Developing for Infinispan 12.0

Table of Contents

Chapter 1. Configuring the Infinispan Maven Repository

Infinispan Java distributions are available from Maven.

Infinispan artifacts are available from Maven central. See the org.infinispan group for available Infinispan artifacts.

1.1. Configuring Your Infinispan 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 Infinispan 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 Infinispan artifact you add as a dependency to your project.

4. Save and close `pom.xml`.

The following example shows the Infinispan version and BOM:

```
<properties>
  <version.infinispan>12.0.0.Dev01</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 Infinispan artifacts as dependencies to your `pom.xml` as required.

Chapter 2. Cache Managers

The main entry point to Infinispan is the `CacheManager` interface that lets you:

- Configure and obtain caches.
- Manage and monitor clustered Infinispan nodes.
- Execute code across your cluster.

If you embed Infinispan in your application, then you use an `EmbeddedCacheManager`. If you run Infinispan 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(); ①
```

① 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. Infinispan Cache Interface

Infinispan 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 Infinispan'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 Infinispan, 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, Infinispan'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* , Infinispan 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

Infinispan's Cache class contains a different 'put' operation called putForExternalRead . This operation is particularly useful when Infinispan 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, Infinispan 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

Infinispan 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.

> 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.

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

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.

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`.

```java
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.

> 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).

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.

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

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.

Infinispan 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 Infinispan 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) { }
}
```

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., Infinispan 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:

```java
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 Infinispan 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. Data Encoding and MediaTypes

Encoding is the data conversion operation done by Infinispan caches before storing data, and when reading back from storage.

4.1. Overview

Encoding allows dealing with a certain data format during API calls (map, listeners, stream, etc) while the format effectively stored is different.

The data conversions are handled by instances of *org.infinispan.commons.dataconversion.Encoder* :

```
public interface Encoder {

   /**
    * Convert data in the read/write format to the storage format.
    *
    * @param content data to be converted, never null.
    * @return Object in the storage format.
    */
   Object toStorage(Object content);

   /**
    * Convert from storage format to the read/write format.
    *
    * @param content data as stored in the cache, never null.
    * @return data in the read/write format
    */
   Object fromStorage(Object content);

   /**
     * Returns the {@link MediaType} produced by this encoder or null if the storage
format is not known.
     */
   MediaType getStorageFormat();
}
```

4.2. Default encoders

Infinispan automatically picks the Encoder depending on the cache configuration. The table below shows which internal Encoder is used for several configurations:

Mode	Configuration	Encoder	Description
Embedded/Server	Default	IdentityEncoder	Passthrough encoder, no conversion done

Mode	Configuration	Encoder	Description
Embedded	StorageType.OFF_HEAP	GlobalMarshallerEncoder	Use the Infinispan internal marshaller to convert to byte[]. May delegate to the configured marshaller in the cache manager.
Embedded	StorageType.BINARY	BinaryEncoder	Use the Infinispan internal marshaller to convert to byte[], except for primitives and String.
Server	StorageType.OFF_HEAP	IdentityEncoder	Store byte[]s directly as received by remote clients

4.3. Overriding programmatically

It is possible to override programmatically the encoding used for both keys and values, by calling the *.withEncoding()* method variants from *AdvancedCache*.

Example, consider the following cache configured as OFF_HEAP:

```
// Read and write POJO, storage will be byte[] since for
// OFF_HEAP the GlobalMarshallerEncoder is used internally:
cache.put(1, new Pojo())
Pojo value = cache.get(1)

// Get the content in its stored format by overriding
// the internal encoder with a no-op encoder (IdentityEncoder)
Cache<?,?> rawContent = cache.getAdvancedCache().withEncoding(IdentityEncoder.class);
byte[] marshalled = (byte[]) rawContent.get(1);
```

The override can be useful if any operation in the cache does not require decoding, such as counting number of entries, or calculating the size of byte[] of an OFF_HEAP cache.

4.4. Defining custom Encoders

A custom encoder can be registered in the *EncoderRegistry*.

Ensure that the registration is done in every node of the cluster, before starting the caches.

Consider a custom encoder used to compress/decompress with gzip:

```java
public class GzipEncoder implements Encoder {

   @Override
   public Object toStorage(Object content) {
      assert content instanceof String;
      return compress(content.toString());
   }

   @Override
   public Object fromStorage(Object content) {
      assert content instanceof byte[];
      return decompress((byte[]) content);
   }

   private byte[] compress(String str) {
      try (ByteArrayOutputStream baos = new ByteArrayOutputStream();
           GZIPOutputStream gis = new GZIPOutputStream(baos)) {
         gis.write(str.getBytes("UTF-8"));
         gis.close();
         return baos.toByteArray();
      } catch (IOException e) {
         throw new RuntimeException("Unabled to compress", e);
      }
   }

   private String decompress(byte[] compressed) {
      try (GZIPInputStream gis = new GZIPInputStream(new ByteArrayInputStream
(compressed));
           BufferedReader bf = new BufferedReader(new InputStreamReader(gis, "UTF-8")
)) {
         StringBuilder result = new StringBuilder();
         String line;
         while ((line = bf.readLine()) != null) {
            result.append(line);
         }
         return result.toString();
      } catch (IOException e) {
         throw new RuntimeException("Unable to decompress", e);
      }
   }

   @Override
   public MediaType getStorageFormat() {
      return MediaType.parse("application/gzip");
   }

   @Override
   public boolean isStorageFormatFilterable() {
      return false;
   }
```

```
    @Override
    public short id() {
        return 10000;
    }
}
```

It can be registered by:

```
GlobalComponentRegistry registry = cacheManager.getGlobalComponentRegistry();
EncoderRegistry encoderRegistry = registry.getComponent(EncoderRegistry.class);
encoderRegistry.registerEncoder(new GzipEncoder());
```

And then be used to write and read data from a cache:

```
AdvancedCache<String, String> cache = ...

// Decorate cache with the newly registered encoder, without encoding keys
(IdentityEncoder)
// but compressing values
AdvancedCache<String, String> compressingCache = (AdvancedCache<String, String>)
cache.withEncoding(IdentityEncoder.class, GzipEncoder.class);

// All values will be stored compressed...
compressingCache.put("297931749", "0412c789a37f5086f743255cfa693dd5");

// ... but API calls deals with String
String stringValue = compressingCache.get("297931749");

// Bypassing the value encoder to obtain the value as it is stored
Object value = compressingCache.withEncoding(IdentityEncoder.class).get("297931749");

// value is a byte[] which is the compressed value
```

4.5. MediaType

A Cache can optionally be configured with a `org.infinispan.commons.dataconversion.MediaType` for keys and values. By describing the data format of the cache, Infinispan is able to convert data on the fly during cache operations.

> The MediaType configuration is more suitable when storing binary data. When using server mode, it's common to have a MediaType configured and clients such as REST or Hot Rod reading and writing in different formats.

The data conversion between MediaType formats are handled by instances of `org.infinispan.commons.dataconversion.Transcoder`

```java
public interface Transcoder {

   /**
    * Transcodes content between two different {@link MediaType}.
    *
    * @param content         Content to transcode.
    * @param contentType     The {@link MediaType} of the content.
    * @param destinationType The target {@link MediaType} to convert.
    * @return the transcoded content.
    */
   Object transcode(Object content, MediaType contentType, MediaType destinationType);

   /**
    * @return all the {@link MediaType} handled by this Transcoder.
    */
   Set<MediaType> getSupportedMediaTypes();
}
```

4.5.1. Configuration

Declarative:

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

Programmatic:

```java
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg.encoding().key().mediaType("text/plain");
cfg.encoding().value().mediaType("application/json");
```

4.5.2. Overriding the MediaType Programmatically

It’s possible to decorate the Cache with a different MediaType, allowing cache operations to be executed sending and receiving different data formats.

Example:

```java
DefaultCacheManager cacheManager = new DefaultCacheManager();

// The cache will store POJO for keys and values
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.encoding().key().mediaType("application/x-java-object");
cfg.encoding().value().mediaType("application/x-java-object");

cacheManager.defineConfiguration("mycache", cfg.build());

Cache<Integer, Person> cache = cacheManager.getCache("mycache");

cache.put(1, new Person("John","Doe"));

// Wraps cache using 'application/x-java-object' for keys but JSON for values
Cache<Integer, byte[]> jsonValuesCache = (Cache<Integer, byte[]>) cache
.getAdvancedCache().withMediaType("application/x-java-object", "application/json");

byte[] json = jsonValuesCache.get(1);
```

Will return the value in JSON format:

```json
{
   "_type":"org.infinispan.sample.Person",
   "name":"John",
   "surname":"Doe"
}
```

Most Transcoders are installed when server mode is used; when using library mode, an extra dependency, *org.infinispan:infinispan-server-core* should be added to the project.

4.5.3. Transcoders and Encoders

Usually there will be none or only one data conversion involved in a cache operation:

- No conversion by default on caches using in embedded or server mode;
- *Encoder* based conversion for embedded caches without MediaType configured, but using OFF_HEAP or BINARY;
- *Transcoder* based conversion for caches used in server mode with multiple REST and Hot Rod clients sending and receiving data in different formats. Those caches will have MediaType configured describing the storage.

But it’s possible to have both encoders and transcoders being used simultaneously for advanced use cases.

Consider an example, a cache that stores marshalled objects (with jboss marshaller) content but for security reasons a transparent encryption layer should be added in order to avoid storing "plain" data to an external store. Clients should be able to read and write data in multiple formats.

This can be achieved by configuring the cache with the the MediaType that describes the storage regardless of the encoding layer:

```
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.encoding().key().mediaType("application/x-jboss-marshalling");
cfg.encoding().key().mediaType("application/x-jboss-marshalling");
```

The transparent encryption can be added by decorating the cache with a special *Encoder* that encrypts/decrypts with storing/retrieving, for example:

```
class Scrambler implements Encoder {

   public Object toStorage(Object content) {
   // Encrypt data
   }

   public Object fromStorage(Object content) {
   // Decrypt data
   }

   @Override
   public boolean isStorageFormatFilterable() {

   }

   public MediaType getStorageFormat() {
   return new MediaType("application", "scrambled");
   }

   @Override
   public short id() {
   //return id
   }
}
```

To make sure all data written to the cache will be stored encrypted, it's necessary to decorate the cache with the Encoder above and perform all cache operations in this decorated cache:

```
Cache<?,?> secureStorageCache = cache.getAdvancedCache().withEncoding(Scrambler.class
).put(k,v);
```

The capability of reading data in multiple formats can be added by decorating the cache with the desired MediaType:

```
// Obtain a stream of values in XML format from the secure cache
secureStorageCache.getAdvancedCache().withMediaType("application/xml","application/xml
").values().stream();
```

Internally, Infinispan will first apply the encoder *fromStorage* operation to obtain the entries, that will be in "application/x-jboss-marshalling" format and then apply a successive conversion to "application/xml" by using the adequate Transcoder.

Chapter 5. Protocol Interoperability

Clients exchange data with Infinispan through endpoints such as REST or Hot Rod.

Each endpoint uses a different protocol so that clients can read and write data in a suitable format. Because Infinispan can interoperate with multiple clients at the same time, it must convert data between client formats and the storage formats.

To configure Infinispan endpoint interoperability, you should define the MediaType that sets the format for data stored in the cache.

5.1. Considerations with Media Types and Endpoint Interoperability

Configuring Infinispan to store data with a specific media type affects client interoperability.

Although REST clients do support sending and receiving encoded binary data, they are better at handling text formats such as JSON, XML, or plain text.

Memcached text clients can handle String-based keys and byte[] values but cannot negotiate data types with the server. These clients do not offer much flexibility when handling data formats because of the protocol definition.

Java Hot Rod clients are suitable for handling Java objects that represent entities that reside in the cache. Java Hot Rod clients use marshalling operations to serialize and deserialize those objects into byte arrays.

Similarly, non-Java Hot Rod clients, such as the C++, C#, and Javascript clients, are suitable for handling objects in the respective languages. However, non-Java Hot Rod clients can interoperate with Java Hot Rod clients using platform independent data formats.

5.2. REST, Hot Rod, and Memcached Interoperability with Text-Based Storage

You can configure key and values with a text-based storage format.

For example, specify `text/plain; charset=UTF-8`, or any other character set, to set plain text as the media type. You can also specify a media type for other text-based formats such as JSON (`application/json`) or XML (`application/xml`) with an optional character set.

The following example configures the cache to store entries with the `text/plain; charset=UTF-8` media type:

```
<cache>
   <encoding>
      <key media-type="text/plain; charset=UTF-8"/>
      <value media-type="text/plain; charset=UTF-8"/>
   </encoding>
</cache>
```

To handle the exchange of data in a text-based format, you must configure Hot Rod clients with the `org.infinispan.commons.marshall.StringMarshaller` marshaller.

REST clients must also send the correct headers when writing and reading from the cache, as follows:

- Write: `Content-Type: text/plain; charset=UTF-8`
- Read: `Accept: text/plain; charset=UTF-8`

Memcached clients do not require any configuration to handle text-based formats.

This configuration is compatible with...	
REST clients	Yes
Java Hot Rod clients	Yes
Memcached clients	Yes
Non-Java Hot Rod clients	No
Querying and Indexing	No
Custom Java objects	No

5.3. REST, Hot Rod, and Memcached Interoperability with Custom Java Objects

If you store entries in the cache as marshalled, custom Java objects, you should configure the cache with the MediaType of the marshalled storage.

Java Hot Rod clients use the JBoss marshalling storage format as the default to store entries in the cache as custom Java objects.

The following example configures the cache to store entries with the `application/x-jboss-marshalling` media type:

```xml
<distributed-cache name="my-cache">
   <encoding>
      <key media-type="application/x-jboss-marshalling"/>
      <value media-type="application/x-jboss-marshalling"/>
   </encoding>
</distributed-cache>
```

If you use the Protostream marshaller, configure the MediaType as `application/x-protostream`. For UTF8Marshaller, configure the MediaType as `text/plain`.

> If only Hot Rod clients interact with the cache, you do not need to configure the MediaType.

Because REST clients are most suitable for handling text formats, you should use primitives such as `java.lang.String` for keys. Otherwise, REST clients must handle keys as `bytes[]` using a supported binary encoding.

REST clients can read values for cache entries in XML or JSON format. However, the classes must be available in the server.

To read and write data from Memcached clients, you must use `java.lang.String` for keys. Values are stored and returned as marshalled objects.

Some Java Memcached clients allow data transformers that marshall and unmarshall objects. You can also configure the Memcached server module to encode responses in different formats, such as 'JSON' which is language neutral. This allows non-Java clients to interact with the data even if the storage format for the cache is Java-specific.

> Storing Java objects in the cache requires you to deploy entity classes to {ProductName}. See Deploying Entity Classes.

This configuration is compatible with…	
REST clients	Yes
Java Hot Rod clients	Yes
Memcached clients	Yes
Non-Java Hot Rod clients	No
Querying and Indexing	No
Custom Java objects	Yes

5.4. Java and Non-Java Client Interoperability with Protobuf

Storing data in the cache as Protobuf encoded entries provides a platform independent configuration that enables Java and Non-Java clients to access and query the cache from any endpoint.

If indexing is configured for the cache, Infinispan automatically stores keys and values with the `application/x-protostream` media type.

If indexing is not configured for the cache, you can configure it to store entries with the `application/x-protostream` media type as follows:

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

Infinispan converts between `application/x-protostream` and `application/json`, which allows REST clients to read and write JSON formatted data. However REST clients must send the correct headers, as follows:

Read Header

```
Read: Accept: application/json
```

Write Header

```
Write: Content-Type: application/json
```

> ❗ The `application/x-protostream` media type uses Protobuf encoding, which requires you to register a Protocol Buffers schema definition that describes the entities and marshallers that the clients use.

This configuration is compatible with...	
REST clients	Yes
Java Hot Rod clients	Yes
Non-Java Hot Rod clients	Yes
Querying and Indexing	Yes
Custom Java objects	Yes

5.5. Custom Code Interoperability

You can deploy custom code with Infinispan. For example, you can deploy scripts, tasks, listeners, converters, and merge policies. Because your custom code can access data directly in the cache, it must interoperate with clients that access data in the cache through different endpoints.

For example, you might create a remote task to handle custom objects stored in the cache while other clients store data in binary format.

To handle interoperability with custom code you can either convert data on demand or store data as Plain Old Java Objects (POJOs).

5.5.1. Converting Data On Demand

If the cache is configured to store data in a binary format such as `application/x-protostream` or `application/x-jboss-marshalling`, you can configure your deployed code to perform cache operations using Java objects as the media type. See Overriding the MediaType Programmatically.

This approach allows remote clients to use a binary format for storing cache entries, which is optimal. However, you must make entity classes available to the server so that it can convert between binary format and Java objects.

Additionally, if the cache uses Protobuf (`application/x-protostream`) as the binary format, you must deploy protostream marshallers so that {ProductName} can unmarshall data from your custom code.

5.5.2. Storing Data as POJOs

Storing unmarshalled Java objects in the server is not recommended. Doing so requires Infinispan to serialize data when remote clients read from the cache and then deserialize data when remote clients write to the cache.

The following example configures the cache to store entries with the `application/x-java-object` media type:

```
<distributed-cache name="my-cache">
   <encoding>
      <key media-type="application/x-java-object"/>
      <value media-type="application/x-java-object"/>
   </encoding>
</distributed-cache>
```

Hot Rod clients must use a supported marshaller when data is stored as POJOs in the cache, either the JBoss marshaller or the default Java serialization mechanism. You must also deploy the classes must be deployed in the server.

REST clients must use a storage format that Infinispan can convert to and from Java objects, currently JSON or XML.

> Storing Java objects in the cache requires you to deploy entity classes to Infinispan. See Deploying Entity Classes.

Memcached clients must send and receive a serialized version of the stored POJO, which is a JBoss marshalled payload by default. However if you configure the client encoding in the appropriate Memcached connector, you change the storage format so that Memcached clients use a platform neutral format such as JSON.

This configuration is compatible with...	
REST clients	Yes
Java Hot Rod clients	Yes
Non-Java Hot Rod clients	No
Querying and Indexing	Yes. However, querying and indexing works with POJOs only if the entities are annotated.
Custom Java objects	Yes

5.6. Deploying Entity Classes

If you plan to store entries in the cache as custom Java objects or POJOs, you must deploy entity classes to Infinispan. Clients always exchange objects as bytes[]. The entity classes represent those custom objects so that Infinispan can serialize and deserialize them.

To make entity classes available to the server, do the following:

1. Create a JAR file that contains the entities and dependencies.
2. Stop Infinispan if it is running. Infinispan only loads entity classes at boot time.
3. Copy the JAR to the server/lib directory of your Infinispan server installation.

