager was built specifically using Java 8 and such has functional APIs in mind, thus all methods take a functional interface as an argument. Also since these arguments will be sent to other nodes they need to be serializable. We even used a nice trick to ensure our lambdas are immediately Serializable. That is by having the arguments implement both Serializable and the real argument type (ie. Runnable or Function). The JRE will pick the most specific class when determining which method to invoke, so in that case your lambdas will always be serializable. It is also possible to use an Externalizer to possibly reduce message size further.

The manager by default will submit a given command to all nodes in the cluster including the node where it was submitted from. You can control on which nodes the task is executed on by using the **filterTargets** methods as is explained in the section.

9.1.1. Filtering execution nodes

It is possible to limit on which nodes the command will be ran. For example you may want to only run a computation on machines in the same rack. Or you may want to perform an operation once in the local site and again on a different site. A cluster executor can limit what nodes it sends requests to at the scope of same or different machine, rack or site level.

SameRack.java

```
EmbeddedCacheManager manager = ...;
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_RACK).submit(...)
```

To use this topology base filtering you must enable topology aware consistent hashing through Server Hinting.

You can also filter using a predicate based on the **Address** of the node. This can also be optionally combined with topology based filtering in the previous code snippet.

We also allow the target node to be chosen by any means using a **Predicate** that will filter out which nodes can be considered for execution. Note this can also be combined with Topology filtering at the same time to allow even more fine control of where you code is executed within the cluster.

Predicate.java

```
EmbeddedCacheManager manager = ...;
// Just filter
manager.executor().filterTargets(a -> a.equals(..)).submit(...)
// Filter only those in the desired topology
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_SITE, a -> a.equals(..)).submit(...)
```

9.1.2. Timeout

Cluster Executor allows for a timeout to be set per invocation. This defaults to the distributed sync timeout as configured on the Transport Configuration. This timeout works in both a clustered and non clustered cache manager. The executor may or may not interrupt the threads executing a task when the timeout expires. However when the timeout occurs any **Consumer** or **Future** will be completed passing back a **TimeoutException**. This value can be overridden by ivoking the timeout method and supplying the desired duration.

9.1.3. Single Node Submission

Cluster Executor can also run in single node submission mode instead of submitting the command to all nodes it will instead pick one of the nodes that would have normally received the command and instead submit it it to only one. Each submission will possibly use a different node to execute the task on. This can be very useful to use the ClusterExecutor as a **java.util.concurrent.Executor** which you may have noticed that ClusterExecutor implements.

SingleNode.java

```
EmbeddedCacheManager manager = ...;
manager.executor().singleNodeSubmission().submit(...)
```

9.1.3.1. Failover

When running in single node submission it may be desirable to also allow the Cluster Executor handle cases where an exception occurred during the processing of a given command by retrying the command again. When this occurs the Cluster Executor will choose a single node again to resubmit the command to up to the desired number of failover attempts. Note the chosen node could be any node that passes the topology or predicate check. Failover is enabled by invoking the overridden singleNodeSubmission method. The given command will be resubmitted again to a single node until either the command completes without exception or the total submission amount is equal to the provided failover count.

9.1.4. Example: PI Approximation

This example shows how you can use the ClusterExecutor to estimate the value of PI.

Pi approximation can greatly benefit from parallel distributed execution via Cluster Executor. Recall that area of the square is $S_a = 4r^2$ and area of the circle is $C_a = pi*r^2$. Substituting r^2 from the second equation into the first one it turns out that $pi = 4 * C_a/S_a$. Now, image that we can shoot very large

