Administration Guide for Infinispan 10.0

Table of Contents

Chapter 1. Configuring Eviction and Expiration

1.1. Eviction and Data Container

Infinispan supports eviction of entries, such that you do not run out of memory. Eviction is typically used in conjunction with a cache store, so that entries are not permanently lost when evicted, since eviction only removes entries from memory and not from cache stores or the rest of the cluster.

Infinispan supports storing data in a few different formats. Data can be stored as the object iself, binary as a byte[], and off-heap which stores the byte[] in native memory.

Passivation is also a popular option when using eviction, so that only a single copy of an entry is maintained - either in memory or in a cache store, but not both. The main benefit of using passivation over a regular cache store is that updates to entries which exist in memory are cheaper since the update doesn't need to be made to the cache store as well.

Eviction occurs on a *local* basis, and is not cluster-wide. Each node runs an eviction thread to analyse the contents of its in-memory container and decide what to evict. Eviction does not take into account the amount of free memory in the JVM as threshold to starts evicting entries. You have to set `size` attribute of the eviction element to be greater than zero in order for eviction to be turned on. If size is too large you can run out of memory. The `size` attribute will probably take some tuning in each use case.

1.2. Enabling Eviction

Eviction is configured by adding the `<memory />` element to your `<*-cache />` configuration sections or using MemoryConfigurationBuilder API programmatic approach.

All cache entry are evicted by piggybacking on user threads that are hitting the cache.

1.2.1. Eviction strategy

Strategies control how the eviction is handled.

The possible choices are

`NONE`
Eviction is not enabled and it is assumed that the user will not invoke evict directly on the cache. If passivation is enabled this will cause aa warning message to be emitted. This is the default strategy.

`MANUAL`
This strategy is just like NONE except that it asssumes the user will be invoking evict directly. This way if passivation is enabled no warning message is logged.

`REMOVE`

This strategy will actually evict "old" entries to make room for incoming ones.

Eviction is handled by Caffeine utilizing the TinyLFU algorithm with an additional admission window. This was chosen as provides high hit rate while also requiring low memory overhead. This provides a better hit ratio than LRU while also requiring less memory than LIRS.

`EXCEPTION`

This strategy actually prevents new entries from being created by throwing a `ContainerFullException`. This strategy only works with transactional caches that always run with 2 phase commit, that is no 1 phase commit or synchronization optimizations allowed.

1.2.2. Eviction types

Eviction type applies only when the size is set to something greater than 0. The eviction type below determines when the container will decide to remove entries.

`COUNT`

This type of eviction will remove entries based on how many there are in the cache. Once the count of entries has grown larger than the `size` then an entry will be removed to make room.

`MEMORY`

This type of eviction will estimate how much each entry will take up in memory and will remove an entry when the total size of all entries is larger than the configured `size`. This type does not work with `OBJECT` storage type below.

1.2.3. Storage type

Infinispan allows the user to configure in what form their data is stored. Each form supports the same features of Infinispan, however eviction can be limited for some forms. There are currently three storage formats that Infinispan provides, they are:

`OBJECT`

Stores the keys and values as objects in the Java heap Only `COUNT` eviction type is supported.

`BINARY`

Stores the keys and values as a byte[] in the Java heap. This will use the configured marshaller for the cache if there is one. Both `COUNT` and `MEMORY` eviction types are supported.

`OFF-HEAP`

Stores the keys and values in native memory outside of the Java heap as bytes. The configured marshaller will be used if the cache has one. Both `COUNT` and `MEMORY` eviction types are supported.

> Both `BINARY` and `OFF-HEAP` violate equality and hashCode that they are dictated by the resulting byte[] they generate instead of the object instance.

1.2.4. More defaults

By default when no `<memory />` element is specified, no eviction takes place, `OBJECT` storage type is used, and a strategy of `NONE` is assumed.