```
├── server
│   ├── lib
│       ├── deployment.my-entities.jar
```

Chapter 6. Marshalling Java Objects

Marshalling converts Java objects into binary format so they can be transferred over the wire or stored to disk. The reverse process, unmarshalling, transforms data from binary format into Java objects.

Infinispan performs marshalling and unmarshalling to:

- Send data to other Infinispan nodes in a cluster.
- Store data in persistent cache stores.
- Store data in binary format to provide lazy deserialization capabilities.

> Infinispan handles marshalling for all internal types. You need to configure marshalling only for the Java objects that you want to store.

Infinispan uses ProtoStream as the default for marshalling Java objects to binary format. Infinispan also provides other Marshaller implementations you can use.

6.1. Using the ProtoStream Marshaller

Infinispan integrates with the ProtoStream API to encode and decode Java objects into Protocol Buffers (Protobuf); a language-neutral, backwards compatible format.

Procedure

1. Create implementations of the ProtoStream `SerializationContextInitializer` interface so that Infinispan can marshall your Java objects.
2. Configure Infinispan to use the implementations.
   - Programmatically:

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
       .addContextInitializers(new LibraryInitializerImpl(), new SCIImpl());
```

   - Declaratively

```
<serialization>
    <context-initializer class="org.infinispan.example.LibraryInitializerImpl"/>
    <context-initializer class="org.infinispan.example.another.SCIImpl"/>
</serialization>
```

Reference

- Creating Serialization Contexts for ProtoStream Marshalling
- Protocol Buffers

6.2. Using JBoss Marshalling

JBoss Marshalling is a serialization-based marshalling library and was the default marshaller in previous Infinispan versions.

> - You should not use serialization-based marshalling with Infinispan. Instead you should use Protostream, which is a high-performance binary wire format that ensures backwards compatibility.
> - JBoss Marshalling and the `AdvancedExternalizer` interface are deprecated and will be removed in a future release. However, Infinispan ignores `AdvancedExternalizer` implementations when persisting data unless you use JBoss Marshalling.

Procedure

1. Add the `infinispan-jboss-marshalling` dependency to your classpath.
2. Configure Infinispan to use the `JBossUserMarshaller`.
   - Programmatically:

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization().marshaller(new JBossUserMarshaller());
```

   - Declaratively:

```
<serialization marshaller=
"org.infinispan.jboss.marshalling.core.JBossUserMarshaller"/>
```

Reference

- Adding Java Classes to Deserialization White Lists
- AdvancedExternalizer

6.3. Using Java Serialization

You can use Java serialization with Infinispan to marshall your objects, but only if your Java objects implement Java's `Serializable` interface.

Procedure

1. Configure Infinispan to use `JavaSerializationMarshaller` as the marshaller.
2. Add your Java classes to the deserialization white list.
   - Programmatically:

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
       .marshaller(new JavaSerializationMarshaller())
       .whiteList()
       .addRegexps("org.infinispan.example.", "
org.infinispan.concrete.SomeClass");
```

- Declaratively:

```
<serialization marshaller=
"org.infinispan.commons.marshall.JavaSerializationMarshaller">
    <white-list>
        <class>org.infinispan.concrete.SomeClass</class>
        <regex>org.infinispan.example.*</regex>
    </white-list>
</serialization>
```

Reference

- Adding Java Classes to Deserialization White Lists
- Serializable
- org.infinispan.commons.marshall.JavaSerializationMarshaller

6.4. Using the Kryo Marshaller

Kryo marshaller is now deprecated and planned for removal. You should replace any Kyro marshaller usage with ProtoStream marshalling.

Infinispan provides a marshalling implementation that uses Kryo libraries.

Prerequisites for Infinispan Servers

To use Kryo marshalling with Infinispan servers, add a JAR that includes the runtime class files for the Kryo marshalling implementation as follows:

1. Download the Kryo Bundle.
2. Add the JAR file to the `server/lib` directory in your Infinispan server installation directory.

Prerequisites for Infinispan Library Mode

To use Kryo marshalling with Infinispan as an embedded library in your application, do the following:

1. Add the `infinispan-marshaller-kryo` dependency to your `pom.xml`.

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

2. Specify the `org.infinispan.marshaller.kryo.KryoMarshaller` class as the marshaller.

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
       .marshaller(new org.infinispan.marshaller.kryo.KryoMarshaller());
```

Procedure

1. Implement a service provider for the `SerializerRegistryService.java` interface.
2. Place all serializer registrations in the `register(Kryo)` method; where serializers are registered with the supplied `Kryo` object using the Kryo API, for example:

```
kryo.register(ExampleObject.class, new ExampleObjectSerializer())
```

3. Specify the full path of implementing classes in your deployment JAR file within:

```
META-INF/services/org/infinispan/marshaller/kryo/SerializerRegistryService
```

Reference

- Kryo on GitHub

6.5. Using the Protostuff Marshaller

Protostuff marshaller is now deprecated and planned for removal. You should replace any Protostuff marshaller usage with ProtoStream marshalling.

Infinispan provides a marshalling implementation that uses Protostuff libraries.

Prerequisites for Infinispan Servers

To use Protostuff marshalling with Infinispan servers, add a JAR that includes the runtime class files for the Protostuff marshalling implementation as follows:

1. Download the Protostuff Bundle JAR.
2. Add the JAR file to the `server/lib` directory in your Infinispan server installation directory.

Prerequisites for Infinispan Library Mode

To use Protostuff marshalling with Infinispan as an embedded library in your application, do the following:

1. Add the `infinispan-marshaller-protostuff` dependency to your `pom.xml`.

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

2. Specify the `org.infinispan.marshaller.protostuff.ProtostuffMarshaller` class as the marshaller.

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
       .marshaller(new org.infinispan.marshaller.protostuff.ProtostuffMarshaller()
);
```

Procedure

Do one of the following to register custom Protostuff schemas for object marshalling:

- Call the `register()` method.

```
RuntimeSchema.register(ExampleObject.class, new ExampleObjectSchema());
```

- Implement a service provider for the `SerializerRegistryService.java` interface that places all schema registrations in the `register()` method.

  You should then specify the full path of implementing classes in your deployment JAR file within:

```
META-INF/services/org/infinispan/marshaller/protostuff/SchemaRegistryService
```

Reference

- Protostuff on GitHub

6.6. Using Custom Marshallers

Infinispan provides a `Marshaller` interface for custom marshallers.

Programmatic procedure

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
      .marshaller(new org.infinispan.example.marshall.CustomMarshaller())
      .whiteList().addRegexp("org.infinispan.example.*");
```

Declarative procedure

```
<serialization marshaller="org.infinispan.example.marshall.CustomMarshaller">
    <white-list>
        <class>org.infinispan.concrete.SomeClass</class>
        <regex>org.infinispan.example.*</regex>
    </white-list>
</serialization>
```

Custom marshaller implementations can access a configured white list via the initialize() method, which is called during startup.

Reference

- org.infinispan.commons.marshall.Marshaller

6.7. Adding Java Classes to Deserialization White Lists

Infinispan does not allow deserialization of arbitrary Java classes for security reasons, which applies to JSON, XML, and marshalled `byte[]` content.

You must add Java classes to a deserialization white list, either using system properties or specifying them in the Infinispan configuration.

System properties

```
// Specify a comma-separated list of fully qualified class names
-Dinfinispan.deserialization.whitelist.classes=java.time.Instant,com.myclass.Entity

// Specify a regular expression to match classes
-Dinfinispan.deserialization.whitelist.regexps=.*
```

Declarative

```
<cache-container>
   <serialization version="1.0" marshaller=
"org.infinispan.marshall.TestObjectStreamMarshaller">
      <white-list>
         <class>org.infinispan.test.data.Person</class>
         <regex>org.infinispan.test.data.*</regex>
       </white-list>
   </serialization>
</cache-container>
```

Java classes that you add to the deserialization whitelist apply to the Infinispan `CacheContainer` and can be deserialized by all caches that the `CacheContainer` controls.

6.8. Storing Deserialized Objects in Infinispan Servers

You can configure Infinispan to use the `application/x-java-object` MediaType as the format for your data. In other words, Infinispan stores your data as Plain Old Java Objects (POJOs) instead of binary content.

If you store POJOs, you must put class files for all custom objects on the Infinispan server classpath.

Procedure

- Add JAR files that contain custom classes and/or service providers for marshaller implementations in the `server/lib` directory.

```
├──── server
│    ├──── lib
│    │    ├──── UserObjects.jar
│        └──── README.txt
```

6.9. Storing Data in Binary Format

Infinispan can store data in its serialized form, in binary format, and then either serialize or deserialize Java objects as needed. This behavior is also referred to as lazy deserialization.

Programmatic procedure

```
ConfigurationBuilder builder = ...
builder.memory().storageType(StorageType.BINARY);
```

Declarative procedure

```
<memory>
   <binary />
</memory>
```

Equality Considerations

When storing data in binary format, Infinispan uses the `WrappedBytes` interface for keys and values. This wrapper class transparently takes care of serialization and deserialization on demand, and internally may have a reference to the object itself being wrapped, or the serialized, byte array representation of the object. This has an effect on the behavior of equality, which is important to note if you implement an `equals()` methods on keys.

The `equals()` method of the wrapper class either compares binary representations (byte arrays) or delegates to the wrapped object instance's `equals()` method, depending on whether both instances being compared are in serialized or deserialized form at the time of comparison. If one of the instances being compared is in one form and the other in another form, then one instance is either serialized or deserialized.

Reference

- org.infinispan.commons.marshall.WrappedBytes.

Chapter 7. Marshalling Custom Java Objects with ProtoStream

Infinispan uses a ProtoStream API to encode and decode Java objects into Protocol Buffers (Protobuf); a language-neutral, backwards compatible format.

- Infinispan ProtoStream library
- Protocol Buffers

7.1. Protobuf Schemas

Protocol Buffers, Protobuf, schemas provide structured representations of your Java objects.

You define Protobuf message types `.proto` schema files as in the following example:

```
package book_sample;

message Book {
    optional string title = 1;
    optional string description = 2;
    optional int32 publicationYear = 3; // no native Date type available in Protobuf

    repeated Author authors = 4;
}

message Author {
    optional string name = 1;
    optional string surname = 2;
}
```

The preceding `.library.proto` file defines an entity (Protobuf message type) named *Book* that is contained in the *book_sample* package. *Book* declares several fields of primitive types and an array (Protobuf repeatable field) named *authors*, which is the *Author* message type.

Protobuf Messages

- You can nest messages but the resulting structure is strictly a tree, never a graph.
- Type inheritance is not possible.
- Collections are not supported but you can emulate arrays with repeated fields.

Reference

- Protocol Buffers Developer Guide

7.2. ProtoStream Serialization Contexts

A ProtoStream `SerializationContext` contains Protobuf type definitions for custom Java objects, loaded from `.proto` schema files, and the accompanying Marshallers for the objects.

The `SerializationContextInitializer` interface registers Java objects and marshallers so that the ProtoStream library can encode your custom objects to Protobuf format, which then enables Infinispan to transmit and store your data.

7.3. ProtoStream Types

ProtoStream can handle the following types, as well as the unboxed equivalents in the case of primitive types, without any additional configuration:

- `String`
- `Integer`
- `Long`
- `Double`
- `Float`
- `Boolean`
- `byte[]`
- `Byte`
- `Short`
- `Character`
- `java.util.Date`
- `java.time.Instant`

To marshall any other Java objects, you must generate, or manually create, `SerializationContextInitializer` implementations that register `.proto` schemas and marshallers with a `SerializationContext`.

7.4. Generating Serialization Context Initializers

Infinispan provides a `protostream-processor` artifact which is a Java annotation processor that can generate `.proto` schemas, marshallers and `SerializationContextInitializer` implementations from annotated Java classes.

Procedure

1. Add the `protostream-processor` dependency to your `pom.xml`.

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

<dependencies>
  <dependency>
    <groupId>org.infinispan.protostream</groupId>
    <artifactId>protostream-processor</artifactId>
    <!--
      This dependency should be declared in the "provided" scope or made "optional"
      because it is a compile-only dependency and is not required at runtime.
      Transitive propagation of this dependency should be also be avoided.
    -->
    <scope>provided</scope>
  </dependency>
</dependencies>
```

2. Annotate the Java objects that you want to marshall with `@ProtoField` and `@ProtoFactory`.

Book.java

```
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;
...

public class Book {
   @ProtoField(number = 1)
   final String title;

   @ProtoField(number = 2)
   final String description;

   @ProtoField(number = 3, defaultValue = "0")
   final int publicationYear;

   @ProtoField(number = 4, collectionImplementation = ArrayList.class)
   final List<Author> authors;

   @ProtoFactory
   Book(String title, String description, int publicationYear, List<Author>
authors) {
      this.title = title;
      this.description = description;
      this.publicationYear = publicationYear;
      this.authors = authors;
   }
   // public Getter methods omitted for brevity
}
```

Author.java

```
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;

public class Author {
   @ProtoField(number = 1)
   final String name;

   @ProtoField(number = 2)
   final String surname;

   @ProtoFactory
   Author(String name, String surname) {
      this.name = name;
      this.surname = surname;
   }
   // public Getter methods omitted for brevity
}
```

3. Define an interface that extends `SerializationContextInitializer` and is annotated with `@AutoProtoSchemaBuilder`.

```
@AutoProtoSchemaBuilder(
      includeClasses = {
            Book.class,
            Author.class,
      },
      schemaFileName = "library.proto", ①
      schemaFilePath = "proto/", ②
      schemaPackageName = "book_sample")
interface LibraryInitializer extends SerializationContextInitializer {
}
```

① names the generated `.proto` schema file.

② sets the path under `target/classes` where the schema file is generated.

During compile-time, `protostream-processor` generates a concrete implementation of the interface that you can use to initialize a ProtoStream `SerializationContext`. By default, implementation names are the annotated class name with an "Impl" suffix.

Examples

The following are examples of a generated schema file and implementation:

target/classes/proto/library.proto

```
// File name: library.proto
// Generated from : org.infinispan.commons.marshall.LibraryInitializer

syntax = "proto2";

package book_sample;



message Book {

    optional string title = 1;

    optional string description = 2;

    optional int32 publicationYear = 3 [default = 0];

    repeated Author authors = 4;
}


message Author {

    optional string name = 1;

    optional string surname = 2;
}
```

LibraryInitializerImpl.java

```
/*
 Generated by
org.infinispan.protostream.annotations.impl.processor.AutoProtoSchemaBuilderAnnotation
Processor
 for class org.infinispan.commons.marshall.LibraryInitializer
 annotated with
@org.infinispan.protostream.annotations.AutoProtoSchemaBuilder(dependsOn=,
service=false, autoImportClasses=false, excludeClasses=,
includeClasses=org.infinispan.commons.marshall.Book,org.infinispan.commons.marshall.Au
thor, basePackages={}, value={}, schemaPackageName="book_sample",
schemaFilePath="proto/", schemaFileName="library.proto", className="")
 */

package org.infinispan.commons.marshall;

/**
 * WARNING: Generated code!
 */
```

```
@javax.annotation.Generated(value =
"org.infinispan.protostream.annotations.impl.processor.AutoProtoSchemaBuilderAnnotatio
nProcessor",
      comments = "Please do not edit this file!")
@org.infinispan.protostream.annotations.impl.OriginatingClasses({
      "org.infinispan.commons.marshall.Author",
      "org.infinispan.commons.marshall.Book"
})
/*@org.infinispan.protostream.annotations.AutoProtoSchemaBuilder(
   className = "LibraryInitializerImpl",
   schemaFileName = "library.proto",
   schemaFilePath = "proto/",
   schemaPackageName = "book_sample",
   service = false,
   autoImportClasses = false,
   classes = {
      org.infinispan.commons.marshall.Author.class,
      org.infinispan.commons.marshall.Book.class
   }
)*/
public class LibraryInitializerImpl implements org.infinispan.commons.marshall
.LibraryInitializer {

   @Override
   public String getProtoFileName() { return "library.proto"; }

   @Override
   public String getProtoFile() { return org.infinispan.protostream
.FileDescriptorSource.getResourceAsString(getClass(), "/proto/library.proto"); }

   @Override
   public void registerSchema(org.infinispan.protostream.SerializationContext serCtx)
{
      serCtx.registerProtoFiles(org.infinispan.protostream.FileDescriptorSource
.fromString(getProtoFileName(), getProtoFile()));
   }

   @Override
   public void registerMarshallers(org.infinispan.protostream.SerializationContext
serCtx) {
      serCtx.registerMarshaller(new org.infinispan.commons.marshall.Book
$___Marshaller_cdc76a682a43643e6e1d7e43ba6d1ef6f794949a45e1a8bc961046cda44c9a85());
      serCtx.registerMarshaller(new org.infinispan.commons.marshall.Author
$___Marshaller_9b67e1c1ecea213b4207541b411fb9af2ae6f658610d2a4ca9126484d57786d1());
   }
}
```

7.5. Manually Implementing Serialization Context Initializers

In some cases you might need to manually define `.proto` schema files and implement ProtoStream marshallers. For example, if you cannot modify Java object classes to add annotations.