number of darts into a square; if we take ratio of darts that land inside a circle over a total number of darts shot we will approximate Ca/Sa value. Since we know that pi = 4 * Ca/Sa we can easily derive approximate value of pi. The more darts we shoot the better approximation we get. In the example below we shoot 1 billion darts but instead of "shooting" them serially we parallelize work of dart shooting across the entire Data Grid cluster. Note this will work in a cluster of 1 was well, but will be slower.

```java
public class PiAppx {

  public static void main (String [] arg){
    EmbeddedCacheManager cacheManager = ..
    boolean isCluster = ..

    int numPoints = 1_000_000_000;
    int numServers = isCluster ? cacheManager.getMembers().size() : 1;
    int numberPerWorker = numPoints / numServers;

    ClusterExecutor clusterExecutor = cacheManager.executor();
    long start = System.currentTimeMillis();
    // We receive results concurrently - need to handle that
    AtomicLong countCircle = new AtomicLong();
    CompletableFuture<Void> fut = clusterExecutor.submitConsumer(m -> {
      int insideCircleCount = 0;
      for (int i = 0; i < numberPerWorker; i++) {
        double x = Math.random();
        double y = Math.random();
        if (insideCircle(x, y))
          insideCircleCount++;
      }
      return insideCircleCount;
    }, (address, count, throwable) -> {
      if (throwable != null) {
        throwable.printStackTrace();
        System.out.println("Address: " + address + " encountered an error: " + throwable);
      } else {
        countCircle.getAndAdd(count);
      }
    });
    fut.whenComplete((v, t) -> {
      // This is invoked after all nodes have responded with a value or exception
      if (t != null) {
        t.printStackTrace();
        System.out.println("Exception encountered while waiting:" + t);
      } else {
        double appxPi = 4.0 * countCircle.get() / numPoints;

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

    // May have to sleep here to keep alive if no user threads left
  }

  private static 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 10. STREAMS

You may want to process a subset or all data in the cache to produce a result. This may bring thoughts of Map Reduce. Data Grid allows the user to do something very similar but utilizes the standard JRE APIs to do so. Java 8 introduced the concept of a Stream which allows functional-style operations on collections rather than having to procedurally iterate over the data yourself. Stream operations can be implemented in a fashion very similar to MapReduce. Streams, just like MapReduce allow you to perform processing upon the entirety of your cache, possibly a very large data set, but in an efficient way.

> **NOTE**
>
> Streams are the preferred method when dealing with data that exists in the cache because streams automatically adjust to cluster topology changes.

Also since we can control how the entries are iterated upon we can more efficiently perform the operations in a cache that is distributed if you want it to perform all of the operations across the cluster concurrently.

A stream is retrieved from the entrySet, keySet or values collections returned from the Cache by invoking the stream or parallelStream methods.

10.1. COMMON STREAM OPERATIONS

This section highlights various options that are present irrespective of what type of underlying cache you are using.

10.2. KEY FILTERING

It is possible to filter the stream so that it only operates upon a given subset of keys. This can be done by invoking the filterKeys method on the **CacheStream**. This should always be used over a Predicate filter and will be faster if the predicate was holding all keys.

If you are familiar with the **AdvancedCache** interface you may be wondering why you even use getAll over this keyFilter. There are some small benefits (mostly smaller payloads) to using getAll if you need the entries as is and need them all in memory in the local node. However if you need to do processing on these elements a stream is recommended since you will get both distributed and threaded parallelism for free.

10.3. SEGMENT BASED FILTERING

> **NOTE**
>
> This is an advanced feature and should only be used with deep knowledge of Data Grid segment and hashing techniques. These segments based filtering can be useful if you need to segment data into separate invocations. This can be useful when integrating with other tools such as Apache Spark.

This option is only supported for replicated and distributed caches. This allows the user to operate upon a subset of data at a time as determined by the KeyPartitioner. The segments can be filtered by invoking filterKeySegments method on the **CacheStream**. This is applied after the key filter but before any intermediate operations are performed.

10.4. LOCAL/INVALIDATION

A stream used with a local or invalidation cache can be used just the same way you would use a stream on a regular collection. Data Grid handles all of the translations if necessary behind the scenes and works with all of the more interesting options (ie. storeAsBinary and a cache loader). Only data local to the node where the stream operation is performed will be used, for example invalidation only uses local entries.

10.5. EXAMPLE

The code below takes a cache and returns a map with all the cache entries whose values contain the string "JBoss"

```
Map<Object, String> jbossValues =
cache.entrySet().stream()
    .filter(e -> e.getValue().contains("JBoss"))
    .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

10.6. DISTRIBUTION/REPLICATION/SCATTERED

This is where streams come into their stride. When a stream operation is performed it will send the various intermediate and terminal operations to each node that has pertinent data. This allows processing the intermediate values on the nodes owning the data, and only sending the final results back to the originating nodes, improving performance.

10.6.1. Rehash Aware

Internally the data is segmented and each node only performs the operations upon the data it owns as a primary owner. This allows for data to be processed evenly, assuming segments are granular enough to provide for equal amounts of data on each node.

When you are utilizing a distributed cache, the data can be reshuffled between nodes when a new node joins or leaves. Distributed Streams handle this reshuffling of data automatically so you don't have to worry about monitoring when nodes leave or join the cluster. Reshuffled entries may be processed a second time, and we keep track of the processed entries at the key level or at the segment level (depending on the terminal operation) to limit the amount of duplicate processing.

It is possible but highly discouraged to disable rehash awareness on the stream. This should only be considered if your request can handle only seeing a subset of data if a rehash occurs. This can be done by invoking CacheStream.disableRehashAware() The performance gain for most operations when a rehash doesn't occur is completely negligible. The only exceptions are for iterator and forEach, which will use less memory, since they do not have to keep track of processed keys.

> **WARNING**
>
> Please rethink disabling rehash awareness unless you really know what you are doing.

10.6.2. Serialization

Since the operations are sent across to other nodes they must be serializable by Data Grid marshalling. This allows the operations to be sent to the other nodes.

The simplest way is to use a CacheStream instance and use a lambda just as you would normally. Data Grid overrides all of the various Stream intermediate and terminal methods to take Serializable versions of the arguments (ie. SerializableFunction, SerializablePredicate...) You can find these methods at CacheStream. This relies on the spec to pick the most specific method as defined here.