In case there is an memory element, this table describes the behaviour of eviction based on information provided in the xml configuration ("-" in Supplied size or Supplied strategy column means that the attribute wasn't supplied)

Supplied size	Example	Eviction behaviour
-	<memory />	no eviction as an object
-	<memory> <object strategy="MANUAL" /> </memory>	no eviction as an object and won't log warning if passivation is enabled
> 0	<memory> <object size="100" /> </memory>	eviction takes place and stored as objects
> 0	<memory> <binary size="100" eviction="MEMORY"/> </memory>	eviction takes place and stored as a binary removing to make sure memory doens't go higher than 100
> 0	<memory> <off-heap size="100" /> </memory>	eviction takes place and stored in off-heap
> 0	<memory> <off-heap size="100" strategy="EXCEPTION" /> </memory>	entries are stored in off-heap and if 100 entries are in container exceptions will be thrown for additional
0	<memory> <object size="0" /> </memory>	no eviction
< 0	<memory> <object size="-1" /> </memory>	no eviction

1.3. Expiration

Similar to, but unlike eviction, is expiration. Expiration allows you to attach lifespan and/or maximum idle times to entries. Entries that exceed these times are treated as invalid and are removed. When removed expired entries are not passivated like evicted entries (if passivation is turned on).

> Unlike eviction, expired entries are removed globally - from memory, cache stores, and cluster-wide.

By default entries created are immortal and do not have a lifespan or maximum idle time. Using the cache API, mortal entries can be created with lifespans and/or maximum idle times. Further, default lifespans and/or maximum idle times can be configured by adding the <expiration /> element to your <*-cache /> configuration sections.

When an entry expires it resides in the data container or cache store until it is accessed again by a user request. An expiration reaper is also available to check for expired entries and remove them at a configurable interval of milliseconds.

You can enable the expiration reaper declaratively with the `reaper-interval` attribute or programmatically with the `enableReaper` method in the `ExpirationConfigurationBuilder` class.

- The expiration reaper cannot be disabled when a cache store is present.
- When using a maximum idle time in a clustered cache, you should always enable the expiration reaper. For more information, see Clustered Max Idle.

1.3.1. Difference between Eviction and Expiration

Both Eviction and Expiration are means of cleaning the cache of unused entries and thus guarding the heap against `OutOfMemory` exceptions, so now a brief explanation of the difference.

With *eviction* you set *maximal number of entries* you want to keep in the cache and if this limit is exceeded, some candidates are found to be removed according to a choosen *eviction strategy* (LRU, LIRS, etc…). Eviction can be setup to work with passivation, which is eviction to a cache store.

With *expiration* you set *time criteria* for entries to specify *how long you want to keep them* in the cache.

lifespan
: Specifies how long entries can remain in the cache before they expire. The default value is `-1`, which is unlimited time.

maximum idle time
: Specifies how long entries can remain idle before they expire. An entry in the cache is idle when no operation is performed with the key. The default value is `-1`, which is unlimited time.

1.3.2. Expiration details

1. *Expiration* is a top-level construct, represented in the configuration as well as in the cache API.
2. While eviction is *local to each cache instance* , expiration is *cluster-wide* . Expiration `lifespan` and `maxIdle` values are replicated along with the cache entry.
3. Maximum idle times for cache entries require additional network messages in clustered environments. For this reason, setting `maxIdle` in clustered caches can result in slower operation times.
4. Expiration lifespan and `maxIdle` are also persisted in CacheStores, so this information survives eviction/passivation.

Maximum Idle Expiration

Maximum idle expiration has different behavior in local and clustered cache environments.

Local Max Idle

In non-clustered cache environments, the `maxIdle` configuration expires entries when:

- accessed directly (`Cache.get`).
- iterated upon (`Cache.size`).
- the expiration reaper thread runs.

Clustered Max Idle

In clustered environments, nodes in the cluster can have different access times for the same entry. Entries do not expire from the cache until they reach the maxium idle time for all owners across the cluster.

When a node detects that an entry has reached the maximum idle time and is expired, the node gets the last time that the entry was accessed from the other owners in the cluster. If the other owners indicate that the entry is expired, that entry is not returned to the requester and removed from the cache.