Procedure

1. Create a `.proto` schema with Protobuf messages.

```
package book_sample;

message Book {
    optional string title = 1;
    optional string description = 2;
    optional int32 publicationYear = 3; // no native Date type available in
Protobuf

    repeated Author authors = 4;
}

message Author {
    optional string name = 1;
    optional string surname = 2;
}
```

2. Use the `org.infinispan.protostream.MessageMarshaller` interface to implement marshallers for your classes.

BookMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;

public class BookMarshaller implements MessageMarshaller<Book> {

   @Override
   public String getTypeName() {
      return "book_sample.Book";
   }

   @Override
   public Class<? extends Book> getJavaClass() {
      return Book.class;
   }

   @Override
   public void writeTo(MessageMarshaller.ProtoStreamWriter writer, Book book)
throws IOException {
      writer.writeString("title", book.getTitle());
      writer.writeString("description", book.getDescription());
      writer.writeInt("publicationYear", book.getPublicationYear());
      writer.writeCollection("authors", book.getAuthors(), Author.class);
   }

   @Override
   public Book readFrom(MessageMarshaller.ProtoStreamReader reader) throws
IOException {
      String title = reader.readString("title");
      String description = reader.readString("description");
      int publicationYear = reader.readInt("publicationYear");
      List<Author> authors = reader.readCollection("authors", new ArrayList<>(),
Author.class);
      return new Book(title, description, publicationYear, authors);
   }
}
```

AuthorMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;

public class AuthorMarshaller implements MessageMarshaller<Author> {

   @Override
   public String getTypeName() {
      return "book_sample.Author";
   }

   @Override
   public Class<? extends Author> getJavaClass() {
      return Author.class;
   }

   @Override
   public void writeTo(MessageMarshaller.ProtoStreamWriter writer, Author author)
throws IOException {
      writer.writeString("name", author.getName());
      writer.writeString("surname", author.getSurname());
   }

   @Override
   public Author readFrom(MessageMarshaller.ProtoStreamReader reader) throws
IOException {
      String name = reader.readString("name");
      String surname = reader.readString("surname");
      return new Author(name, surname);
   }
}
```

3. Create a `SerializationContextInitializer` implementation that registers the `.proto` schema and the ProtoStream marshaller implementations with a `SerializationContext`.

ManualSerializationContextInitializer.java

```
import org.infinispan.protostream.FileDescriptorSource;
import org.infinispan.protostream.SerializationContext;
import org.infinispan.protostream.SerializationContextInitializer;
...

public class ManualSerializationContextInitializer implements
SerializationContextInitializer {
   @Override
   public String getProtoFileName() {
      return "library.proto";
   }

   @Override
   public String getProtoFile() throws UncheckedIOException {
      // Assumes that the file is located in a Jar's resources, we must provide the
path to the library.proto file
      return FileDescriptorSource.getResourceAsString(getClass(), "/" +
getProtoFileName());
   }

   @Override
   public void registerSchema(SerializationContext serCtx) {
      serCtx.registerProtoFiles(FileDescriptorSource.fromString(getProtoFileName(),
getProtoFile()));
   }

   @Override
   public void registerMarshallers(SerializationContext serCtx) {
      serCtx.registerMarshaller(new AuthorMarshaller());
      serCtx.registerMarshaller(new BookMarshaller());
   }
}
```

Chapter 8. Clustered Locks

A *clustered lock* is a lock which is distributed and shared among all nodes in the Infinispan cluster and currently provides a way to execute code that will be synchronized between the nodes in a given cluster.

8.1. Installation

In order to start using the clustered locks, you needs to add the dependency in your Maven `pom.xml` file:

pom.xml

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

8.2. ClusteredLock Configuration

Currently there is a single type of `ClusteredLock` supported : non reentrant, NODE ownership lock.

8.2.1. Ownership

- `NODE` When a `ClusteredLock` is defined, this lock can be used from all the nodes in the Infinispan cluster. When the ownership is NODE type, this means that the owner of the lock is the Infinispan node that acquired the lock at a given time. This means that each time we get a `ClusteredLock` instance with the `ClusteredCacheManager`, this instance will be the same instance for each Infinispan node. This lock can be used to synchronize code between Infinispan nodes. The advantage of this lock is that any thread in the node can release the lock at a given time.
- `INSTANCE` - not yet supported

When a `ClusteredLock` is defined, this lock can be used from all the nodes in the Infinispan cluster. When the ownership is INSTANCE type, this means that the owner of the lock is the actual instance we acquired when `ClusteredLockManager.get("lockName")` is called.

This means that each time we get a `ClusteredLock` instance with the `ClusteredCacheManager`, this instance will be a new instance. This lock can be used to synchronize code between Infinispan nodes and inside each Infinispan node. The advantage of this lock is that only the instance that called 'lock' can release the lock.

8.2.2. Reentrancy

When a `ClusteredLock` is configured reentrant, the owner of the lock can reacquire the lock as many consecutive times as it wants while holding the lock.

Currently, only non reentrant locks are supported. This means that when two consecutive `lock` calls are sent for the same owner, the first call will acquire the lock if it's available, and the second call will block.

8.3. ClusteredLockManager Interface

The `ClusteredLockManager` interface, **marked as experimental**, is the entry point to define, retrieve and remove a lock. It automatically listen to the creation of `EmbeddedCacheManager` and proceeds with the registration of an instance of it per `EmbeddedCacheManager`. It starts the internal caches needed to store the lock state.

Retrieving the `ClusteredLockManager` is as simple as invoking the `EmbeddedClusteredLockManagerFactory.from(EmbeddedCacheManager)` as shown in the example below:

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

// retrieve the ClusteredLockManager
ClusteredLockManager clusteredLockManager = EmbeddedClusteredLockManagerFactory.from
(manager);
```

```
@Experimental
public interface ClusteredLockManager {

   boolean defineLock(String name);

   boolean defineLock(String name, ClusteredLockConfiguration configuration);

   ClusteredLock get(String name);

   ClusteredLockConfiguration getConfiguration(String name);

   boolean isDefined(String name);

   CompletableFuture<Boolean> remove(String name);

   CompletableFuture<Boolean> forceRelease(String name);
}
```

- `defineLock` : Defines a lock with the specified name and the default `ClusteredLockConfiguration`. It does not overwrite existing configurations.
- `defineLock(String name, ClusteredLockConfiguration configuration)` : Defines a lock with the specified name and `ClusteredLockConfiguration`. It does not overwrite existing configurations.
- `ClusteredLock get(String name)` : Get's a `ClusteredLock` by it's name. A call of `defineLock` must be done at least once in the cluster. See ownership level section to understand the implications of `get` method call.

Currently, the only ownership level supported is **NODE**.

- `ClusteredLockConfiguration getConfiguration(String name)` :

Returns the configuration of a `ClusteredLock`, if such exists.

- `boolean isDefined(String name)` : Checks if a lock is already defined.
- `CompletableFuture<Boolean> remove(String name)` : Removes a `ClusteredLock` if such exists.
- `CompletableFuture<Boolean> forceRelease(String name)` : Releases - or unlocks - a `ClusteredLock`, if such exists, no matter who is holding it at a given time. Calling this method may cause concurrency issues and has to be used in **exceptional situations**.

8.4. ClusteredLock Interface

`ClusteredLock` interface, **marked as experimental**, is the interface that implements the clustered locks.

```
@Experimental
public interface ClusteredLock {

   CompletableFuture<Void> lock();

   CompletableFuture<Boolean> tryLock();

   CompletableFuture<Boolean> tryLock(long time, TimeUnit unit);

   CompletableFuture<Void> unlock();

   CompletableFuture<Boolean> isLocked();

   CompletableFuture<Boolean> isLockedByMe();
}
```

- `lock` : Acquires the lock. If the lock is not available then call blocks until the lock is acquired. Currently, **there is no maximum time specified for a lock request to fail**, so this could cause thread starvation.
- `tryLock` Acquires the lock only if it is free at the time of invocation, and returns `true` in that case. This method does not block (or wait) for any lock acquisition.
- `tryLock(long time, TimeUnit unit)` If the lock is available this method returns immediately with `true`. If the lock is not available then the call waits until :
  - The lock is acquired
  - The specified waiting time elapses

If the time is less than or equal to zero, the method will not wait at all.

- `unlock`

Releases the lock. Only the holder of the lock may release the lock.

- `isLocked` Returns `true` when the lock is locked and `false` when the lock is released.
- `isLockedByMe` Returns `true` when the lock is owned by the caller and `false` when the lock is owned by someone else or it's released.

8.4.1. Usage Examples

```
EmbeddedCache cm = ...;
ClusteredLockManager cclm = EmbeddedClusteredLockManagerFactory.from(cm);

lock.tryLock()
  .thenCompose(result -> {
     if (result) {
      try {
          // manipulate protected state
          } finally {
             return lock.unlock();
          }
     } else {
        // Do something else
     }
  });
}
```

8.4.2. ClusteredLockManager Configuration

You can configure `ClusteredLockManager` to use different strategies for locks, either declaratively or programmatically, with the following attributes:

`num-owners`
: Defines the total number of nodes in each cluster that store the states of clustered locks. The default value is `-1`, which replicates the value to all nodes.

`reliability`
: Controls how clustered locks behave when clusters split into partitions or multiple nodes leave a cluster. You can set the following values:

  - `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.

The following is an example declarative configuration for `ClusteredLockManager`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
        xmlns="urn:infinispan:config:12.0">
    ...
    <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:12.0"
                         num-owners = "3"
                         reliability="AVAILABLE">
            <clustered-lock name="lock1" />
            <clustered-lock name="lock2" />
        </clustered-locks>
    </cache-container>
    ...
</infinispan>
```

Chapter 9. Clustered Counters

Clustered counters are counters which are distributed and shared among all nodes in the Infinispan 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.

9.1. Installation and Configuration

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

pom.xml

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

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

configuration.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan>
    <cache-container ...>
        <!-- if needed to persist counter, global state needs to be configured -->
        <global-state>
            ...
        </global-state>
        <!-- your caches configuration goes here -->
         <counters xmlns="urn:infinispan:config:counters:12.0" 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 value="0"/>
             </strong-counter>
             <strong-counter name="c3" initial-value="3" storage="PERSISTENT">
                 <upper-bound value="5"/>
             </strong-counter>
             <strong-counter name="c4" initial-value="4" storage="VOLATILE">
                 <lower-bound value="0"/>
                 <upper-bound value="10"/>
             </strong-counter>
             <weak-counter name="c5" initial-value="5" storage="PERSISTENT"
concurrency-level="1"/>
         </counters>
    </cache-container>
</infinispan>
```

or programmatically, in the `GlobalConfigurationBuilder`:

```
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.VO
LATILE);
builder.addStrongCounter().name("c3").initialValue(3).upperBound(5).storage(Storage.PE
RSISTENT);
builder.addStrongCounter().name("c4").initialValue(4).lowerBound(0).upperBound(10).sto
rage(Storage.VOLATILE);
builder.addWeakCounter().name("c5").initialValue(5).concurrencyLevel(1).storage(Storag
e.PERSISTENT);
```

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).storage(Storage.VOLATILE).build());
manager.defineCounter("c3", CounterConfiguration.builder(CounterType.BOUNDED_STRONG)
.initialValue(3).upperBound(5).storage(Storage.PERSISTENT).build());
manager.defineCounter("c4", CounterConfiguration.builder(CounterType.BOUNDED_STRONG)
.initialValue(4).lowerBound(0).upperBound(10).storage(Storage.VOLATILE).build());
manager.defineCounter("c2", CounterConfiguration.builder(CounterType.WEAK)
.initialValue(5).concurrencyLevel(1).storage(Storage.PERSISTENT).build());
```

`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", ...);
}
```

Per cluster attributes:

- `num-owners`: Sets the number of counter's copies to keep cluster-wide. A smaller number will make update operations faster but will support a lower number of server crashes. It **must be positive** and its default value is `2`.
- `reliability`: Sets the counter's update behavior in a network partition. Default value is `AVAILABLE` and valid values are:
  - `AVAILABLE`: all partitions are able to read and update the counter's value.
  - `CONSISTENT`: only the primary partition (majority of nodes) will be able to read and update the counter's value. The remaining partitions can only read its value.

Per counter attributes:

- `initial-value` [common]: Sets the counter's initial value. Default is `0` (zero).
- `storage` [common]: Sets the counter's behavior when the cluster is shutdown and restarted. Default value is `VOLATILE` and valid values are:
  - `VOLATILE`: the counter's value is only available in memory. The value will be lost when a cluster is shutdown.
  - `PERSISTENT`: the counter's value is stored in a private and local persistent store. The value is kept when the cluster is shutdown and restored after a restart.

> On-demand and `VOLATILE` counters will lose its value and configuration after a cluster shutdown. They must be defined again after the restart.

* `lower-bound` [strong]: Sets the strong consistent counter's lower bound. Default value is `Long.MIN_VALUE`.
* `upper-bound` [strong]: Sets the strong consistent counter's upper bound. Default value is `Long.MAX_VALUE`.

> If neither the `lower-bound` or `upper-bound` are configured, the strong counter is set as unbounded.

> The `initial-value` must be between `lower-bound` and `upper-bound` inclusive.

* `concurrency-level` [weak]: Sets the number of concurrent updates. Its value **must be positive** and the default value is `16`.

9.1.1. List counter names

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

9.2. The `CounterManager` interface.

The `CounterManager` interface is the entry point to define, retrieve and remove a counter. It 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);
```

For Hot Rod client, the `CounterManager` is registered in the RemoteCacheManager and it can be retrieved like:

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

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

9.2.1. Remove a counter via CounterManager

> use with caution.

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.

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

9.3. 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.

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

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

The strong counter provides uses a single key stored in Infinispan 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");
```

> 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.

> A operation is considered completed when the `CompletableFuture` is completed.

> 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.

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();
```

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)

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:

```
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()`)

```java
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);
```

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

The `WeakCounter` stores the counter's value in multiple keys in Infinispan 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.

> 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:

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

Weak Counter Interface

The `WeakCounter` adds the following methods:

```java
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.

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.

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());
```

9.4. 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();
}
```

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.

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

Chapter 10. Locking and Concurrency

Infinispan 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

10.1. Locking implementation details

Infinispan'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, Infinispan'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.

10.1.1. How does it work in clustered caches?

In clustered caches, each key has a node responsible to lock the key. This node is called primary owner.

Non Transactional caches

1. The write operation is sent to the primary owner of the key.
2. The primary owner tries to lock the key.
   a. If it succeeds, it forwards the operation to the other owners;
   b. Otherwise, an exception is thrown.

> If the operation is conditional and it fails on the primary owner, it is not forwarded to the other owners.

> If the operation is executed locally in the primary owner, the first step is skipped.

10.1.2. Transactional caches

The transactional cache supports optimistic and pessimistic locking mode. Refer to Transaction Locking for more information.

10.1.3. Isolation levels

Isolation level affects what transactions can read when running concurrently with other transaction. Refer to Isolation Levels for more information.

10.1.4. 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.

10.1.5. 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);
```

10.1.6. 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 Infinispan.

Configuration example:

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

Or

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

10.1.7. 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);
```

10.1.8. 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.

10.2. Data Versioning

Infinispan 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 Infinispan, such as when using Infinispan 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.

Write skew checks cannot and will not be performed in the case of external data versioning.

Chapter 11. Using the Infinispan CDI Extension

Infinispan 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.

11.1. CDI Dependencies

Update your `pom.xml` with one of the following dependencies to include the Infinispan CDI extension in your project:

Embedded (Library) Mode

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

Server Mode

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

11.2. Injecting Embedded Caches

Set up CDI beans to inject embedded caches.