In our previous example we used a **Collector** to collect all the results into a **Map**. Unfortunately the Collectors class doesn't produce Serializable instances. Thus if you need to use these, there are two ways to do so:

One option would be to use the CacheCollectors class which allows for a **Supplier<Collector>** to be provided. This instance could then use the Collectors to supply a **Collector** which is not serialized.

```
Map<Object, String> jbossValues = cache.entrySet().stream()
        .filter(e -> e.getValue().contains("Jboss"))
        .collect(CacheCollectors.serializableCollector(() -> Collectors.toMap(Map.Entry::getKey,
Map.Entry::getValue)));
```

Alternatively, you can avoid the use of CacheCollectors and instead use the overloaded **collect** methods that take **Supplier<Collector>**. These overloaded **collect** methods are only available via **CacheStream** interface.

```
Map<Object, String> jbossValues = cache.entrySet().stream()
        .filter(e -> e.getValue().contains("Jboss"))
        .collect(() -> Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

If however you are not able to use the **Cache** and **CacheStream** interfaces you cannot utilize **Serializable** arguments and you must instead cast the lambdas to be **Serializable** manually by casting the lambda to multiple interfaces. It is not a pretty sight but it gets the job done.

```
Map<Object, String> jbossValues = map.entrySet().stream()
        .filter((Serializable & Predicate<Map.Entry<Object, String>>) e ->
e.getValue().contains("Jboss"))
        .collect(CacheCollectors.serializableCollector(() -> Collectors.toMap(Map.Entry::getKey,
Map.Entry::getValue)));
```

The recommended and most performant way is to use an **AdvancedExternalizer** as this provides the smallest payload. Unfortunately this means you cannot use lamdbas as advanced externalizers require defining the class before hand.

You can use an advanced externalizer as shown below:

```
Map<Object, String> jbossValues = cache.entrySet().stream()
        .filter(new ContainsFilter("Jboss"))
        .collect(() -> Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

class ContainsFilter implements Predicate<Map.Entry<Object, String>> {
  private final String target;

  ContainsFilter(String target) {
    this.target = target;
  }
```

```java
      @Override
      public boolean test(Map.Entry<Object, String> e) {
         return e.getValue().contains(target);
      }
   }

   class JbossFilterExternalizer implements AdvancedExternalizer<ContainsFilter> {

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

      @Override
      public Integer getId() {
         return CUSTOM_ID;
      }

      @Override
      public void writeObject(ObjectOutput output, ContainsFilter object) throws IOException {
         output.writeUTF(object.target);
      }

      @Override
      public ContainsFilter readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
         return new ContainsFilter(input.readUTF());
      }
   }
```

You could also use an advanced externalizer for the collector supplier to reduce the payload size even further.

```java
Map<Object, String> map = (Map<Object, String>) cache.entrySet().stream()
        .filter(new ContainsFilter("Jboss"))
        .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

 class ToMapCollectorSupplier<K, U> implements Supplier<Collector<Map.Entry<K, U>, ?, Map<K,
U>>> {
    static final ToMapCollectorSupplier INSTANCE = new ToMapCollectorSupplier();

    private ToMapCollectorSupplier() { }

    @Override
    public Collector<Map.Entry<K, U>, ?, Map<K, U>> get() {
       return Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue);
    }
  }

   class ToMapCollectorSupplierExternalizer implements
AdvancedExternalizer<ToMapCollectorSupplier> {

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

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

    @Override
    public void writeObject(ObjectOutput output, ToMapCollectorSupplier object) throws IOException
{
    }

    @Override
    public ToMapCollectorSupplier readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
      return ToMapCollectorSupplier.INSTANCE;
    }
  }
```

10.7. PARALLEL COMPUTATION

Distributed streams by default try to parallelize as much as possible. It is possible for the end user to control this and actually they always have to control one of the options. There are 2 ways these streams are parallelized.

Local to each node When a stream is created from the cache collection the end user can choose between invoking stream or parallelStream method. Depending on if the parallel stream was picked will enable multiple threading for each node locally. Note that some operations like a rehash aware iterator and forEach operations will always use a sequential stream locally. This could be enhanced at some point to allow for parallel streams locally.

Users should be careful when using local parallelism as it requires having a large number of entries or operations that are computationally expensive to be faster. Also it should be noted that if a user uses a parallel stream with **forEach** that the action should not block as this would be executed on the common pool, which is normally reserved for computation operations.

Remote requests When there are multiple nodes it may be desirable to control whether the remote requests are all processed at the same time concurrently or one at a time. By default all terminal operations except the iterator perform concurrent requests. The iterator, method to reduce overall memory pressure on the local node, only performs sequential requests which actually performs slightly better.

If a user wishes to change this default however they can do so by invoking the sequentialDistribution or parallelDistribution methods on the **CacheStream**.

10.8. TASK TIMEOUT

It is possible to set a timeout value for the operation requests. This timeout is used only for remote requests timing out and it is on a per request basis. The former means the local execution will not timeout and the latter means if you have a failover scenario as described above the subsequent requests each have a new timeout. If no timeout is specified it uses the replication timeout as a default timeout. You can set the timeout in your task by doing the following:

```
CacheStream<Map.Entry<Object, String>> stream = cache.entrySet().stream();
stream.timeout(1, TimeUnit.MINUTES);
```

For more information about this, please check the java doc in timeout javadoc.

10.9. INJECTION

The Stream has a terminal operation called forEach which allows for running some sort of side effect operation on the data. In this case it may be desirable to get a reference to the **Cache** that is backing this Stream. If your **Consumer** implements the CacheAware interface the **injectCache** method be invoked before the accept method from the **Consumer** interface.

10.10. DISTRIBUTED STREAM EXECUTION

Distributed streams execution works in a fashion very similar to map reduce. Except in this case we are sending zero to many intermediate operations (map, filter etc.) and a single terminal operation to the various nodes. The operation basically comes down to the following:

1. The desired segments are grouped by which node is the primary owner of the given segment

2. A request is generated to send to each remote node that contains the intermediate and terminal operations including which segments it should process

    a. The terminal operation will be performed locally if necessary

    b. Each remote node will receive this request and run the operations and subsequently send the response back

3. The local node will then gather the local response and remote responses together performing any kind of reduction required by the operations themselves.

4. Final reduced response is then returned to the user

In most cases all operations are fully distributed, as in the operations are all fully applied on each remote node and usually only the last operation or something related may be reapplied to reduce the results from multiple nodes. One important note is that intermediate values do not actually have to be serializable, it is the last value sent back that is the part desired (exceptions for various operations will be highlighted below).

Terminal operator distributed result reductions The following paragraphs describe how the distributed reductions work for the various terminal operators. Some of these are special in that an intermediate value may be required to be serializable instead of the final result.

allMatch noneMatch anyMatch

> The allMatch operation is ran on each node and then all the results are logically anded together locally to get the appropriate value. The noneMatch and anyMatch operations use a logical or instead. These methods also have early termination support, stopping remote and local operations once the final result is known.

collect

> The collect method is interesting in that it can do a few extra steps. The remote node performs everything as normal except it doesn't perform the final finisher upon the result and instead sends back the fully combined results. The local thread then combines the remote and local result into a value which is then finally finished. The key here to remember is that the final value doesn't have to be serializable but rather the values produced from the supplier and combiner methods.

count

> The count method just adds the numbers together from each node.

findAny findFirst

The findAny operation returns just the first value they find, whether it was from a remote node or locally. Note this supports early termination in that once a value is found it will not process others. Note the findFirst method is special since it requires a sorted intermediate operation, which is detailed in the exceptions section.

max min

The max and min methods find the respective min or max value on each node then a final reduction is performed locally to ensure only the min or max across all nodes is returned.

reduce

The various reduce methods 1 , 2 , 3 will end up serializing the result as much as the accumulator can do. Then it will accumulate the local and remote results together locally, before combining if you have provided that. Note this means a value coming from the combiner doesn't have to be Serializable.

10.11. KEY BASED REHASH AWARE OPERATORS

The iterator, spliterator and forEach are unlike the other terminal operators in that the rehash awareness has to keep track of what keys per segment have been processed instead of just segments. This is to guarantee an exactly once (iterator & spliterator) or at least once behavior (forEach) even under cluster membership changes.

The **iterator** and **spliterator** operators when invoked on a remote node will return back batches of entries, where the next batch is only sent back after the last has been fully consumed. This batching is done to limit how many entries are in memory at a given time. The user node will hold onto which keys it has processed and when a given segment is completed it will release those keys from memory. This is why sequential processing is preferred for the iterator method, so only a subset of segment keys are held in memory at once, instead of from all nodes.

The **forEach()** method also returns batches, but it returns a batch of keys after it has finished processing at least a batch worth of keys. This way the originating node can know what keys have been processed already to reduce chances of processing the same entry again. Unfortunately this means it is possible to have an at least once behavior when a node goes down unexpectedly. In this case that node could have been processing a batch and not yet completed one and those entries that were processed but not in a completed batch will be ran again when the rehash failure operation occurs. Note that adding a node will not cause this issue as the rehash failover doesn't occur until all responses are received.

These operations batch sizes are both controlled by the same value which can be configured by invoking distributedBatchSize method on the **CacheStream**. This value will default to the **chunkSize** configured in state transfer. Unfortunately this value is a tradeoff with memory usage vs performance vs at least once and your mileage may vary.

Using iterator with replicated and distributed caches

When a node is the primary or backup owner of all requested segments for a distributed stream, Data Grid performs the **iterator** or **spliterator** terminal operations locally, which optimizes performance as remote iterations are more resource intensive.

This optimization applies to both replicated and distributed caches. However, Data Grid performs iterations remotely when using cache stores that are both **shared** and have **write-behind** enabled. In this case performing the iterations remotely ensures consistency.

10.12. INTERMEDIATE OPERATION EXCEPTIONS

There are some intermediate operations that have special exceptions, these are skip, peek, sorted 12. & distinct. All of these methods have some sort of artificial iterator implanted in the stream processing to

guarantee correctness, they are documented as below. Note this means these operations may cause possibly severe performance degradation.

Skip

An artificial iterator is implanted up to the intermediate skip operation. Then results are brought locally so it can skip the appropriate amount of elements.

Sorted

WARNING: This operation requires having all entries in memory on the local node. An artificial iterator is implanted up to the intermediate sorted operation. All results are sorted locally. There are possible plans to have a distributed sort which returns batches of elements, but this is not yet implemented.

Distinct

WARNING: This operation requires having all or nearly all entries in memory on the local node. Distinct is performed on each remote node and then an artificial iterator returns those distinct values. Then finally all of those results have a distinct operation performed upon them.

The rest of the intermediate operations are fully distributed as one would expect.

10.13. EXAMPLES

Word Count

Word count is a classic, if overused, example of map/reduce paradigm. Assume we have a mapping of key → sentence stored on Data Grid nodes. Key is a String, each sentence is also a String, and we have to count occurrence of all words in all sentences available. The implementation of such a distributed task could be defined as follows:

```java
public class WordCountExample {

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

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

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

```
      c2.put("214", "RedHat community");

      Map<String, Long> wordCountMap = c1.entrySet().parallelStream()
        .map(e -> e.getValue().split("\\s"))
        .flatMap(Arrays::stream)
        .collect(() -> Collectors.groupingBy(Function.identity(), Collectors.counting()));
  }
}
```

In this case it is pretty simple to do the word count from the previous example.

However what if we want to find the most frequent word in the example? If you take a second to think about this case you will realize you need to have all words counted and available locally first. Thus we actually have a few options.

We could use a finisher on the collector, which is invoked on the user thread after all the results have been collected. Some redundant lines have been removed from the previous example.

```
public class WordCountExample {
  public static void main(String[] args) {
    // Lines removed

    String mostFrequentWord = c1.entrySet().parallelStream()
      .map(e -> e.getValue().split("\\s"))
      .flatMap(Arrays::stream)
      .collect(() -> 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;
          }));

  }
}
```

Unfortunately the last step is only going to be ran in a single thread, which if we have a lot of words could be quite slow. Maybe there is another way to parallelize this with Streams.

We mentioned before we are in the local node after processing, so we could actually use a stream on the map results. We can therefore use a parallel stream on the results.

```
public class WordFrequencyExample {
  public static void main(String[] args) {
    // Lines removed

    Map<String, Long> wordCount = c1.entrySet().parallelStream()
        .map(e -> e.getValue().split("\\s"))
        .flatMap(Arrays::stream)
        .collect(() -> Collectors.groupingBy(Function.identity(), Collectors.counting()));
```

```
    Optional<Map.Entry<String, Long>> mostFrequent =
wordCount.entrySet().parallelStream().reduce(
        (e1, e2) -> e1.getValue() > e2.getValue() ? e1 : e2);
```

This way you can still utilize all of the cores locally when calculating the most frequent element.

Remove specific entries

Distributed streams can also be used as a way to modify data where it lives. For example you may want to remove all entries in your cache that contain a specific word.

```java
public class RemoveBadWords {
  public static void main(String[] args) {
    // Lines removed
    String word = ..

    c1.entrySet().parallelStream()
       .filter(e -> e.getValue().contains(word))
       .forEach((c, e) -> c.remove(e.getKey()));
```

If we carefully note what is serialized and what is not, we notice that only the word along with the operations are serialized across to other nods as it is captured by the lambda. However the real saving piece is that the cache operation is performed on the primary owner thus reducing the amount of network traffic required to remove these values from the cache. The cache is not captured by the lambda as we provide a special BiConsumer method override that when invoked on each node passes the cache to the BiConsumer

One thing to keep in mind using the **forEach** command in this manner is that the underlying stream obtains no locks. The cache remove operation will still obtain locks naturally, but the value could have changed from what the stream saw. That means that the entry could have been changed after the stream read it but the remove actually removed it.

We have specifically added a new variant which is called **LockedStream**.

Plenty of other examples

The **Streams** API is a JRE tool and there are lots of examples for using it. Just remember that your operations need to be Serializable in some way.

CHAPTER 11. JCACHE (JSR-107) API

Data Grid provides an implementation of JCache 1.0 API (JSR-107). JCache specifies 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 type 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 the Data Grid implementation of the specification, and explains key aspects of the API.