The following points apply to using the `maxIdle` configuration with clustered caches:

- If one or more owner in the cluster detects that an entry is not expired, then a `Cache.get` operation returns the entry. The last access time for that entry is also updated to the current time.
- When the expiration reaper finds entries that might be expired with the maximum idle time, all nodes update the last access time for those entries to the most recent access time before the `maxIdle` time. In this way, the reaper prevents invalid expiration of entries.
- Clustered transactional caches do **not** remove entries that are expired with the maximum idle time on `Cache.get` operations. These expired entries are removed with the expiration reaper thread only, otherwise deadlocking can occur.
- Iteration across a clustered cache returns entries that might be expired with the maximum idle time. This behavior ensures performance because no remote invocations are performed during the iteration. However this does not refresh any expired entries, which are removed by the expiration reaper or when accessed directly (`Cache.get`).

> - Clustered caches should always use the expiration reaper with the `maxIdle` configuration.
> - When using `maxIdle` expiration with exception-based eviction, entries that are expired but not removed from the cache count towards the size of the data container.

Configuration

Eviction and Expiration may be configured using the programmatic or declarative XML configuration. This configuration is on a per-cache basis. Valid eviction/expiration-related configuration elements are:

```xml
<!-- Eviction -->
<memory>
   <object size="2000"/>
</memory>
<!-- Expiration -->
<expiration lifespan="1000" max-idle="500" interval="1000" />
```

Programmatically, the same would be defined using:

```java
Configuration c = new ConfigurationBuilder()
               .memory().size(2000)
               .expiration().wakeUpInterval(5000l).lifespan(1000l).maxIdle(500l)
               .build();
```

Memory Based Eviction Configuration

Memory based eviction may require some additional configuration options if you are using your own custom types (as Infinispan is normally used). In this case Infinispan cannot estimate the memory usage of your classes and as such you are required to use `storeAsBinary` when memory based eviction is used.

```xml
<!-- Enable memory based eviction with 1 GB/> -->
<memory>
   <binary size="1000000000" eviction="MEMORY"/>
</memory>
```

```java
Configuration c = new ConfigurationBuilder()
               .memory()
               .storageType(StorageType.BINARY)
               .evictionType(EvictionType.MEMORY)
               .size(1_000_000_000)
               .build();
```

Default values

Eviction is disabled by default. Default values are used:

- size: -1 is used if not specified, which means unlimited entries.
- 0 means no entries, and the eviction thread will strive to keep the cache empty.

Expiration lifespan and maxIdle both default to -1, which means that entries will be created immortal by default. This can be overridden per entry with the API.

Using expiration

Expiration allows you to set either a lifespan or a maximum idle time on each key/value pair stored in the cache. This can either be set cache-wide using the configuration, as described above, or it can be defined per-key/value pair using the Cache interface. Any values defined per key/value pair overrides the cache-wide default for the specific entry in question.

For example, assume the following configuration:

```xml
<expiration lifespan="1000" />
```

```java
// this entry will expire in 1000 millis
cache.put("pinot noir", pinotNoirPrice);

// this entry will expire in 2000 millis
cache.put("chardonnay", chardonnayPrice, 2, TimeUnit.SECONDS);

// this entry will expire 1000 millis after it is last accessed
cache.put("pinot grigio", pinotGrigioPrice, -1,
          TimeUnit.SECONDS, 1, TimeUnit.SECONDS);

// this entry will expire 1000 millis after it is last accessed, or
// in 5000 millis, which ever triggers first
cache.put("riesling", rieslingPrice, 5,
          TimeUnit.SECONDS, 1, TimeUnit.SECONDS);
```

1.3.3. Expiration designs

Central to expiration is an ExpirationManager.

The purpose of the ExpirationManager is to drive the expiration thread which periodically purges items from the DataContainer. If the expiration thread is disabled (wakeupInterval set to -1) expiration can be kicked off manually using ExprationManager.processExpiration(), for example from another maintenance thread that may run periodically in your application.