Procedure

1. Create a cache qualifier annotation.

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

@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GreetingCache { ①
}
```

① 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.

> 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;
    }
}
```

11.3. 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;

    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.

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

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

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

public class Config {

    @RemoteGreetingCache
    @Produces
    @ApplicationScoped ①
    public ConfigurationBuilder builder = new ConfigurationBuilder(); ②
        builder.addServer().host("localhost").port(11222);
        return new RemoteCacheManager(builder.build());
    }
}
```

① 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.

② creates a new `RemoteCacheManager` instance that is bound to the `@RemoteGreetingCache` qualifier.

11.4. 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.

> **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 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 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")
```

11.5. 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 12. Infinispan Transactions

Infinispan 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 Infinispan 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`.

Infinispan 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 Infinispan 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 Infinispan 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();
```

12.1. 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 Infinispan and how locks are being acquired see the section below. More details about the configuration settings are available in Configuration reference.

12.2. Isolation levels

Infinispan 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 Infinispan. 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()
```

12.3. Transaction locking

12.3.1. 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).

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

12.3.2. 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.
```

> For conditional commands, the validation still happens on the originator.

12.3.3. 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.

12.4. 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.

Infinispan automatically performs write skew checks to ensure data consistency for REPEATABLE_READ isolation levels in optimistic transactions. This allows Infinispan 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.

12.4.1. 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.

- 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
```

12.5. 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.

12.6. Enlisting Synchronizations

By default Infinispan registers itself as a first class participant in distributed transactions through

XAResource. There are situations where Infinispan 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 Infinispan is used as a 2nd level cache in Hibernate.

Infinispan allows transaction enlistment through Synchronization. To enable it just use `NON_XA` transaction mode.

`Synchronization`s 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 Infinispan 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 Infinispan as a `Synchronization` makes the `TransactionManager` skip writing the log to the disk (performance improvement).

12.7. 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.

> Generally speaking, one should use batching API whenever the only participant in the transaction is an Infinispan 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 Infinispan, then batching can be used. If one account is in a database and the other is Infinispan, then distributed transactions are required.

> You *do not* have to have a transaction manager defined to use batching.

12.7.1. 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
```

12.7.2. 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.

12.8. 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.

12.8.1. 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 Infinispan. When `TransactionManager.commit()` is invoked, both resources prepare successfully (1st phase). During the commit (2nd) phase, the database successfully applies the changes whilst Infinispan 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 Infinispan (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 Infinispan ends up in a consistent state.

12.8.2. How does it work

Recovery is coordinated by the transaction manager. The transaction manager works with Infinispan 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 Infinispan cluster and replay the commit of transactions or force the rollback. Infinispan provides JMX tooling for this - this is explained extensively in the Transaction recovery and reconciliation section.

12.8.3. Configuring recovery

Recovery is *not* enabled by default in Infinispan. If disabled, the `TransactionManager` won't be able to work with Infinispan 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.

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

Enable JMX support

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

12.8.4. Recovery cache

In order to track in-doubt transactions and be able to reply them, Infinispan 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 Infinispan will configure a local cache for you.

It is possible (though not mandated) to share same recovery cache between all the Infinispan 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.

12.8.5. 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 Infinispan `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.

12.8.6. 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 Infinispan server through JMX, and lists the in doubt transactions. The image below demonstrates JConsole connecting to an Infinispan node that has an in doubt transaction.

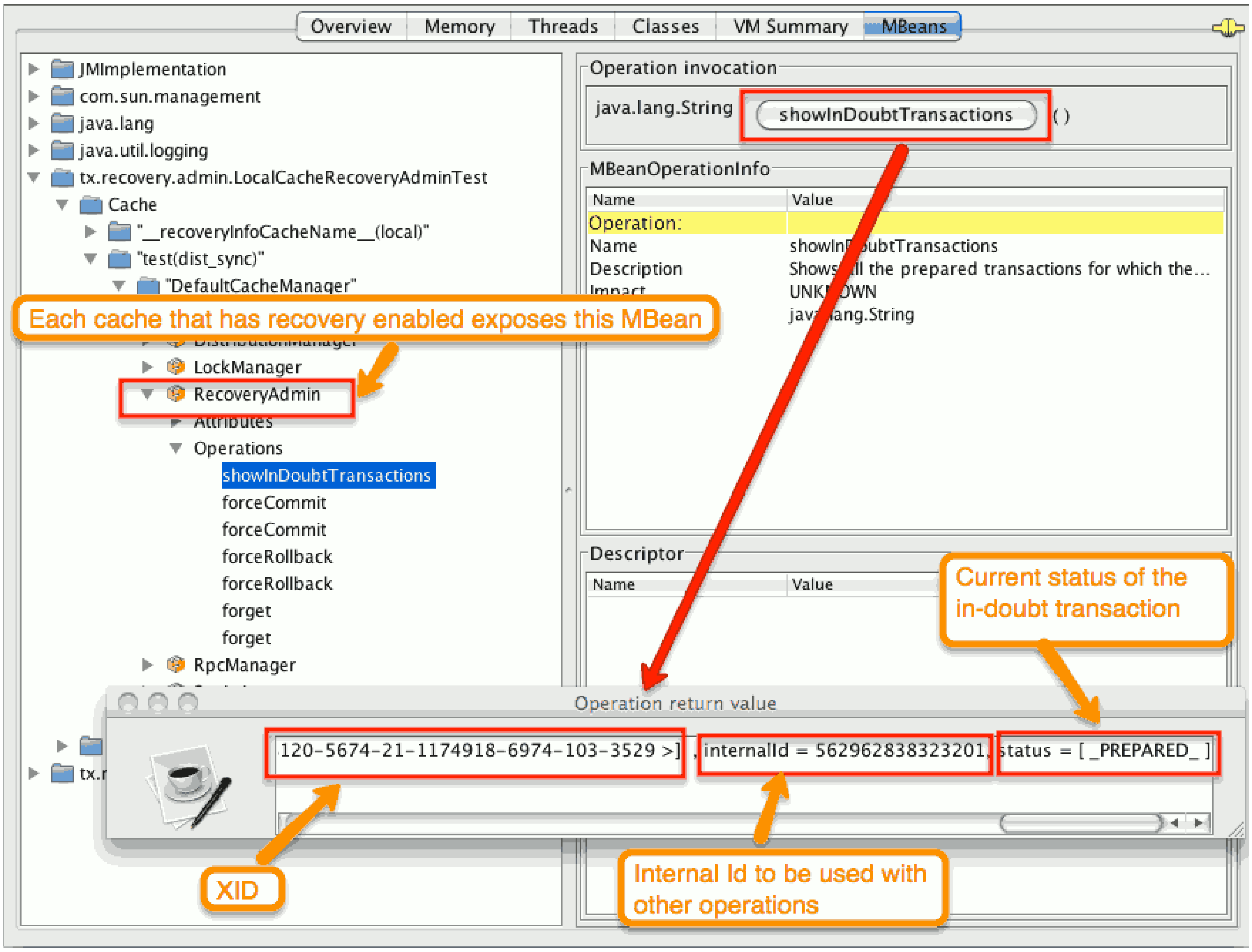

Figure 1. Show in-doubt transactions

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 Infinispan 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 Infinispan’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.

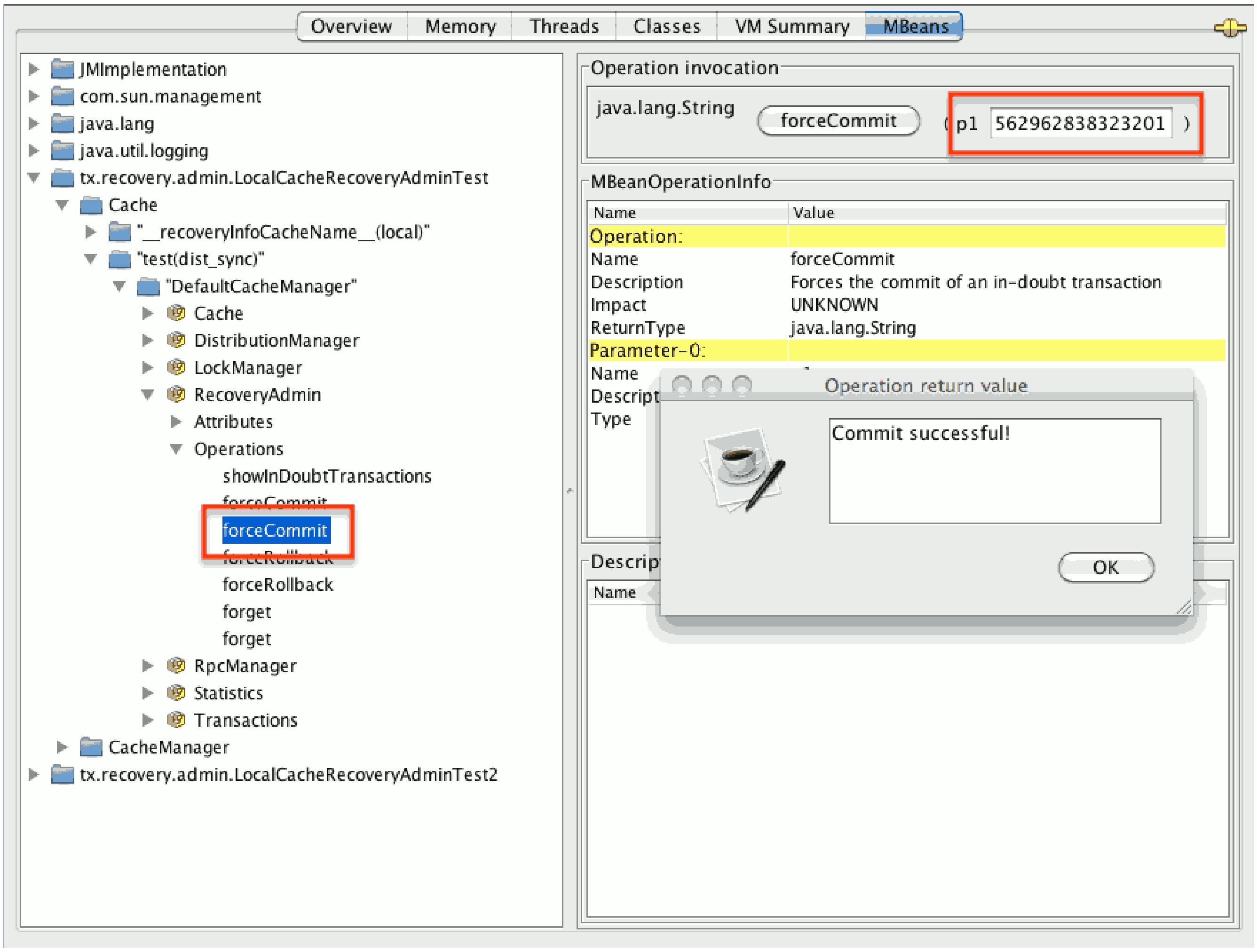

Figure 2. Force commit

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

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.

12.8.7. Want to know more?

The recovery design document describes in more detail the insides of transaction recovery implementation.

Chapter 13. Functional Map API

Infinispan provides an experimental API for interacting with your data which takes advantage of the functional programming additions and improved asynchronous programming capabilities available in Java 8.

Infinispan's Functional Map API is a distilled map-like asynchronous API which uses functions to interact with data.

13.1. Asynchronous and Lazy

Being an asynchronous API, all methods that return a single result, return a CompletableFuture which wraps the result, so you can use the resources of your system more efficiently by having the possibility to receive callbacks when the `CompletableFuture` has completed, or you can chain or compose them with other CompletableFuture.

For those operations that return multiple results, the API returns instances of a Traversable interface which offers a lazy pull-style API for working with multiple results.

`Traversable`, being a lazy pull-style API, can still be asynchronous underneath since the user can decide to work on the traversable at a later stage, and the `Traversable` implementation itself can decide when to compute those results.

13.2. Function transparency

Since the content of the functions is transparent to Infinispan, the API has been split into 3 interfaces for read-only (`ReadOnlyMap`), read-write (`ReadWriteMap`) and write-only (`WriteOnlyMap`) operations respectively, in order to provide hints to the Infinispan internals on the type of work needed to support functions.

13.3. Constructing Functional Maps

To construct any of the read-only, write-only or read-write map instances, an Infinispan `AdvancedCache` is required, which is retrieved from the Cache Manager, and using the `AdvancedCache` , static method factory methods are used to create `ReadOnlyMap` , `ReadWriteMap` or `WriteOnlyMap`

```
import org.infinispan.functional.FunctionalMap.*;
import org.infinispan.functional.impl.*;
import org.infinispan.AdvancedCache;

AdvancedCache<String, String> cache = ...

FunctionalMapImpl<String, String> functionalMap = FunctionalMapImpl.create(cache);
ReadOnlyMap<String, String> readOnlyMap = ReadOnlyMapImpl.create(functionalMap);
WriteOnlyMap<String, String> writeOnlyMap = WriteOnlyMapImpl.create(functionalMap);
ReadWriteMap<String, String> readWriteMap = ReadWriteMapImpl.create(functionalMap);
```

At this stage, the Functional Map API is experimental and hence the way FunctionalMap, ReadOnlyMap, WriteOnlyMap and ReadWriteMap are constructed is temporary.

13.4. Read-Only Map API

Read-only operations have the advantage that no locks are acquired for the duration of the operation. Here's an example on how to the equivalent operation for Map.get(K):

```
import org.infinispan.functional.EntryView.ReadEntryView;
import org.infinispan.functional.FunctionalMap.ReadOnlyMap;

ReadOnlyMap<String, String> readOnlyMap = ...
CompletableFuture<Optional<String>> readFuture = readOnlyMap.eval("key1",
ReadEntryView::find);
readFuture.thenAccept(System.out::println);
```

Read-only map also exposes operations to retrieve multiple keys in one go:

```
import org.infinispan.functional.EntryView.*;
import org.infinispan.functional.FunctionalMap.*;
import org.infinispan.functional.Traversable;

ReadOnlyMap<String, String> readOnlyMap = ...

Set<String> keys = new HashSet<>(Arrays.asList("key1", "key2"));
Traversable<String> values = readOnlyMap.evalMany(keys, ReadEntryView::get);
values.forEach(System.out::println);
```

Finally, read-only map also exposes methods to read all existing keys as well as entries, which include both key and value information.

13.4.1. Read-Only Entry View

The function parameters for read-only maps provide the user with a read-only entry view to interact with the data in the cache, which include these operations:

- `key()` method returns the key for which this function is being executed.
- `find()` returns a Java 8 `Optional` wrapping the value if present, otherwise it returns an empty optional. Unless the value is guaranteed to be associated with the key, it's recommended to use `find()` to verify whether there's a value associated with the key.
- `get()` returns the value associated with the key. If the key has no value associated with it, calling `get()` throws a `NoSuchElementException`. `get()` can be considered as a shortcut of `ReadEntryView.find().get()` which should be used only when the caller has guarantees that there's definitely a value associated with the key.

- `findMetaParam(Class<T> type)` allows metadata parameter information associated with the cache entry to be looked up, for example: entry lifespan, last accessed time…etc. See Metadata Parameter Handling to find out more.

13.5. Write-Only Map API

Write-only operations include operations that insert or update data in the cache and also removals. Crucially, a write-only operation does not attempt to read any previous value associated with the key. This is an important optimization since that means neither the cluster nor any persistence stores will be looked up to retrieve previous values. In the main Infinispan Cache, this kind of optimization was achieved using a local-only per-invocation flag, but the use case is so common that in this new functional API, this optimization is provided as a first-class citizen.

Using write-only map API , an operation equivalent to `javax.cache.Cache` (`JCache`) 's void returning `put` can be achieved this way, followed by an attempt to read the stored value using the read-only map API:

```
import org.infinispan.functional.EntryView.*;
import org.infinispan.functional.FunctionalMap.*;

WriteOnlyMap<String, String> writeOnlyMap = ...
ReadOnlyMap<String, String> readOnlyMap = ...

CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", "value1",
   (v, view) -> view.set(v));
CompletableFuture<String> readFuture = writeFuture.thenCompose(r ->
   readOnlyMap.eval("key1", ReadEntryView::get));
readFuture.thenAccept(System.out::println);
```

Multiple key/value pairs can be stored in one go using `evalMany` API:

```
WriteOnlyMap<String, String> writeOnlyMap = ...

Map<K, String> data = new HashMap<>();
data.put("key1", "value1");
data.put("key2", "value2");
CompletableFuture<Void> writerAllFuture = writeOnlyMap.evalMany(data, (v, view) ->
view.set(v));
writerAllFuture.thenAccept(x -> "Write completed");
```

To remove all contents of the cache, there are two possibilities with different semantics. If using `evalAll` each cached entry is iterated over and the function is called with that entry's information. Using this method also results in listeners being invoked.

```
WriteOnlyMap<String, String> writeOnlyMap = ...

CompletableFuture<Void> removeAllFuture = writeOnlyMap.evalAll(WriteEntryView::remove
);
removeAllFuture.thenAccept(x -> "All entries removed");
```

The alternative way to remove all entries is to call `truncate` operation which clears the entire cache contents in one go without invoking any listeners and is best-effort:

```
WriteOnlyMap<String, String> writeOnlyMap = ...