11.1. CREATING EMBEDDED CACHES

Prerequisites

1. Ensure that **cache-api** is on your classpath.

2. Add the following dependency to your **pom.xml**:

```xml
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-jcache</artifactId>
</dependency>
```

Procedure

- Create embedded caches that use the default JCache API configuration as follows:

```java
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>());
```

11.1.1. Configuring embedded caches

- Pass the URI for custom Data Grid configuration to the **CachingProvider.getCacheManager(URI)** call as follows:

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

// Load configuration from an absolute filesystem path
URI uri = URI.create("file:///path/to/infinispan.xml");
// Load configuration from a classpath resource
// URI uri = this.getClass().getClassLoader().getResource("infinispan.xml").toURI();

// Create a cache manager using the above configuration
CacheManager cacheManager = Caching.getCachingProvider().getCacheManager(uri,
    this.getClass().getClassLoader(), null);
```

> **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. 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 Data Grid, data stored must be marshallable.

Alternatively, JCache can be configured to store data by reference (just like Data Grid or JDK Collections work). To do that, simply call:

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

11.2. CREATING REMOTE CACHES

Prerequisites

1. Ensure that **cache-api** is on your classpath.

2. Add the following dependency to your **pom.xml**:

   ```xml
   <dependency>
     <groupId>org.infinispan</groupId>
     <artifactId>infinispan-jcache-remote</artifactId>
   </dependency>
   ```

Procedure

- Create caches on remote Data Grid servers and use the default JCache API configuration as follows:

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

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

11.2.1. Configuring remote caches

Hot Rod configuration files include **infinispan.client.hotrod.cache.*** properties that you can use to customize remote caches.

- Pass the URI for your **hotrod-client.properties** file to the **CachingProvider.getCacheManager(URI)** call as follows:

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

// Load configuration from an absolute filesystem path
URI uri = URI.create("file:///path/to/hotrod-client.properties");
// Load configuration from a classpath resource
// URI uri = this.getClass().getClassLoader().getResource("hotrod-client.properties").toURI();

// Retrieve the system wide cache manager via org.infinispan.jcache.remote.JCachingProvider
CacheManager cacheManager =
Caching.getCachingProvider("org.infinispan.jcache.remote.JCachingProvider")
    .getCacheManager(uri, this.getClass().getClassLoader(), null);
```

11.3. STORE AND RETRIEVE DATA

Even though JCache API does not extend neither java.util.Map not java.util.concurrent.ConcurrentMap, it providers 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. So, 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 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, V) which avoids returning the potentially expensive operation of returning the previous value.

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

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

Operation	java.util.concurrent.Concurrent Map<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)**
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's obvious to see 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 JCache API. In fact, there's 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:

Operation	java.util.concurrent.Concurrent Map<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.Concurrent Map<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()**

11.5. CLUSTERING JCACHE INSTANCES

Data Grid JCache implementation goes beyond the specification in order to provide the possibility to cluster caches using the standard API. Given a Data Grid configuration file configured to replicate caches like this:

infinispan.xml

```
<infinispan>
  <cache-container default-cache="namedCache">
    <transport cluster="jcache-cluster" />
    <replicated-cache name="namedCache" />
  </cache-container>
</infinispan>
```

You can 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);
  }
}
```

CHAPTER 12. MULTIMAP CACHE

MutimapCache is a type of Data Grid Cache that maps keys to values in which each key can contain multiple values.

12.1. INSTALLATION AND CONFIGURATION

pom.xml

```xml
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-multimap</artifactId>
</dependency>
```

12.2. MULTIMAPCACHE API

MultimapCache API exposes several methods to interact with the Multimap Cache. These methods are non-blocking in most cases; see limitations for more information.

```java
public interface MultimapCache<K, V> {

    CompletableFuture<Optional<CacheEntry<K, Collection<V>>>> getEntry(K key);

    CompletableFuture<Void> remove(SerializablePredicate<? super V> p);

    CompletableFuture<Void> put(K key, V value);

    CompletableFuture<Collection<V>> get(K key);

    CompletableFuture<Boolean> remove(K key);

    CompletableFuture<Boolean> remove(K key, V value);

    CompletableFuture<Void> remove(Predicate<? super V> p);

    CompletableFuture<Boolean> containsKey(K key);

    CompletableFuture<Boolean> containsValue(V value);

    CompletableFuture<Boolean> containsEntry(K key, V value);

    CompletableFuture<Long> size();

    boolean supportsDuplicates();

}
```

CompletableFuture<Void> put(K key, V value)

Puts a key-value pair in the multimap cache.