The expiration manager processes expirations in the following manner:

1. Causes the data container to purge expired entries
2. Causes cache stores (if any) to purge expired entries

Chapter 2. Persistence

Persistence allows configuring external (persistent) storage engines complementary to the default in memory storage offered by Infinispan. An external persistent storage might be useful for several reasons:

- Increased Durability. Memory is volatile, so a cache store could increase the life-span of the information store in the cache.
- Write-through. Interpose Infinispan as a caching layer between an application and a (custom) external storage engine.
- Overflow Data. By using eviction and passivation, one can store only the "hot" data in memory and overflow the data that is less frequently used to disk.

The integration with the persistent store is done through the following SPI: CacheLoader, CacheWriter, AdvancedCacheLoader and AdvancedCacheWriter (discussed in the following sections).

These SPIs allow for the following features:

- Alignment with JSR-107. The CacheWriter and CacheLoader interface are similar to the the loader and writer in JSR 107. This should considerably help writing portable stores across JCache compliant vendors.
- Simplified Transaction Integration. All necessary locking is handled by Infinispan automatically and implementations don't have to be concerned with coordinating concurrent access to the store. Even though concurrent writes on the same key are not going to happen (depending locking mode in use), implementors should expect operations on the store to happen from multiple/different threads and code the implementation accordingly.
- Parallel Iteration. It is now possible to iterate over entries in the store with multiple threads in parallel.
- Reduced Serialization. This translates in less CPU usage. The new API exposes the stored entries in serialized format. If an entry is fetched from persistent storage for the sole purpose of being sent remotely, we no longer need to deserialize it (when reading from the store) and serialize it back (when writing to the wire). Now we can write to the wire the serialized format as read from the storage directly.

2.1. Configuration

Stores (readers and/or writers) can be configured in a chain. Cache read operation looks at all of the specified `CacheLoader` s, in the order they are configured, until it finds a valid and non-null element of data. When performing writes all cache `CacheWriter` s are written to, except if the `ignoreModifications` element has been set to true for a specific cache writer.

> *Implementing both a CacheWriter and CacheLoader*
>
> Store providers should implement both the `CacheWriter` and the `CacheLoader` interfaces. Stores that do this are considered both for reading and writing (assuming `read-only=false`) data.

This is the configuration of a custom (not shipped with infinispan) store:

```
<local-cache name="myCustomStore">
   <persistence passivation="false">
      <store
         class="org.acme.CustomStore"
         fetch-state="false" preload="true" shared="false"
         purge="true" read-only="false" segmented="true">

         <write-behind modification-queue-size="123" thread-pool-size="23" />

         <property name="myProp">${system.property}</property>
      </store>
   </persistence>
</local-cache>
```

Parameters that you can use to configure persistence are as follows:

`connection-attempts`
: Sets the maximum number of attempts to start each configured CacheWriter/CacheLoader. If the attempts to start are not successful, an exception is thrown and the cache does not start.

`connection-interval`
: Specifies the time, in milliseconds, to wait between connection attempts on startup. A negative or zero value means no wait between connection attempts.

`availability-interval`
: Specifies the time, in milliseconds, between availability checks to determine if the PersistenceManager is available. In other words, this interval sets how often stores/loaders are polled via their `org.infinispan.persistence.spi.CacheWriter#isAvailable` or `org.infinispan.persistence.spi.CacheLoader#isAvailable` implementation. If a single store/loader is not available, an exception is thrown during cache operations.

`passivation`
: Enables passivation. The default value is `false` (boolean).

  This property has a significant impact on Infinispan interactions with the loaders. See Cache Passivation for more information.

`class`
: Defines the class of the store and must implement `CacheLoader`, `CacheWriter`, or both.

`fetch-state`
: Fetches the persistent state of a cache when joining a cluster. The default value is `false`

(boolean).

The purpose of this property is to retrieve the persistent state of a cache and apply it to the local cache store of a node when it joins a cluster. Fetching the persistent state does not apply if a cache store is shared because it accesses the same data as the other stores.