CompletableFuture<Void> truncateFuture = writeOnlyMap.truncate();
truncateFuture.thenAccept(x -> "Cache contents cleared");
```

13.5.1. Write-Only Entry View

The function parameters for write-only maps provide the user with a write-only entry view to modify the data in the cache, which include these operations:

- `set(V, MetaParam.Writable…)` method allows for a new value to be associated with the cache entry for which this function is executed, and it optionally takes zero or more metadata parameters to be stored along with the value. See Metadata Parameter Handling for more information.
- `remove()` method removes the cache entry, including both value and metadata parameters associated with this key.

13.6. Read-Write Map API

The final type of operations we have are readwrite operations, and within this category CAS-like (CompareAndSwap) operations can be found. This type of operations require previous value associated with the key to be read and for locks to be acquired before executing the function. The vast majority of operations within `ConcurrentMap` and `JCache` APIs fall within this category, and they can easily be implemented using the read-write map API . Moreover, with read-write map API , you can make CASlike comparisons not only based on value equality but based on metadata parameter equality such as version information, and you can send back previous value or boolean instances to signal whether the CASlike comparison succeeded.

Implementing a write operation that returns the previous value associated with the cache entry is easy to achieve with the read-write map API:

```
import org.infinispan.functional.EntryView.*;
import org.infinispan.functional.FunctionalMap.*;

ReadWriteMap<String, String> readWriteMap = ...

CompletableFuture<Optional<String>> readWriteFuture = readWriteMap.eval("key1",
"value1",
   (v, view) -> {
      Optional<V> prev = rw.find();
      view.set(v);
      return prev;
   });
readWriteFuture.thenAccept(System.out::println);
```

`ConcurrentMap.replace(K, V, V)` is a replace function that compares the value present in the map and if it's equals to the value passed in as first parameter, the second value is stored, returning a boolean indicating whether the replace was successfully completed. This operation can easily be implemented using the read-write map API:

```
ReadWriteMap<String, String> readWriteMap = ...

String oldValue = "old-value";
CompletableFuture<Boolean> replaceFuture = readWriteMap.eval("key1", "value1", (v,
view) -> {
   return view.find().map(prev -> {
      if (prev.equals(oldValue)) {
         rw.set(v);
         return true; // previous value present and equals to the expected one
      }
      return false; // previous value associated with key does not match
   }).orElse(false); // no value associated with this key
});
replaceFuture.thenAccept(replaced -> System.out.printf("Value was replaced? %s%n",
replaced));
```

The function in the example above captures `oldValue` which is an external value to the function which is valid use case.

Read-write map API contains `evalMany` and `evalAll` operations which behave similar to the write-only map offerings, except that they enable previous value and metadata parameters to be read.

13.6.1. Read-Write Entry View

The function parameters for read-write maps provide the user with the possibility to query the information associated with the key, including value and metadata parameters, and the user can also use this read-write entry view to modify the data in the cache.

The operations are exposed by read-write entry views are a union of the operations exposed by read-only entry views and write-only entry views.

13.7. Metadata Parameter Handling

Metadata parameters provide extra information about the cache entry, such as version information, lifespan, last accessed/used time…etc. Some of these can be provided by the user, e.g. version, lifespan…etc, but some others are computed internally and can only be queried, e.g. last accessed/used time.

The functional map API provides a flexible way to store metadata parameters along with an cache entry. To be able to store a metadata parameter, it must extend `MetaParam.Writable` interface, and implement the methods to allow the internal logic to extra the data. Storing is done via the `set(V, MetaParam.Writable…)` method in the write-only entry view or read-write entry view function parameters.

Querying metadata parameters is available via the `findMetaParam(Class)` method available via read-write entry view or read-only entry views or function parameters.

Here is an example showing how to store metadata parameters and how to query them:

```
import java.time.Duration;
import org.infinispan.functional.EntryView.*;
import org.infinispan.functional.FunctionalMap.*;
import org.infinispan.functional.MetaParam.*;

WriteOnlyMap<String, String> writeOnlyMap = ...
ReadOnlyMap<String, String> readOnlyMap = ...

CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", "value1",
   (v, view) -> view.set(v, new MetaLifespan(Duration.ofHours(1).toMillis())));
CompletableFuture<MetaLifespan> readFuture = writeFuture.thenCompose(r ->
   readOnlyMap.eval("key1", view -> view.findMetaParam(MetaLifespan.class).get()));
readFuture.thenAccept(System.out::println);
```

If the metadata parameter is generic, for example `MetaEntryVersion<T>` , retrieving the metadata parameter along with a specific type can be tricky if using `.class` static helper in a class because it does not return a `Class<T>` but only `Class`, and hence any generic information in the class is lost:

```
ReadOnlyMap<String, String> readOnlyMap = ...

CompletableFuture<String> readFuture = readOnlyMap.eval("key1", view -> {
   // If caller depends on the typed information, this is not an ideal way to retrieve
it
   // If the caller does not depend on the specific type, this works just fine.
   Optional<MetaEntryVersion> version = view.findMetaParam(MetaEntryVersion.class);
   return view.get();
});
```

When generic information is important the user can define a static helper method that coerces the static class retrieval to the type requested, and then use that helper method in the call to `findMetaParam`:

```
class MetaEntryVersion<T> implements MetaParam.Writable<EntryVersion<T>> {
   ...
   public static <T> T type() { return (T) MetaEntryVersion.class; }
   ...
}

ReadOnlyMap<String, String> readOnlyMap = ...

CompletableFuture<String> readFuture = readOnlyMap.eval("key1", view -> {
   // The caller wants guarantees that the metadata parameter for version is numeric
   // e.g. to query the actual version information
   Optional<MetaEntryVersion<Long>> version = view.findMetaParam(MetaEntryVersion.
type());
   return view.get();
});
```

Finally, users are free to create new instances of metadata parameters to suit their needs. They are stored and retrieved in the very same way as done for the metadata parameters already provided by the functional map API.

13.8. Invocation Parameter

Per-invocation parameters are applied to regular functional map API calls to alter the behaviour of certain aspects. Adding per invocation parameters is done using the `withParams(Param<?>…)` method.

`Param.FutureMode` tweaks whether a method returning a `CompletableFuture` will span a thread to invoke the method, or instead will use the caller thread. By default, whenever a call is made to a method returning a `CompletableFuture` , a separate thread will be span to execute the method asynchronously. However, if the caller will immediately block waiting for the `CompletableFuture` to complete, spanning a different thread is wasteful, and hence `Param.FutureMode.COMPLETED` can be passed as per-invocation parameter to avoid creating that extra thread. Example:

```
import org.infinispan.functional.EntryView.*;
import org.infinispan.functional.FunctionalMap.*;
import org.infinispan.functional.Param.*;

ReadOnlyMap<String, String> readOnlyMap = ...
ReadOnlyMap<String, String> readOnlyMapCompleted = readOnlyMap.withParams(FutureMode
.COMPLETED);
Optional<String> readFuture = readOnlyMapCompleted.eval("key1", ReadEntryView::find)
.get();
```

Param.PersistenceMode controls whether a write operation will be propagated to a persistence store. The default behaviour is for all write-operations to be propagated to the persistence store if the cache is configured with a persistence store. By passing PersistenceMode.SKIP as parameter, the write operation skips the persistence store and its effects are only seen in the in-memory contents of the cache. PersistenceMode.SKIP can be used to implement an `Cache.evict()` method which removes data from memory but leaves the persistence store untouched:

```
import org.infinispan.functional.EntryView.*;
import org.infinispan.functional.FunctionalMap.*;
import org.infinispan.functional.Param.*;

WriteOnlyMap<String, String> writeOnlyMap = ...
WriteOnlyMap<String, String> skiPersistMap = writeOnlyMap.withParams(PersistenceMode
.SKIP);
CompletableFuture<Void> removeFuture = skiPersistMap.eval("key1", WriteEntryView:
:remove);
```

Note that there’s no need for another PersistenceMode option to skip reading from the persistence store, because a write operation can skip reading previous value from the store by calling a write-only operation via the WriteOnlyMap.

Finally, new Param implementations are normally provided by the functional map API since they tweak how the internal logic works. So, for the most part of users, they should limit themselves to using the Param instances exposed by the API. The exception to this rule would be advanced users who decide to add new interceptors to the internal stack. These users have the ability to query these parameters within the interceptors.

13.9. Functional Listeners

The functional map offers a listener API, where clients can register for and get notified when events take place. These notifications are post-event, so that means the events are received after the event has happened.

The listeners that can be registered are split into two categories: write listeners and read-write listeners.

13.9.1. Write Listeners

Write listeners enable user to register listeners for any cache entry write events that happen in either a read-write or write-only functional map.

Listeners for write events cannot distinguish between cache entry created and cache entry modify/update events because they don't have access to the previous value. All they know is that a new non-null entry has been written.

However, write event listeners can distinguish between entry removals and cache entry create/modify-update events because they can query what the new entry's value via `ReadEntryView.find()` method.

Adding a write listener is done via the WriteListeners interface which is accessible via both `ReadWriteMap.listeners()` and `WriteOnlyMap.listeners()` method.

A write listener implementation can be defined either passing a function to `onWrite(Consumer<ReadEntryView<K, V>>)` method, or passing a WriteListener implementation to `add(WriteListener<K, V>)` method. Either way, all these methods return an AutoCloseable instance that can be used to de-register the function listener:

```
import org.infinispan.functional.EntryView.*;
import org.infinispan.functional.FunctionalMap.*;
import org.infinispan.functional.Listeners.WriteListeners.WriteListener;

WriteOnlyMap<String, String> woMap = ...

AutoCloseable writeFunctionCloseHandler = woMap.listeners().onWrite(written -> {
   // `written` is a ReadEntryView of the written entry
   System.out.printf("Written: %s%n", written.get());
});
AutoCloseable writeCloseHanlder = woMap.listeners().add(new WriteListener<String,
String>() {
   @Override
   public void onWrite(ReadEntryView<K, V> written) {
      System.out.printf("Written: %s%n", written.get());
   }
});

// Either wrap handler in a try section to have it auto close...
try(writeFunctionCloseHandler) {
   // Write entries using read-write or write-only functional map API
   ...
}
// Or close manually
writeCloseHanlder.close();
```

13.9.2. Read-Write Listeners

Read-write listeners enable users to register listeners for cache entry created, modified and removed events, and also register listeners for any cache entry write events.

Entry created, modified and removed events can only be fired when these originate on a read-write functional map, since this is the only one that guarantees that the previous value has been read, and hence the differentiation between create, modified and removed can be fully guaranteed.

Adding a read-write listener is done via the `ReadWriteListeners` interface which is accessible via `ReadWriteMap.listeners()` method.

If interested in only one of the event types, the simplest way to add a listener is to pass a function to either `onCreate` , `onModify` or `onRemove` methods. All these methods return an AutoCloseable instance that can be used to de-register the function listener:

```
import org.infinispan.functional.EntryView.*;
import org.infinispan.functional.FunctionalMap.*;

ReadWriteMap<String, String> rwMap = ...
AutoCloseable createClose = rwMap.listeners().onCreate(created -> {
   // `created` is a ReadEntryView of the created entry
   System.out.printf("Created: %s%n", created.get());
});
AutoCloseable modifyClose = rwMap.listeners().onModify((before, after) -> {
   // `before` is a ReadEntryView of the entry before update
   // `after` is a ReadEntryView of the entry after update
   System.out.printf("Before: %s%n", before.get());
   System.out.printf("After: %s%n", after.get());
});
AutoCloseable removeClose = rwMap.listeners().onRemove(removed -> {
   // `removed` is a ReadEntryView of the removed entry
   System.out.printf("Removed: %s%n", removed.get());
});
AutoCloseable writeClose = woMap.listeners().onWrite(written -> {
   // `written` is a ReadEntryView of the written entry
   System.out.printf("Written: %s%n", written.get());
});
...
// Either wrap handler in a try section to have it auto close...
try(createClose) {
   // Create entries using read-write functional map API
   ...
}
// Or close manually
modifyClose.close();
```

If listening for two or more event types, it’s better to pass in an implementation of `ReadWriteListener` interface via the `ReadWriteListeners.add()` method. `ReadWriteListener` offers the same `onCreate`/`onModify`/`onRemove` callbacks with default method implementations that are empty:

```java
import org.infinispan.functional.EntryView.*;
import org.infinispan.functional.FunctionalMap.*;
import org.infinispan.functional.Listeners.ReadWriteListeners.ReadWriteListener;

ReadWriteMap<String, String> rwMap = ...
AutoCloseable readWriteClose = rwMap.listeners.add(new ReadWriteListener<String,
String>() {
   @Override
   public void onCreate(ReadEntryView<String, String> created) {
      System.out.printf("Created: %s%n", created.get());
   }

   @Override
   public void onModify(ReadEntryView<String, String> before, ReadEntryView<String,
String> after) {
      System.out.printf("Before: %s%n", before.get());
      System.out.printf("After: %s%n", after.get());
   }

   @Override
   public void onRemove(ReadEntryView<String, String> removed) {
      System.out.printf("Removed: %s%n", removed.get());
   }
);
AutoCloseable writeClose = rwMap.listeners.add(new WriteListener<String, String>() {
   @Override
   public void onWrite(ReadEntryView<K, V> written) {
      System.out.printf("Written: %s%n", written.get());
   }
);

// Either wrap handler in a try section to have it auto close...
try(readWriteClose) {
   // Create/update/remove entries using read-write functional map API
   ...
}
// Or close manually
writeClose.close();
```

13.10. Marshalling of Functions

Running functional map in a cluster of nodes involves marshalling and replication of the operation parameters under certain circumstances.

To be more precise, when write operations are executed in a cluster, regardless of read-write or write-only operations, all the parameters to the method and the functions are replicated to other nodes.

There are multiple ways in which a function can be marshalled. The simplest way, which is also the most costly option in terms of payload size, is to mark the function as `Serializable`:

```
import org.infinispan.functional.EntryView.*;
import org.infinispan.functional.FunctionalMap.*;

WriteOnlyMap<String, String> writeOnlyMap = ...

// Force a function to be Serializable
Consumer<WriteEntryView<String>> function =
   (Consumer<WriteEntryView<String>> & Serializable) wv -> wv.set("one");

CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", function);
```

Infinispan provides overloads for all functional methods that make lambdas passed directly to the API serializable by default; the compiler automatically selects this overload if that's possible. Therefore you can call

```
WriteOnlyMap<String, String> writeOnlyMap = ...
CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", wv -> wv.set("one"));
```

without doing the cast described above.

A more economical way to marshall a function is to provide an Infinispan `Externalizer` for it:

```java
import org.infinispan.functional.EntryView.*;
import org.infinispan.functional.FunctionalMap.*;
import org.infinispan.commons.marshall.Externalizer;
import org.infinispan.commons.marshall.SerializeFunctionWith;

WriteOnlyMap<String, String> writeOnlyMap = ...

// Force a function to be Serializable
Consumer<WriteEntryView<String>> function = new SetStringConstant<>();
CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", function);

@SerializeFunctionWith(value = SetStringConstant.Externalizer0.class)
class SetStringConstant implements Consumer<WriteEntryView<String>> {
   @Override
   public void accept(WriteEntryView<String> view) {
      view.set("value1");
   }

   public static final class Externalizer0 implements Externalizer<Object> {
      public void writeObject(ObjectOutput oo, Object o) {
         // No-op
      }
      public Object readObject(ObjectInput input) {
         return new SetStringConstant<>();
      }
   }
}
```

To help users take advantage of the tiny payloads generated by `Externalizer`-based functions, the functional API comes with a helper class called `org.infinispan.commons.marshall.MarshallableFunctions` which provides marshallable functions for some of the most commonly user functions.

In fact, all the functions required to implement `ConcurrentMap` and `JCache` using the functional map API have been defined in `MarshallableFunctions`. For example, here is an implementation of JCache's `boolean putIfAbsent(K, V)` using functional map API which can be run in a cluster:

```java
import org.infinispan.functional.EntryView.*;
import org.infinispan.functional.FunctionalMap.*;
import org.infinispan.commons.marshall.MarshallableFunctions;

ReadWriteMap<String, String> readWriteMap = ...