```java
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");
        });
```

The output of this code is as follows:

```
Marie is a girl name
Oihana is a girl name
```

CompletableFuture<Collection<V>> get(K key)

Asynchronous that returns a view collection of the values associated with key in this multimap cache, if any. Any changes to the retrieved collection won't change the values in this multimap cache. When this method returns an empty collection, it means the key was not found.

CompletableFuture<Boolean> remove(K key)

Asynchronous that removes the entry associated with the key from the multimap cache, if such exists.

CompletableFuture<Boolean> remove(K key, V value)

Asynchronous that removes a key-value pair from the multimap cache, if such exists.

CompletableFuture<Void> remove(Predicate<? super V> p)

Asynchronous method. Removes every value that match the given predicate.

CompletableFuture<Boolean> containsKey(K key)

Asynchronous that returns true if this multimap contains the key.

CompletableFuture<Boolean> containsValue(V value)

Asynchronous that returns true if this multimap contains the value in at least one key.

CompletableFuture<Boolean> containsEntry(K key, V value)

Asynchronous that returns true if this multimap contains at least one key-value pair with the value.

CompletableFuture<Long> size()

Asynchronous that returns the number of key-value pairs in the multimap cache. It doesn't return the distinct number of keys.

boolean supportsDuplicates()

Asynchronous that returns true if the multimap cache supports duplicates. This means that the content of the multimap can be 'a' → ['1', '1', '2']. For now this method will always return false, as duplicates are not yet supported. The existence of a given value is determined by 'equals' and `hashcode' method's contract.

12.3. CREATING A MULTIMAP CACHE

Currently the MultimapCache is configured as a regular cache. This can be done either by code or XML configuration. See how to configure a regular Cache in the section link to [configure a cache].

12.3.1. Embedded mode

```
// 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);
```

12.4. LIMITATIONS

In almost every case the Multimap Cache will behave as a regular Cache, but some limitations exist in the current version, as follows:

12.4.1. Support for duplicates

Duplicates are not supported yet. This means that the multimap won't contain any duplicate key-value pair. Whenever put method is called, if the key-value pair already exist, this key-value par won't be added. Methods used to check if a key-value pair is already present in the Multimap are the **equals** and **hashcode**.

12.4.2. Eviction

For now, the eviction works per key, and not per key-value pair. This means that whenever a key is evicted, all the values associated with the key will be evicted too.

12.4.3. Transactions

Implicit transactions are supported through the auto-commit and all the methods are non blocking. Explicit transactions work without blocking in most of the cases. Methods that will block are **size**, **containsEntry** and **remove(Predicate<? super V> p)**

CHAPTER 13. DATA GRID MODULES FOR RED HAT JBOSS EAP

To use Data Grid inside applications deployed to Red Hat JBoss EAP, you should install Data Grid modules that:

- Let you deploy applications without packaging Data Grid JAR files in your WAR or EAR file.

- Allow you to use a Data Grid version that is independent to the one bundled with Red Hat JBoss EAP.

> **IMPORTANT**
>
> Red Hat JBoss EAP (EAP) applications can directly handle the **infinispan** subsystem without the need to separately install Data Grid modules. Red Hat provides support for this functionality since EAP version 7.4. However, your deployment requires the EAP modules to use advanced capabilities such as indexing and querying.

13.1. INSTALLING DATA GRID MODULES

Download and install Data Grid modules for Red Hat JBoss EAP.

Prerequisites

1. JDK 8 or later.

2. An existing Red Hat JBoss EAP installation.

Procedure

1. Log in to the Red Hat customer portal.

2. Download the ZIP archive for the modules from the Data Grid software downloads.

3. Extract the ZIP archive and copy the contents of **modules** to the **modules** directory of your Red Hat JBoss EAP installation so that you get the resulting structure:
   $EAP_HOME/modules/system/add-ons/rhdg/org/infinispan/rhdg-8.3

13.2. CONFIGURING APPLICATIONS TO USE DATA GRID MODULES

After you install Data Grid modules for Red Hat JBoss EAP, configure your application to use Data Grid functionality.

Procedure

1. In your project **pom.xml** file, mark the required Data Grid dependencies as *provided*.

2. Configure your artifact archiver to generate the appropriate **MANIFEST.MF** file.

pom.xml

```
<dependencies>
  <dependency>
```

```
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-core</artifactId>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-cachestore-jdbc</artifactId>
      <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:rhdg-8.3 services</Dependencies>
          </manifestEntries>
        </archive>
       </configuration>
     </plugin>
   </plugins>
  </build>
```

Data Grid functionality is packaged as a single module, **org.infinispan**, that you can add as an entry to your application's manifest as follows:

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan:rhdg-8.3 services
```

AWS dependencies

If you require AWS dependencies, such as S3_PING, add the following module to your application's manifest:

```
Manifest-Version: 1.0
Dependencies: com.amazonaws.aws-java-sdk:rhdg-8.3 services
```

CHAPTER 14. EXTERNALIZING HTTP SESSIONS FROM RED HAT JBOSS WEB SERVER TO RED HAT DATA GRID

Achieve high availability by externalizing HTTP session data from JBoss Web Server deployments to Data Grid Server clusters through the **org.apache.catalina.Manager** interface.

14.1. INSTALLING THE TOMCAT SESSION CLIENT

Install the Tomcat session client to externalize HTTP sessions from Red Hat JBoss Web Server applications to Red Hat Data Grid.

Procedure

1. Download the **redhat-datagrid-8.1.1-tomcat<$version>-session-client.zip** archive from the Data Grid Software Downloads.