This property can be `true` for one configured cache loader only. If more than one cache loader fetches the persistent state, a configuration exception is thrown when the cache service starts.

`preload`
: Pre-loads data into memory from the cache loader when the cache starts. The default value is `false` (boolean).

This property is useful when data in the cache loader is required immediately after startup to prevent delays with cache operations when the data is loaded lazily. This property can provide a "warm cache" on startup but it impacts performance because it affects start time.

Pre-loading data is done locally, so any data loaded is stored locally in the node only. The pre-loaded data is not replicated or distributed. Likewise, Infinispan pre-loads data only up to the maximum configured number of entries in eviction.

`shared`
: Determines if the cache loader is shared between cache instances. The default value is `false` (boolean).

This property prevents duplicate writes of data to the cache loader by different cache instances. An example is where all cache instances in a cluster use the same JDBC settings for the same remote, shared database.

`segmented`
: Configures a cache store to segment data. The default value is `false` (boolean).

If `true` the cache store stores data in buckets. The `hash.numSegments` property configures how many buckets there are for storing data.

Depending on the cache store implementation, segmenting data can cause slower write operations. However, performance improves for other cache operations. See Segmented Stores for more information.

`read-only`
: Prevents new data from being persisted to the cache store. The default value is `false` (boolean).

`purge`
: Empties the specified cache loader at startup. The default value is `false` (boolean). This property takes effect only if the `read-only` property is set to `false`.

`max-batch-size`
: Sets the maximum size of a batch to insert of delete from the cache store. The default value is #{AbstractStore-maxBatchSize}.

If the value is less than `1`, no upper limit applies to the number of operations in a batch.

`write-behind`

Asynchronously persists data to the cache store. The default value is `false` (boolean). See Asynchronous Write-Behind for more information.

> You can define additional attributes in the `properties` section to configure specific aspects of each cache loader, such as the `myProp` attribute in the previous example.
>
> Other cache loaders with more complex configurations also include additional properties. See the following JDBC cache store configuration for examples.

The preceding configuration applies a generic cache store implementation. However, the default Infinispan store implementation has a more complex configuration schema, in which the `properties` section is replaced with XML attributes:

```xml
<persistence passivation="false">
   <!-- note that class is missing and is induced by the fileStore element name -->
   <file-store
           shared="false" preload="true"
           fetch-state="true"
           read-only="false"
           purge="false"
           path="${java.io.tmpdir}">
      <write-behind thread-pool-size="5" />
   </file-store>
</persistence>
```

The same configuration can be achieved programmatically:

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

2.2. Cache Passivation

A CacheWriter can be used to enforce entry passivation and activation on eviction in a cache. Cache passivation is the process of removing an object from in-memory cache and writing it to a secondary data store (e.g., file system, database) on eviction. Cache activation is the process of restoring an object from the data store into the in-memory cache when it's needed to be used. In order to fully support passivation, a store needs to be both a CacheWriter and a CacheLoader. In both cases, the configured cache store is used to read from the loader and write to the data writer.

When an eviction policy in effect evicts an entry from the cache, if passivation is enabled, a notification that the entry is being passivated will be emitted to the cache listeners and the entry will be stored. When a user attempts to retrieve a entry that was evicted earlier, the entry is (lazily) loaded from the cache loader into memory. When the entry has been loaded a notification is emitted to the cache listeners that the entry has been activated. In order to enable passivation just set passivation to true (false by default). When passivation is used, only the first cache loader configured is used and all others are ignored.

2.2.1. Limitations

Due to the unique nature of passivation, it is not supported with some other store configurations.

- Transactional store - Passivation writes/removes entries from the store outside the scope of the actual Infinispan commit boundaries.
- Shared store - Shared store requires entries always being in the store for other owners. Thus passivation makes no sense as we can't remove the entry from the store.