CompletableFuture<Boolean> future = readWriteMap.eval("key1,
   MarshallableFunctions.setValueIfAbsentReturnBoolean());
future.thenAccept(stored -> System.out.printf("Value was put? %s%n", stored));
```

13.11. Use Cases for Functional API

This new API is meant to complement existing Key/Value Infinispan API offerings, so you'll still be able to use `ConcurrentMap` or `JCache` standard APIs if that's what suits your use case best.

The target audience for this new API is either:

- Distributed or persistent caching/inmemorydatagrid users that want to benefit from CompletableFuture and/or Traversable for async/lazy data grid or caching data manipulation. The clear advantage here is that threads do not need to be idle waiting for remote operations to complete, but instead these can be notified when remote operations complete and then chain them with other subsequent operations.
- Users who want to go beyond the standard operations exposed by `ConcurrentMap` and `JCache`, for example, if you want to do a replace operation using metadata parameter equality instead of value equality, or if you want to retrieve metadata information from values and so on.

Chapter 14. Indexing and Searching

Infinispan provides a search API that lets you index and search cache values stored as Java POJOs or as objects encoded as Protocol Buffers.

14.1. Overview

Searching is possible both in library and client/server mode (for Java, C#, Node.js and other clients), and Infinispan can index data using Apache Lucene, offering an efficient full-text capable search engine in order to cover a wide range of data retrieval use cases.

Indexing configuration relies on a schema definition, and for that Infinispan can use annotated Java classes when in library mode, and protobuf schemas for remote clients. By standardizing on protobuf, Infinispan allows full interoperability between Java and non-Java clients.

Infinispan has its own query language called Ickle, which is string-based and adds support for full-text searching. Ickle support searches over indexed data, partially indexed data or non-indexed data.

Finally, Infinispan has support for Continuous Queries, which works in a reverse manner to the other APIs: instead of creating, executing a query and obtain results, it allows a client to register queries that will be evaluated continuously as data in the cluster changes, generating notifications whenever the changed data matches the queries.

14.2. Indexing Entry Values

Indexing entry values in Infinispan caches dramatically improves search performance and allows you to perform full-text queries. However, indexing can degrade write throughput for Infinispan clusters. For this reason you should plan to use strategies to optimise query performance, depending on the cache mode and your use case. More information on query performance guide.

14.2.1. Configuration

To enable indexing via XML, you need to add the `<indexing>` element to your cache configuration, specify the entities that are indexed and optionally pass additional properties.

> The presence of an `<indexing>` element which omits the `enabled` attribute will auto-enable indexing for your convenience, even though the default value of the `enabled` attribute is defined as `"false"` in the XSD schema. In the programmatic config, `enabled()` must be used.

Declaratively

```
<infinispan>
   <cache-container default-cache="default">
      <replicated-cache name="default">
         <indexing>
            <indexed-entities>
               <indexed-entity>com.acme.Book</indexed-entity>
            </indexed-entities>
            <property name="property.name">some value</property>
         </indexing>
      </replicated-cache>
   </cache-container>
</infinispan>
```

Programmatically

```
import org.infinispan.configuration.cache.*;

ConfigurationBuilder cacheCfg = ...
cacheCfg.indexing().enable()
            .addIndexedEntity(Book.class)
      .addProperty("property name", "propery value")
```

14.2.2. Specifying Indexed Entities

It is recommended to declare the indexed types, as they will be mandatory in the next Infinispan version.

Declaratively

```
<infinispan>
   <cache-container default-cache="default">
      <replicated-cache name="default">
         <indexing>
            <indexed-entities>
                <indexed-entity>com.acme.query.test.Car</indexed-entity>
                <indexed-entity>com.acme.query.test.Truck</indexed-entity>
            </indexed-entities>
         </indexing>
      </replicated-cache>
   </cache-container>
</infinispan>
```

Programmatically

```
cacheCfg.indexing()
      .addIndexedEntity(Car.class)
      .addIndexedEntity(Truck.class)
```

When the cache is storing protobuf, the indexed types should be the `Message` declared in the protobuf schema. For example, for the schema below:

```
package book_sample;

message Book {
    optional string title = 1;
    optional string description = 2;
    optional int32 publicationYear = 3; // no native Date type available in Protobuf

    repeated Author authors = 4;
}

message Author {
    optional string name = 1;
    optional string surname = 2;
}
```

The config should be:

```
<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="books">
      <indexing>
        <indexed-entities>
          <indexed-entity>book_sample.Book</indexed-entity>
        </indexed-entities>
      </indexing>
    </replicated-cache>
  </cache-container>
</infinispan>
```

14.2.3. Index Storage

Infinispan can store indexes in the file system or in memory (`local-heap`). File system is the recommended and the default configuration, and memory indexes should only be used for small to medium indexes that don't need to survive restart.

Configuration for file system indexes:

```
<replicated-cache name="myCache">
   <indexing>
      <indexed-entities>
         <indexed-entity>com.acme.Book</indexed-entity>
      </indexed-entities>
      <!-- Optional: this is the default setting -->
      <property name="default.directory_provider">filesystem</property>
      <!-- Optional: define base folder for indexes -->
      <property name="default.indexBase">${java.io.tmpdir}/baseDir</property>
   </indexing>
</replicated-cache>
```

Configuration for memory indexes:

```
<replicated-cache name="myCache">
   <indexing>
      <indexed-entities>
         <indexed-entity>com.acme.Book</indexed-entity>
      </indexed-entities>
      <property name="default.directory_provider">local-heap</property>
   </indexing>
</replicated-cache>
```

14.2.4. Index Manager

Infinispan uses internally a component called "Index Manager" to control how new data is applied to the index and when the data is visible to searches.

The default Index Manager `directory-based` writes to the index as soon as the data is written to the cache. The downside is it can slow down considerably cache writes specially under heavy writing scenarios, since it needs to do constant costly operations called "flushes" on the index.

The `near-real-time` index manager is similar to the default index manager but takes advantage of the Near-Real-Time features of Lucene. It has better write performance because it flushes the index to the underlying store less often. The drawback is that unflushed index changes can be lost in case of a non-clean shutdown. Can be used in conjunction with `local-heap` or `filesystem`.

Example with `local-heap`:

```
<replicated-cache name="default">
    <indexing>
        <property name="default.indexmanager">near-real-time</property>
        <property name="default.directory_provider">local-heap</property>
    </indexing>
</replicated-cache>
```

Example with `filesystem`:

```
<replicated-cache name="default">
    <indexing>
        <property name="default.indexmanager">near-real-time</property>
    </indexing>
</replicated-cache>
```

14.2.5. Rebuilding Indexes

Rebuilding an index reconstructs it from data stored in the cache. You need to rebuild indexes if you change things like definitions of indexed types or Analyzers. Likewise you might need to rebuild indexes if they are deleted for some reason. Beware it might take some time as it needs to reprocess all data in the grid!

```
Indexer indexer = Search.getIndexer(cache);
CompletionStage<Void> future = index.run();
```

14.3. Searching

Create relational and full-text queries in both Library and Remote Client-Server mode with the Ickle query language.

To use the API, first obtain a QueryFactory to the cache and then call the `.create()` method, passing in the string to use in the query. Each `QueryFactory` instance is bound to the same `Cache` instance as the `Search`, but it is otherwise a stateless and thread-safe object that can be used for creating multiple queries in parallel.

For instance:

```
// Remote Query, using protobuf
QueryFactory qf = org.infinispan.client.hotrod.Search.getQueryFactory(remoteCache);
Query q = qf.create("from sample_bank_account.Transaction where amount > 20");

// Embedded Query using Java Objects
QueryFactory qf = org.infinispan.query.Search.getQueryFactory(cache);
Query q = qf.create("from com.acme.Book where price > 20");

// Execute the query
QueryResult<Book> queryResult = q.execute();
```

A query will always target a single entity type and is evaluated over the contents of a single cache. Running a query over multiple caches or creating queries that target several entity types (joins) is not supported.

Executing the query and fetching the results is as simple as invoking the `run()` method of the `Query` object. Once executed, calling `run()` on the same instance will re-execute the query.

14.3.1. Pagination

You can limit the number of returned results by using the `Query.maxResults(int maxResults)`. This can be used in conjunction with `Query.startOffset(long startOffset)` to achieve pagination of the result set.

```
// sorted by year and match all books that have "clustering" in their title
// and return the third page of 10 results
Query<Book> query = queryFactory.create("FROM com.acme.Book WHERE title like
'%clustering%' ORDER BY year").startOffset(20).maxResults(10)
```

14.3.2. Number of Hits

The `QueryResult` object has the `.hitCount()` method to return the total number of results of the query, regardless of any pagination parameter. The hit count is only available for indexed queries for performance reasons.

14.3.3. Iteration

The `Query` object has the `.iterator()` method to obtain the results lazily. It returns an instance of `CloseableIterator` that must be closed after usage.

> The iteration support for Remote Queries is currently limited, as it will first fetch all entries to the client before iterating.

14.3.4. Using Named Query Parameters

Instead of building a new Query object for every execution it is possible to include named parameters in the query which can be substituted with actual values before execution. This allows a query to be defined once and be efficiently executed many times. Parameters can only be used on the right-hand side of an operator and are defined when the query is created by supplying an object produced by the `org.infinispan.query.dsl.Expression.param(String paramName)` method to the operator instead of the usual constant value. Once the parameters have been defined they can be set by invoking either `Query.setParameter(parameterName, value)` or `Query.setParameters(parameterMap)` as shown in the examples below.

```
QueryFactory queryFactory = Search.getQueryFactory(cache);
// Defining a query to search for various authors and publication years
Query<Book> query = queryFactory.create("SELECT title FROM com.acme.Book WHERE author
= :authorName AND publicationYear = :publicationYear").build();

// Set actual parameter values
query.setParameter("authorName", "Doe");
query.setParameter("publicationYear", 2010);

// Execute the query
List<Book> found = query.list();
```

Alternatively, you can supply a map of actual parameter values to set multiple parameters at once:

Setting multiple named parameters at once

```
Map<String, Object> parameterMap = new HashMap<>();
parameterMap.put("authorName", "Doe");
parameterMap.put("publicationYear", 2010);

query.setParameters(parameterMap);
```

> A significant portion of the query parsing, validation and execution planning effort is performed during the first execution of a query with parameters. This effort is not repeated during subsequent executions leading to better performance compared to a similar query using constant values instead of query parameters.

14.3.5. Ickle Query Language Parser Syntax

The Ickle query language is small subset of the JPQL query language, with some extensions for full-text.

The parser syntax has some notable rules:

- Whitespace is not significant.
- Wildcards are not supported in field names.
- A field name or path must always be specified, as there is no default field.
- `&&` and `||` are accepted instead of `AND` or `OR` in both full-text and JPA predicates.
- `!` may be used instead of `NOT`.
- A missing boolean operator is interpreted as `OR`.
- String terms must be enclosed with either single or double quotes.
- Fuzziness and boosting are not accepted in arbitrary order; fuzziness always comes first.
- `!=` is accepted instead of `<>`.
- Boosting cannot be applied to `>`,`>=`,`<`,`⇐` operators. Ranges may be used to achieve the same

result.

Filtering operators

Ickle support many filtering operators that can be used for both indexed and non-indexed fields.

Operator	Description	Example
in	Checks that the left operand is equal to one of the elements from the Collection of values given as argument.	FROM Book WHERE isbn IN ('ZZ', 'X1234')
like	Checks that the left argument (which is expected to be a String) matches a wildcard pattern that follows the JPA rules.	FROM Book WHERE title LIKE '%Java%'
=	Checks that the left argument is an exact match of the given value	FROM Book WHERE name = 'Programming Java'
!=	Checks that the left argument is different from the given value	FROM Book WHERE language != 'English'
>	Checks that the left argument is greater than the given value.	FROM Book WHERE price > 20
>=	Checks that the left argument is greater than or equal to the given value.	FROM Book WHERE price >= 20
<	Checks that the left argument is less than the given value.	FROM Book WHERE year < 2012
⇐	Checks that the left argument is less than or equal to the given value.	FROM Book WHERE price ⇐ 50
between	Checks that the left argument is between the given range limits.	FROM Book WHERE price BETWEEN 50 AND 100

Boolean conditions

Combining multiple attribute conditions with logical conjunction (`and`) and disjunction (`or`) operators in order to create more complex conditions is demonstrated in the following example. The well known operator precedence rule for boolean operators applies here, so the order of the operators is irrelevant. Here `and` operator still has higher priority than `or` even though `or` was invoked first.

```
# match all books that have "Data Grid" in their title
# or have an author named "Manik" and their description contains "clustering"

FROM com.acme.Book WHERE title LIKE '%Data Grid%' OR author.name = 'Manik' AND
description like '%clustering%'
```

Boolean negation has highest precedence among logical operators and applies only to the next simple attribute condition.

```
# match all books that do not have "Data Grid" in their title and are authored by
"Manik"
FROM com.acme.Book WHERE title != 'Data Grid' AND author.name = 'Manik'
```

Nested conditions

Changing the precedence of logical operators is achieved with parenthesis:

```
# match all books that have an author named "Manik" and their title contains
# "Data Grid" or their description contains "clustering"
FROM com.acme.Book WHERE author.name = 'Manik' AND ( title like '%Data Grid%' OR
description like '% clustering%')
```

Selecting attributes

In some use cases returning the whole domain object is overkill if only a small subset of the attributes are actually used by the application, especially if the domain entity has embedded entities. The query language allows you to specify a subset of attributes (or attribute paths) to return - the projection. If projections are used then the `QueryResult.list()` will not return the whole domain entity but will return a `List` of `Object[]`, each slot in the array corresponding to a projected attribute.

```
# match all books that have "Data Grid" in their title or description
# and return only their title and publication year
SELECT title, publicationYear FROM com.acme.Book WHERE title like '%Data Grid%' OR
description like '%Data Grid%'
```

Sorting

Ordering the results based on one or more attributes or attribute paths is done with the `ORDER BY` clause. If multiple sorting criteria are specified, then the order will dictate their precedence.

```
# match all books that have "Data Grid" in their title or description
# and return them sorted by the publication year and title
FROM com.acme.Book WHERE title like '%Data Grid%' ORDER BY publicationYear DESC, title
ASC
```

Grouping and Aggregation

Infinispan has the ability to group query results according to a set of grouping fields and construct aggregations of the results from each group by applying an aggregation function to the set of values that fall into each group. Grouping and aggregation can only be applied to projection queries (queries with one or more field in the SELECT clause).

The supported aggregations are: avg, sum, count, max, min.

The set of grouping fields is specified with the `GROUP BY` clause and the order used for defining grouping fields is not relevant. All fields selected in the projection must either be grouping fields or else they must be aggregated using one of the grouping functions described below. A projection field can be aggregated and used for grouping at the same time. A query that selects only grouping fields but no aggregation fields is legal. Example: Grouping Books by author and counting them.

```
SELECT author, COUNT(title) FROM com.acme.Book WHERE title LIKE '%engine%' GROUP BY
author
```

> A projection query in which all selected fields have an aggregation function applied and no fields are used for grouping is allowed. In this case the aggregations will be computed globally as if there was a single global group.

Aggregations

The following aggregation functions can be applied to a field:

- `avg()` - Computes the average of a set of numbers. Accepted values are primitive numbers and instances of `java.lang.Number`. The result is represented as `java.lang.Double`. If there are no non-null values the result is `null` instead.
- `count()` - Counts the number of non-null rows and returns a `java.lang.Long`. If there are no non-null values the result is `0` instead.
- `max()` - Returns the greatest value found. Accepted values must be instances of `java.lang.Comparable`. If there are no non-null values the result is `null` instead.
- `min()` - Returns the smallest value found. Accepted values must be instances of `java.lang.Comparable`. If there are no non-null values the result is `null` instead.
- `sum()` - Computes the sum of a set of Numbers. If there are no non-null values the result is `null` instead. The following table indicates the return type based on the specified field.

Table 1. Table sum return type

Field Type	Return Type
Integral (other than BigInteger)	Long
Float or Double	Double
BigInteger	BigInteger
BigDecimal	BigDecimal

Evaluation of queries with grouping and aggregation

Aggregation queries can include filtering conditions, like usual queries. Filtering can be performed in two stages: before and after the grouping operation. All filter conditions defined before invoking the `groupBy()` method will be applied before the grouping operation is performed, directly to the cache entries (not to the final projection). These filter conditions can reference any fields of the queried entity type, and are meant to restrict the data set that is going to be the input for the grouping stage. All filter conditions defined after invoking the `groupBy()` method will be applied to the projection that results from the projection and grouping operation. These filter conditions can either reference any of the `groupBy()` fields or aggregated fields. Referencing aggregated fields that are not specified in the select clause is allowed; however, referencing non-aggregated and non-grouping fields is forbidden. Filtering in this phase will reduce the amount of groups based on their properties. Sorting can also be specified similar to usual queries. The ordering operation is performed after the grouping operation and can reference any of the `groupBy()` fields or aggregated fields.

Using Full-text search

Fuzzy Queries

To execute a fuzzy query add `~` along with an integer, representing the distance from the term used, after the term. For instance

```
FROM sample_bank_account.Transaction WHERE description : 'cofee'~2
```

Range Queries

To execute a range query define the given boundaries within a pair of braces, as seen in the following example:

```
FROM sample_bank_account.Transaction WHERE amount : [20 to 50]
```

Phrase Queries

A group of words can be searched by surrounding them in quotation marks, as seen in the following example:

```
FROM sample_bank_account.Transaction WHERE description : 'bus fare'
```

Proximity Queries

To execute a proximity query, finding two terms within a specific distance, add a ~ along with the distance after the phrase. For instance, the following example will find the words canceling and fee provided they are not more than 3 words apart:

```
FROM sample_bank_account.Transaction WHERE description : 'canceling fee'~3
```

Wildcard Queries

To search for "text" or "test", use the ? single-character wildcard search:

```
FROM sample_bank_account.Transaction where description : 'te?t'
```

To search for "test", "tests", or "tester", use the * multi-character wildcard search:

```
FROM sample_bank_account.Transaction where description : 'test*'
```

Regular Expression Queries

Regular expression queries can be performed by specifying a pattern between /. Ickle uses Lucene's regular expression syntax, so to search for the words `moat` or `boat` the following could be used:

```
FROM sample_library.Book  where title : /[mb]oat/
```

Boosting Queries

Terms can be boosted by adding a ^ after the term to increase their relevance in a given query, the higher the boost factor the more relevant the term will be. For instance to search for titles containing beer and wine with a higher relevance on beer, by a factor of 3, the following could be used:

```
FROM sample_library.Book WHERE title : beer^3 OR wine
```

14.4. Embedded Search

Embedded searching is available when Infinispan is used as a library. No protobuf mapping is required, and both indexing and searching are done on top of Java objects.

14.4.1. Quick example

We're going to store `Book` instances in an Infinispan cache called "books". `Book` instances will be indexed, so we enable indexing for the cache:

Infinispan configuration:

infinispan.xml

```
<infinispan>
    <cache-container>
        <transport cluster="infinispan-cluster"/>
        <distributed-cache name="books">
            <indexing>
                <indexed-entities>
                    <indexed-entity>com.acme.Book</indexed-entity>
                </indexed-entities>
            </indexing>
        </distributed-cache>
    </cache-container>
</infinispan>
```

Obtaining the cache:

```
import org.infinispan.Cache;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.manager.EmbeddedCacheManager;

EmbeddedCacheManager manager = new DefaultCacheManager("infinispan.xml");
Cache<String, Book> cache = manager.getCache("books");
```

Each `Book` will be defined as in the following example; we have to choose which properties are indexed, and for each property we can optionally choose advanced indexing options using the annotations defined in the Hibernate Search project.

Book.java

```
import org.hibernate.search.annotations.*;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;

//Values you want to index need to be annotated with @Indexed, then you pick which
fields and how they are to be indexed:
@Indexed
public class Book {
   @Field String title;
   @Field String description;
   @Field @DateBridge(resolution=Resolution.YEAR) Date publicationYear;
   @IndexedEmbedded Set<Author> authors = new HashSet<Author>();
}
```

Author.java

```
public class Author {
   @Field String name;
   @Field String surname;
   // hashCode() and equals() omitted
}
```

Now assuming we stored several `Book` instances in our Infinispan `Cache` , we can search them for any matching field as in the following example.

QueryExample.java

```
// get the query factory from the cache:
QueryFactory queryFactory = org.infinispan.query.Search.getQueryFactory(cache);

// create an Ickle query that will do a full-text search (operator ':') on fields
'title' and 'authors.name'
Query<Book> fullTextQuery = queryFactory.create("FROM com.acme.Book WHERE
title:'infinispan' AND authors.name:'sanne'")

// The ('=') operator is not a full-text operator, thus can be used in both indexed
and non-indexed caches
Query<Book> exactMatchQuery = queryFactory.create("FROM com.acme.Book WHERE title =
'Programming Infinispan' AND authors.name = 'Sanne Grinnovero'")

// Full-text and non-full text operators can be part of the same query
Query q = queryFactory.create("FROM com.query.Book b where b.author.name = 'Stephen'
and b.description : (+'dark' -'tower')");

// get the results
List<Book> found=query.execute().list();
```

Apart from `list()` you have the option for obtaining on `iterator()`, or use pagination.

14.4.2. Mapping Entities

Infinispan relies on the rich API of Hibernate Search in order to define fine grained configuration for indexing at entity level. This configuration includes which fields are annotated, which analyzers should be used, how to map nested objects and so on. Detailed documentation is available at the Hibernate Search manual.

@DocumentId

Unlike Hibernate Search, using `@DocumentId` to mark a field as identifier does not apply to Infinispan values; in Infinispan the identifier for all `@Indexed` objects is the key used to store the value. You can still customize how the key is indexed using a combination of `@Transformable` , custom types and custom `FieldBridge` implementations.

@Transformable keys

The key for each value needs to be indexed as well, and the key instance must be transformed in a `String`. Infinispan includes some default transformation routines to encode common primitives, but to use a custom key you must provide an implementation of `org.infinispan.query.Transformer` .

Registering a key Transformer via annotations

You can annotate your key class with `org.infinispan.query.Transformable` and your custom transformer implementation will be picked up automatically:

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

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

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

Registering a key Transformer via the cache indexing configuration

Use the `key-transformers` xml element in both embedded and server config:

```
<replicated-cache name="test">
    <indexing auto-config="true">
        <key-transformers>
            <key-transformer key="com.mycompany.CustomKey" transformer=
"com.mycompany.CustomTransformer"/>
        </key-transformers>
    </indexing>
</replicated-cache>
```

Alternatively, use the Java configuration API (embedded mode):

```
   ConfigurationBuilder builder = ...
   builder.indexing().enable()
         .addKeyTransformer(CustomKey.class, CustomTransformer.class);
```

Programmatic mapping

Instead of using annotations to map an entity to the index, it's also possible to configure it programmatically.

In the following example we map an object `Author` which is to be stored in the grid and made searchable on two properties but without annotating the class.

```java
import org.apache.lucene.search.Query;
import org.hibernate.search.cfg.Environment;
import org.hibernate.search.cfg.SearchMapping;
import org.hibernate.search.query.dsl.QueryBuilder;
import org.infinispan.Cache;
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.configuration.cache.Index;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.query.CacheQuery;
import org.infinispan.query.Search;
import org.infinispan.query.SearchManager;

import java.io.IOException;
import java.lang.annotation.ElementType;
import java.util.Properties;

SearchMapping mapping = new SearchMapping();
mapping.entity(Author.class).indexed()
       .property("name", ElementType.METHOD).field()
       .property("surname", ElementType.METHOD).field();

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

Configuration infinispanConfiguration = new ConfigurationBuilder()
        .indexing().index(Index.NONE)
        .withProperties(properties)
        .build();

DefaultCacheManager cacheManager = new DefaultCacheManager(infinispanConfiguration);

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

Author author = new Author(1, "Manik", "Surtani");
cache.put(author.getId(), author);

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

14.5. Remote Search

Remote search is very similar to embedded with the notable difference that data must use Google Protocol Buffers as an encoding for both over-the-wire and storage. Furthermore, it's necessary to write (or generate from Java classes) a protobuf schema defining the data structure and indexing elements instead of relying on Hibernate Search annotations.

The usage of protobuf allows remote query to work not only for Java, but for REST, C# and Node.js clients.

14.5.1. A remote query example

We are going to revisit the Book Sample from embedded query, but this time using the Java Hot Rod client and the Infinispan server. An object called `Book` will be stored in a Infinispan cache called "books". Book instances will be indexed, so we enable indexing for the cache:

infinispan.xml

```
<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="books">
      <indexing>
        <indexed-entities>
          <indexed-entity>book_sample.Book</indexed-entity>
        </indexed-entities>
      </indexing>
    </replicated-cache>
  </cache-container>
</infinispan>
```

Alternatively, if indexing the cache is not indexed, we configure the `<encoding>` as `application/x-protostream` to make sure the storage is queryable:

infinispan.xml

```
<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="books">
      <encoding media-type="application/x-protostream"/>
    </replicated-cache>
  </cache-container>
</infinispan>
```

Each `Book` will be defined as in the following example: we use `@Protofield` annotations to identify the protocol buffers message fields and the `@ProtoDoc` annotation on the fields to configure indexing attributes:

Book.java

```java
import org.infinispan.protostream.annotations.ProtoDoc;
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;

@ProtoDoc("@Indexed")
public class Book {

   @ProtoDoc("@Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO)")
   @ProtoField(number = 1)
   final String title;

   @ProtoDoc("@Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO)")
   @ProtoField(number = 2)
   final String description;

   @ProtoDoc("@Field(index=Index.YES, analyze = Analyze.YES, store = Store.NO)")
   @ProtoField(number = 3, defaultValue = "0")
   final int publicationYear;


   @ProtoFactory
   Book(String title, String description, int publicationYear) {
      this.title = title;
      this.description = description;
      this.publicationYear = publicationYear;
   }
   // public Getter methods omitted for brevity
}
```

The annotations above will generate during compilation the artifacts necessary to read, write and query `Book` instances. To enable this generation, use the `@AutoProtoSchemaBuilder` annotation in a newly created class with empty constructor or interface:

RemoteQueryInitializer.java

```java
import org.infinispan.protostream.SerializationContextInitializer;
import org.infinispan.protostream.annotations.AutoProtoSchemaBuilder;

@AutoProtoSchemaBuilder(
      includeClasses = {
            Book.class
      },
      schemaFileName = "book.proto",
      schemaFilePath = "proto/",
      schemaPackageName = "book_sample")
public interface RemoteQueryInitializer extends SerializationContextInitializer {
}
```

After compilation, a file `book.proto` file will be created in the configured `schemaFilePath`, along with an implementation `RemoteQueryInitializerImpl.java` of the annotated interface. This concrete class can be used directly in the Hot Rod client code to initialize the serialization context.

Putting all together:

RemoteQuery.java

```
package org.infinispan;

import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.List;

import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.Search;
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.query.dsl.Query;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.remote.client.ProtobufMetadataManagerConstants;

public class RemoteQuery {

   public static void main(String[] args) throws Exception {
      ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
      // RemoteQueryInitializerImpl is generated
      clientBuilder.addServer().host("127.0.0.1").port(11222)
            .security().authentication().username("user").password("user")
            .addContextInitializers(new RemoteQueryInitializerImpl());

      RemoteCacheManager remoteCacheManager = new RemoteCacheManager(clientBuilder
.build());

      // Grab the generated protobuf schema and registers in the server.
      Path proto = Paths.get(RemoteQuery.class.getClassLoader()
            .getResource("proto/book.proto").toURI());
      String protoBufCacheName = ProtobufMetadataManagerConstants
.PROTOBUF_METADATA_CACHE_NAME;
      remoteCacheManager.getCache(protoBufCacheName).put("book.proto", Files
.readString(proto));

      // Obtain the 'books' remote cache
      RemoteCache<Object, Object> remoteCache = remoteCacheManager.getCache("books");

      // Add some Books
      Book book1 = new Book("Infinispan in Action", "Learn Infinispan with using it",
2015);
      Book book2 = new Book("Cloud-Native Applications with Java and Quarkus", "Build
robust and reliable cloud applications", 2019);
```

```
        remoteCache.put(1, book1);
        remoteCache.put(2, book2);

        // Execute a full-text query
        QueryFactory queryFactory = Search.getQueryFactory(remoteCache);
        Query<Book> query = queryFactory.create("FROM book_sample.Book WHERE
title:'java'");

        List<Book> list = query.execute().list(); // Voila! We have our book back from
the cache!
    }
}
```

14.5.2. Indexing of Protobuf encoded entries

As seen in Remote Query example, one step necessary to query protobuf entities is to provide the client and server with the relevant metadata about entities (.proto file).

The descriptors are stored in a dedicated cache on the server named `___protobuf_metadata`. Both keys and values in this cache are plain strings. Registering a new schema is therefore as simple as performing a `put()` operation on this cache using the schema's name as key and the schema file itself as the value.

Alternatively you can use the CLI (via the `cache-container=*:register-proto-schemas()` operation), the Management Console, the REST endpoint `/rest/v2/schemas` or the `ProtobufMetadataManager` MBean via JMX. Be aware that, when security is enabled, access to the schema cache via the remote protocols requires that the user belongs to the '___schema_manager' role.

> Even if indexing is enabled for a cache no fields of Protobuf encoded entries will be indexed unless you use the `@Indexed` and `@Field` inside protobuf schema documentation annotations `(@ProtoDoc)` to specify what fields need to get indexed.

14.5.3. Analysis

Analysis is a process that converts input data into one or more terms that you can index and query. While in Embedded Query mapping is done through Hibernate Search annotations, that supports a rich set of Lucene based analyzers, in client-server mode the analyzer definitions are declared in a platform neutral way

Default Analyzers

Infinispan provides a set of default analyzers for remote query as follows:

Definition	Description
`standard`	Splits text fields into tokens, treating whitespace and punctuation as delimiters.

Definition	Description
`simple`	Tokenizes input streams by delimiting at non-letters and then converting all letters to lowercase characters. Whitespace and non-letters are discarded.
`whitespace`	Splits text streams on whitespace and returns sequences of non-whitespace characters as tokens.
`keyword`	Treats entire text fields as single tokens.
`stemmer`	Stems English words using the Snowball Porter filter.
`ngram`	Generates n-gram tokens that are 3 grams in size by default.
`filename`	Splits text fields into larger size tokens than the `standard` analyzer, treating whitespace as a delimiter and converts all letters to lowercase characters.

These analyzer definitions are based on Apache Lucene and are provided "as-is". For more information about tokenizers, filters, and CharFilters, see the appropriate Lucene documentation.

Using Analyzer Definitions

To use analyzer definitions, reference them by name in the `.proto` schema file.

1. Include the `Analyze.YES` attribute to indicate that the property is analyzed.
2. Specify the analyzer definition with the `@Analyzer` annotation.

The following example shows referenced analyzer definitions:

```
/* @Indexed */
message TestEntity {

    /* @Field(store = Store.YES, analyze = Analyze.YES, analyzer =
@Analyzer(definition = "keyword")) */
    optional string id = 1;

    /* @Field(store = Store.YES, analyze = Analyze.YES, analyzer =
@Analyzer(definition = "simple")) */
    optional string name = 2;
}
```

If using Java classes annotated with `@ProtoField`, the declaration is similar:

```
@ProtoDoc("@Field(store = Store.YES, analyze = Analyze.YES, analyzer =
@Analyzer(definition = \"keyword\"))")
@ProtoField(number = 1)
final String id;

@ProtoDoc("@Field(store = Store.YES, analyze = Analyze.YES, analyzer =
@Analyzer(definition = \"simple\"))")
@ProtoField(number = 2)
final String description;
```

Creating Custom Analyzer Definitions

If you require custom analyzer definitions, do the following:

1. Create an implementation of the `ProgrammaticSearchMappingProvider` interface packaged in a `JAR` file.
2. Provide a file named `org.infinispan.query.spi.ProgrammaticSearchMappingProvider` in the `META-INF/services/` directory of your `JAR`. This file should contain the fully qualified class name of your implementation.
3. Copy the `JAR` to the `lib/` directory of your Infinispan installation.

> Your jar must be available to the Infinispan server during startup. You cannot add it if the server is already running.

The following is an example implementation of the `ProgrammaticSearchMappingProvider` interface:

```
import org.apache.lucene.analysis.core.LowerCaseFilterFactory;
import org.apache.lucene.analysis.core.StopFilterFactory;
import org.apache.lucene.analysis.standard.StandardFilterFactory;
import org.apache.lucene.analysis.standard.StandardTokenizerFactory;
import org.hibernate.search.cfg.SearchMapping;
import org.infinispan.Cache;
import org.infinispan.query.spi.ProgrammaticSearchMappingProvider;

public final class MyAnalyzerProvider implements ProgrammaticSearchMappingProvider
{

   @Override
   public void defineMappings(Cache cache, SearchMapping searchMapping) {
      searchMapping
            .analyzerDef("standard-with-stop", StandardTokenizerFactory.class)
               .filter(StandardFilterFactory.class)
               .filter(LowerCaseFilterFactory.class)
               .filter(StopFilterFactory.class);
   }
}
```

14.6. Continuous Query

Continuous Queries allow an application to register a listener which will receive the entries that currently match a query filter, and will be continuously notified of any changes to the queried data set that result from further cache operations. This includes incoming matches, for values that have joined the set, updated matches, for matching values that were modified and continue to match, and outgoing matches, for values that have left the set. By using a Continuous Query the application receives a steady stream of events instead of having to repeatedly execute the same query to discover changes, resulting in a more efficient use of resources. For instance, all of the following use cases could utilize Continuous Queries:

- Return all persons with an age between 18 and 25 (assuming the Person entity has an `age` property and is updated by the user application).
- Return all transactions higher than $2000.
- Return all times where the lap speed of F1 racers were less than 1:45.00s (assuming the cache contains Lap entries and that laps are entered live during the race).

14.6.1. Continuous Query Execution

A continuous query uses a listener that is notified when:

- An entry starts matching the specified query, represented by a `Join` event.
- A matching entry is updated and continues to match the query, represented by an `Update` vent.
- An entry stops matching the query, represented by a `Leave` event.

When a client registers a continuous query listener it immediately begins to receive the results currently matching the query, received as `Join` events as described above. In addition, it will receive subsequent notifications when other entries begin matching the query, as `Join` events, or stop matching the query, as `Leave` events, as a consequence of any cache operations that would normally generate creation, modification, removal, or expiration events. Updated cache entries will generate `Update` events if the entry matches the query filter before and after the operation. To summarize, the logic used to determine if the listener receives a `Join`, `Update` or `Leave` event is:

1. If the query on both the old and new values evaluate false, then the event is suppressed.
2. If the query on the old value evaluates false and on the new value evaluates true, then a `Join` event is sent.
3. If the query on both the old and new values evaluate true, then an `Update` event is sent.
4. If the query on the old value evaluates true and on the new value evaluates false, then a `Leave` event is sent.
5. If the query on the old value evaluates true and the entry is removed or expired, then a `Leave` event is sent.