2. Extract the archive to your filesystem.

3. Copy the contents of the **lib/** directory from the extracted archive into **$CATALINA_HOME/lib**.

14.2. CONFIGURING THE SESSION MANAGER

Configure the **HotRodManager** class for the session manager to define how the Tomcat session client connects to Red Hat Data Grid Server and stores data in remote caches.

Prerequisites

- Install the Tomcat session client.

- Install at least one Data Grid Server instance.

- Create a cache on Data Grid Server to use as a template for storing HTTP session data.

Procedure

1. Open either **$CATALINA_HOME/conf/context.xml** or **/WEB-INF/context.xml** for editing.

2. Specify **org.wildfly.clustering.tomcat.hotrod.HotRodManager** as the value for the **className** property.

3. Specify the name of the cache to use as a template with the **configurationName** property.

4. Define any other configuration properties for the **HotRodManager** class as appropriate.

5. Set Hot Rod client configuration properties without the **infinispan.client.hotrod.** prefix.

   a. Specify the list of Data Grid Server nodes with the **server_list** property.

   b. Specify Data Grid credentials with the **auth_username** and **auth_password** properties.

6. Specify common attributes for the Tomcat session manager as required.

7. Save and close **context.xml**.

Configuration example

```
<Manager className="org.wildfly.clustering.tomcat.hotrod.HotRodManager"
    configurationName="mycache"
    persistenceStrategy="FINE"
    maxActiveSessions="100"
    server_list="192.0.2.0:11222;192.0.2.0:11223;192.0.2.0:11224"
    protocol_version="2.9"
    auth_username="admin"
    auth_password="changeme"
    auth_realm="default"
    sasl_mechanism="DIGEST-MD5"
    auth_server_name="infinispan"/>
```

Verification

To verify that the Tomcat session client stores data in remote caches, do the following:

1. Open the Data Grid Console in any browser.
   By default the console is available at **http://127.0.0.1:11222/console/**.

2. Check that the Tomcat session client has created caches for each deployed application.

14.2.1. Hot Rod manager configuration properties

The following table lists and describes configuration properties for the **HotRodManager** class:

Property	Description
className	Specifies **org.wildfly.clustering.tomcat.hotrod.HotRodManager** as the session manager.
configurationName	Specifies a remote cache on Data Grid Server to use as a template for storing HTTP session data.
persistenceStrategy	Defines how sessions map to entries in the cache. **COARSE** stores all attributes of a session in a single cache entry. This is the default. **FINE** stores session attributes in separate cache entries.
maxActiveSessions	Defines the maximum number of sessions to store in the cache. The default is no maximum (limitless).

Additional resources

- Data Grid documentation

- Common Attributes for Tomcat 7.0 Manager

- [Common Attributes for Tomcat 8.0 Manager](#)

- [Common Attributes for Tomcat 8.5 Manager](#)

- [Common Attributes for Tomcat 9.0 Manager](#)

CHAPTER 15. CUSTOM INTERCEPTORS

IMPORTANT

Custom interceptors are deprecated in Data Grid and will be removed in a future version.

Custom interceptors are a way of extending Data Grid by being able to influence or respond to any modifications to cache. Example of such modifications are: elements are added/removed/updated or transactions are committed.

15.1. ADDING CUSTOM INTERCEPTORS DECLARATIVELY

Custom interceptors can be added on a per named cache basis. This is because each named cache have its own interceptor stack. Following xml snippet depicts the ways in which a custom interceptor can be added.

```xml
<local-cache name="cacheWithCustomInterceptors">
  <!-- Define custom interceptors. -->
  <!-- Custom interceptors should extend
       org.infinispan.interceptors.BaseCustomAsyncInterceptor -->
  <custom-interceptors>
    <interceptor position="FIRST" class="com.mycompany.CustomInterceptor1">
      <property name="attributeOne">value1</property>
      <property name="attributeTwo">value2</property>
    </interceptor>
    <interceptor position="LAST" class="com.mycompany.CustomInterceptor2"/>
    <interceptor index="3" class="com.mycompany.CustomInterceptor1"/>
    <interceptor before="org.infinispanpan.interceptors.CallInterceptor"
            class="com.mycompany.CustomInterceptor2"/>
    <interceptor after="org.infinispanpan.interceptors.CallInterceptor"
            class="com.mycompany.CustomInterceptor1"/>
  </custom-interceptors>
</local-cache>
```

15.2. ADDING CUSTOM INTERCEPTORS PROGRAMMATICALLY

In order to do that one needs to obtain a reference to the **AdvancedCache**. This can be done as follows:

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

Then one of the *addInterceptor()* methods should be used to add the actual interceptor. For further documentation refer to **AdvancedCache** javadoc.

15.3. CUSTOM INTERCEPTOR DESIGN

When writing a custom interceptor, you need to abide by the following rules.

- Custom interceptors must declare a public, empty constructor to enable construction.

- Custom interceptors will have setters for any property defined through property tags used in the XML configuration.