> Continuous Queries can use all query capabilities except: grouping, aggregation, and sorting operations.

14.6.2. Running Continuous Queries

To create a continuous query, do the following:

1. Create a Query object. See the searching section
2. Obtain the ContinuousQuery (`org.infinispan.query.api.continuous.ContinuousQuery` object of your cache by calling the appropriate method:
   - `org.infinispan.client.hotrod.Search.getContinuousQuery(RemoteCache<K, V> cache)` for remote mode
   - `org.infinispan.query.Search.getContinuousQuery(Cache<K, V> cache)` for embedded mode
3. Register the query and a continuous query listener (`org.infinispan.query.api.continuous.ContinuousQueryListener`) as follows:

```
continuousQuery.addContinuousQueryListener(query, listener);
```

The following example demonstrates a simple continuous query use case in embedded mode:

Registering a Continuous Query

```
import org.infinispan.query.api.continuous.ContinuousQuery;
import org.infinispan.query.api.continuous.ContinuousQueryListener;
import org.infinispan.query.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

[...]

// We have a cache of Persons
Cache<Integer, Person> cache = ...

// We begin by creating a ContinuousQuery instance on the cache
ContinuousQuery<Integer, Person> continuousQuery = Search.getContinuousQuery(cache);

// Define our query. In this case we will be looking for any Person instances under 21
years of age.
QueryFactory queryFactory = Search.getQueryFactory(cache);
Query query = queryFactory.create("FROM Person p WHERE p.age < 21");

final Map<Integer, Person> matches = new ConcurrentHashMap<Integer, Person>();

// Define the ContinuousQueryListener
ContinuousQueryListener<Integer, Person> listener = new ContinuousQueryListener
<Integer, Person>() {
    @Override
    public void resultJoining(Integer key, Person value) {
```

```
        matches.put(key, value);
    }

    @Override
    public void resultUpdated(Integer key, Person value) {
        // we do not process this event
    }

    @Override
    public void resultLeaving(Integer key) {
        matches.remove(key);
    }
};

// Add the listener and the query
continuousQuery.addContinuousQueryListener(query, listener);

[...]

// Remove the listener to stop receiving notifications
continuousQuery.removeContinuousQueryListener(listener);
```

As Person instances having an age less than 21 are added to the cache they will be received by the listener and will be placed into the `matches` map, and when these entries are removed from the cache or their age is modified to be greater or equal than 21 they will be removed from `matches`.

14.6.3. Removing Continuous Queries

To stop the query from further execution just remove the listener:

```
continuousQuery.removeContinuousQueryListener(listener);
```

14.6.4. Notes on performance of Continuous Queries

Continuous queries are designed to provide a constant stream of updates to the application, potentially resulting in a very large number of events being generated for particularly broad queries. A new temporary memory allocation is made for each event. This behavior may result in memory pressure, potentially leading to `OutOfMemoryErrors` (especially in remote mode) if queries are not carefully designed. To prevent such issues it is strongly recommended to ensure that each query captures the minimal information needed both in terms of number of matched entries and size of each match (projections can be used to capture the interesting properties), and that each `ContinuousQueryListener` is designed to quickly process all received events without blocking and to avoid performing actions that will lead to the generation of new matching events from the cache it listens to.

14.7. Statistics

Query Statistics can be obtained from the SearchManager, as demonstrated in the following code snippet.

```
SearchManager searchManager = Search.getSearchManager(cache);
org.hibernate.search.stat.Statistics statistics = searchManager.getStatistics();
```

> This data is also available via JMX through the Hibernate Search StatisticsInfoMBean registered under the name `org.infinispan:type=Query,manager="{name-of-cache-manager}",cache="{name-of-cache}",component=Statistics`. Please note this MBean is always registered by Infinispan but the statistics are collected only if statistics collection is enabled at cache level.

> Hibernate Search has its own configuration properties `hibernate.search.jmx_enabled` and `hibernate.search.generate_statistics` for JMX statistics as explained here. Using them with Infinispan Query is forbidden as it will only lead to duplicated MBeans and unpredictable results.

14.8. Performance Tuning

14.8.1. Batch writing in SYNC mode

By default, the Index Managers work in sync mode, meaning when data is written to Infinispan, it will perform the indexing operations synchronously. This synchronicity guarantees indexes are always consistent with the data (and thus visible in searches), but can slowdown write operations since it will also perform a commit to the index. Committing is an extremely expensive operation in Lucene, and for that reason, multiple writes from different nodes can be automatically batched into a single commit to reduce the impact.

So, when doing data loads to Infinispan with index enabled, try to use multiple threads to take advantage of this batching.

If using multiple threads does not result in the required performance, an alternative is to load data with indexing temporarily disabled and run a re-indexing operation afterwards. This can be done writing data with the SKIP_INDEXING flag:

```
cache.getAdvancedCache().withFlags(Flag.SKIP_INDEXING).put("key","value");
```

14.8.2. Writing using async mode

If it's acceptable a small delay between data writes and when that data is visible in queries, an index manager can be configured to work in **async mode**. The async mode offers much better writing performance, since in this mode commits happen at a configurable interval.

Configuration:

```
<distributed-cache name="default">
    <indexing>
        <!-- Index data in async mode -->
        <property name="default.worker.execution">async</property>
        <!-- Optional: configure the commit interval, default is 1000ms -->
        <property name="default.index_flush_interval">500</property>
    </indexing>
</distributed-cache>
```

14.8.3. Index reader async strategy

Lucene internally works with snapshots of the index: once an `IndexReader` is opened, it will only see the index changes up to the point it was opened; further index changes will not be visible until the `IndexReader` is refreshed. The Index Managers used in Infinispan by default will check the freshness of the index readers before every query and refresh them if necessary.

It is possible to tune this strategy to relax this freshness checking to a pre-configured interval by using the `reader.strategy` configuration set as `async`:

```
<distributed-cache name="default">
    <indexing>
        <property name="default.reader.strategy">async</property>
        <!-- refresh reader every 1s, default is 5s -->
        <property name="default.reader.async_refresh_period_ms">1000</property>
    </indexing>
</distributed-cache>
```

14.8.4. Lucene Options

It is possible to apply tuning options in Lucene directly. For more details, see the Hibernate Search manual.

Chapter 15. 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 Infinispan. However Infinispan can provide many more benefits that aren't immediately apparent. Since Infinispan 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.

> 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.

15.1. 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.

> 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 inteface 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.

15.1.1. 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(...)
```

15.1.2. 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.

15.1.3. 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(...)
```

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.

15.1.4. 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 Sa = 4r2 and area of the circle is Ca=pi*r2. Substituting r2 from the second equation into the first one it turns out that pi = 4 * Ca/Sa. 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 Infinispan cluster. Note this will work in a cluster of 1 was well, but will be slower.

```
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 16. 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. Infinispan 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.

> 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.

16.1. Common stream operations

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

16.2. 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.

16.3. Segment based filtering

> This is an advanced feature and should only be used with deep knowledge of Infinispan 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.

16.4. 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. Infinispan 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.

16.5. 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));
```

16.6. 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.

16.6.1. 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.

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

16.6.2. Serialization

Since the operations are sent across to other nodes they must be serializable by Infinispan 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. Infinispan 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;
      }

      @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;
      }
   }
```

16.7. 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`.

16.8. 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.

16.9. 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.

16.10. Distributed Stream execution

Distributed streams execution works in a fashion very similiar 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.

16.11. 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, Infinispan 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, Infinispan 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.

16.12. Intermediate operation exceptions

There are some intermediate operations that have special exceptions, these are skip, peek, sorted 1 2. & 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.

16.13. 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 Infinispan 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 17. JCache (JSR-107) API

Infinispan 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 Infinispan implementation of the specification, and explains key aspects of the API.

17.1. Creating embedded caches

Prerequisites

1. Ensure that `cache-api` is on your classpath.
2. Add the following dependency to your `pom.xml`:

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

Procedure

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

```
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>());
```

17.1.1. Configuring embedded caches

- Pass the URI for custom Infinispan 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);
```

> 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. Infinispan has so far implemented this using serialization/marshalling to make copies to store in the cache, and that way adhere to the spec. Hence, if using default JCache configuration with Infinispan, data stored must be marshallable.

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

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

17.2. 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 Infinispan servers and use the default JCache API configuration as follows:

```
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>());
```

17.2.1. 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);
```

17.3. 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:

```java
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).

> 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.

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

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

17.5. Clustering JCache instances

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

infinispan.xml

```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:

```java
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 18. Multimap Cache

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

18.1. Installation and configuration

pom.xml

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

18.2. 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.

```
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.

18.3. 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].

18.3.1. 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);
```

18.4. Limitations

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

18.4.1. 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`.

18.4.2. 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.

18.4.3. 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 19. Anchored Keys module

Infinispan version 11 introduces an experimental module that allows scaling up a cluster and adding new nodes without expensive **state transfer**.

19.1. Background

For background, the preferred way to scale up the storage capacity of a Infinispan cluster is to use distributed caches. A distributed cache stores each key/value pair on `num-owners` nodes, and each node can compute the location of a key (aka the key owners) directly.

Infinispan achieves this by statically mapping cache keys to `num-segments` **consistent hash segments**, and then dynamically mapping segments to nodes based on the cache's **topology** (roughly the current plus the historical membership of the cache). Whenever a new node joins the cluster, the cache is **rebalanced**, and the new node replaces an existing node as the owner of some segments. The key/value pairs in those segments are copied to the new node and removed from the no-longer-owner node via **state transfer**.

> Because the allocation of segments to nodes is based on random UUIDs generated at start time, it is common (though less so after ISPN-11679), for segments to also move from one old node to another old node.

19.2. Architecture

The basic idea is to skip the static mapping of keys to segments and to map keys directly to nodes.

When a key/value pair is inserted into the cache, the newest member becomes the **anchor owner** of that key, and the only node storing the actual value. In order to make the anchor location available without an extra remote lookup, all the other nodes store a reference to the anchor owner.

That way, when another node joins, it only needs to receive the location information from the existing nodes, and values can stay on the anchor owner, minimizing the amount of traffic.

19.3. Limitations

Only one node can be added at a time

An external actor (e.g. a Kubernetes/OpenShift operator, or a human administrator) must monitor the load on the current nodes, and add a new node whenever the newest node is close to "full".

> Because the anchor owner information is replicated on all the nodes, and values are never moved off a node, the memory usage of each node will keep growing as new entries and nodes are added.

There is no redundancy

Every value is only stored on a single node. When a node crashes or even stops gracefully, the values stored on that node are lost.

Transactions are not supported

A later version may add transaction support, but the fact that any node stop or crash loses entries makes transactions a lot less valuable compared to a distributed cache.

Hot Rod clients do not know the anchor owner

Hot Rod clients cannot use the topology information from the servers to locate the anchor owner. Instead, the server receiving a Hot Rod get request must make an additional request to the anchor owner in order to retrieve the value.

19.4. Configuration

The module is still very young and does not yet support many Infinispan features.

Eventually, if it proves useful, it may become another cache mode, just like scattered caches. For now, configuring a cache with anchored keys requires a replicated cache with a custom element `anchored-keys`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
      xmlns="urn:infinispan:config:12.0"
      xmlns:anchored="urn:infinispan:config:anchored:12.0"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:12.0
            https://infinispan.org/schemas/infinispan-config-12.0.xsd
            urn:infinispan:config:anchored:12.0
            https://infinispan.org/schemas/infinispan-anchored-config-12.0.xsd">

    <cache-container default-cache="default">
        <transport/>
        <replicated-cache name="default">
            <anchored:anchored-keys/>
        </replicated-cache>
    </cache-container>

</infinispan>
```

When the `<anchored-keys/>` element is present, the module automatically enables anchored keys and makes some required configuration changes:

- Disables `await-initial-transfer`
- Enables conflict resolution with the equivalent of

  `<partition-handling when-split="ALLOW_READ_WRITES" merge-policy="PREFER_NON_NULL"/>`

The cache will fail to start if these attributes are explicitly set to other values, if state transfer is disabled, or if transactions are enabled.

19.5. Implementation status

Basic operations are implemented: `put`, `putIfAbsent`, `get`, `replace`, `remove`, `putAll`, `getAll`.

19.5.1. Functional commands

The `FunctionalMap` API is not implemented.

Other operations that rely on the functional API's implementation do not work either: `merge`, `compute`, `computeIfPresent`, `computeIfAbsent`.

19.5.2. Partition handling

When a node crashes, surviving nodes do not remove anchor references pointing to that node. In theory, this could allow merges to skip conflict resolution, but currently the `PREFERRED_NON_NULL` merge policy is configured automatically and cannot be changed.

19.5.3. Listeners

Cluster listeners and client listeners are implemented and receive the correct notifications.

Non-clustered embedded listeners currently receive notifications on all the nodes, not just the node where the value is stored.

19.6. Performance considerations

19.6.1. Client/Server Latency

The client always contacts the primary owner, so any read has a `(N-1)/N` probability of requiring a unicast RPC from the primary to the anchor owner.

Writes require the primary to send the value to one node and the anchor address to all the other nodes, which is currently done with `N-1` unicast RPCs.

In theory we could send in parallel one unicast RPC for the value and one multicast RPC for the address, but that would need additional logic to ignore the address on the anchor owner and with TCP multicast RPCs are implemented as parallel unicasts anyway.

19.6.2. Memory overhead

Compared to a distributed cache with one owner, an anchored-keys cache contains copies of all the keys and their locations, plus the overhead of the cache itself.

Therefore, a node with anchored-keys caches should stop accepting new entries when it has less than `(<key size> + <per-key overhead>) * <number of entries not yet inserted>` bytes available.

> The number of entries not yet inserted is obviously very hard to estimate. In the future we may provide a way to limit the overhead of key location information, e.g. by using a distributed cache.

The per-key overhead is lowest for off-heap storage, around 63 bytes: 8 bytes for the entry reference in `MemoryAddressHash.memory`, 29 bytes for the off-heap entry header, and 26 bytes for the serialized `RemoteMetadata` with the owner's address.

The per-key overhead of the ConcurrentHashMap-based on-heap cache, assuming a 64-bit JVM with compressed OOPS, would be around 92 bytes: 32 bytes for `ConcurrentHashMap.Node`, 32 bytes for `MetadataImmortalCacheEntry`, 24 bytes for `RemoteMetadata`, and 4 bytes in the `ConcurrentHashMap.table` array.

19.6.3. State transfer

State transfer does not transfer values, only keys and anchor owner information.

Assuming that the values are much bigger compared to the keys, state transfer for an anchored keys cache should also be much faster compared to the state transfer of a distributed cache of a similar size. But for small values, there may not be a visible improvement.

The initial state transfer does not block a joiner from starting, because it will just ask another node for the anchor owner. However, the remote lookups can be expensive, especially in embedded mode, but also in server mode, if the client is not `HASH_DISTRIBUTION_AWARE`.

Chapter 20. Custom Interceptors

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

Custom interceptors are a way of extending Infinispan 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.

20.1. 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.

```
<local-cache name="cacheWithCustomInterceptors">
      <!--
      Define custom interceptors.  All custom interceptors need to extend
org.jboss.cache.interceptors.base.CommandInterceptor
      -->
      <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>
```

20.2. Adding custom interceptors programatically

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

```
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.

20.3. 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.

Chapter 21. Extending Infinispan

Infinispan can be extended to provide the ability for an end user to add additional configurations, operations and components outside of the scope of the ones normally provided by Infinispan.

21.1. Custom Commands

Infinispan makes use of a command/visitor pattern to implement the various top-level methods you see on the public-facing API.

While the core commands - and their corresponding visitors - are hard-coded as a part of Infinispan's core module, module authors can extend and enhance Infinispan by creating new custom commands.

As a module author (such as infinispan-query, etc.) you can define your own commands.

You do so by:

1. Create a `META-INF/services/org.infinispan.commands.module.ModuleCommandExtensions` file and ensure this is packaged in your jar.
2. Implementing `ModuleCommandFactory` and `ModuleCommandExtensions`
3. Specifying the fully-qualified class name of the `ModuleCommandExtensions` implementation in `META-INF/services/org.infinispan.commands.module.ModuleCommandExtensions`.
4. Implement your custom commands and visitors for these commands

21.1.1. An Example

Here is an example of an `META-INF/services/org.infinispan.commands.module.ModuleCommandExtensions` file, configured accordingly:

org.infinispan.commands.module.ModuleCommandExtensions

```
org.infinispan.query.QueryModuleCommandExtensions
```

21.1.2. Preassigned Custom Command Id Ranges

This is the list of `Command` identifiers that are used by Infinispan based modules or frameworks. Infinispan users should avoid using ids within these ranges. (RANGES to be finalised yet!) Being this a single byte, ranges can't be too large.

Infinispan Query:	100 - 119
Hibernate Search:	120 - 139
Hot Rod Server:	140 - 141

21.2. Extending the configuration builders and parsers

If your custom module requires configuration, it is possible to enhance Infinispan’s configuration builders and parsers. Look at the custom module tests for a detail example on how to implement this.