2.2.2. Cache Loader Behavior with Passivation Disabled vs Enabled

When passivation is disabled, whenever an element is modified, added or removed, then that modification is persisted in the backend store via the cache loader. There is no direct relationship between eviction and cache loading. If you don't use eviction, what's in the persistent store is basically a copy of what's in memory. If you do use eviction, what's in the persistent store is basically a superset of what's in memory (i.e. it includes entries that have been evicted from memory). When passivation is enabled, and with an unshared store, there is a direct relationship between eviction and the cache loader. Writes to the persistent store via the cache loader only occur as part of the eviction process. Data is deleted from the persistent store when the application reads it back into memory. In this case, what's in memory and what's in the persistent store are two subsets of the total information set, with no intersection between the subsets. With a shared store, entries which have been passivated in the past will continue to exist in the store, although they may have a stale value if this has been overwritten in memory.

The following is a simple example, showing what state is in RAM and in the persistent store after each step of a 6 step process:

Operation	Passivation Off	Passivation On, Shared Off	Passivation On, Shared On
Insert keyOne	**Memory:** keyOne **Disk:** keyOne	**Memory:** keyOne **Disk:** (none)	**Memory:** keyOne **Disk:** (none)
Insert keyTwo	**Memory:** keyOne, keyTwo **Disk:** keyOne, keyTwo	**Memory:** keyOne, keyTwo **Disk:** (none)	**Memory:** keyOne, keyTwo **Disk:** (none)
Eviction thread runs, evicts keyOne	**Memory:** keyTwo **Disk:** keyOne, keyTwo	**Memory:** keyTwo **Disk:** keyOne	**Memory:** keyTwo **Disk:** keyOne

Operation	Passivation Off	Passivation On, Shared Off	Passivation On, Shared On
Read keyOne	**Memory:** keyOne, keyTwo **Disk:** keyOne, keyTwo	**Memory:** keyOne, keyTwo **Disk:** (none)	**Memory:** keyOne, keyTwo **Disk:** keyOne
Eviction thread runs, evicts keyTwo	**Memory:** keyOne **Disk:** keyOne, keyTwo	**Memory:** keyOne **Disk:** keyTwo	**Memory:** keyOne **Disk:** keyOne, keyTwo
Remove keyTwo	**Memory:** keyOne **Disk:** keyOne	**Memory:** keyOne **Disk:** (none)	**Memory:** keyOne **Disk:** keyOne

2.3. Cache Loaders and transactional caches

When a cache is transactional and a cache loader is present, the cache loader won't be enlisted in the transaction in which the cache is part. That means that it is possible to have inconsistencies at cache loader level: the transaction to succeed applying the in-memory state but (partially) fail applying the changes to the store. Manual recovery would not work with caches stores.

2.4. Write-Through And Write-Behind Caching

Infinispan can optionally be configured with one or several cache stores allowing it to store data in a persistent location such as shared JDBC database, a local filesystem, etc. Infinispan can handle updates to the cache store in two different ways:

- Write-Through (Synchronous)
- Write-Behind (Asynchronous)

2.4.1. Write-Through (Synchronous)

In this mode, which is supported in version 4.0, when clients update a cache entry, i.e. via a Cache.put() invocation, the call will not return until Infinispan has gone to the underlying cache store and has updated it. Normally, this means that updates to the cache store are done within the boundaries of the client thread.

The main advantage of this mode is that the cache store is updated at the same time as the cache, hence the cache store is consistent with the cache contents. On the other hand, using this mode reduces performance because the latency of having to access and update the cache store directly impacts the duration of the cache operation.

Configuring a write-through or synchronous cache store does not require any particular configuration option. By default, unless marked explicitly as write-behind or asynchronous, all cache stores are write-through or synchronous. Please find below a sample configuration file of a write-through unshared local file cache store:

```xml
<persistence passivation="false">
   <file-store fetch-state="true"
               read-only="false"
               purge="false" path="${java.io.tmpdir}"/>
</persistence>
```

2.4.2. Write-Behind (Asynchronous)

In this mode, updates to the cache are asynchronously written to the cache store. Infinispan puts pending changes into a modification queue so that it can quickly store changes.

The configured number of threads consume the queue and apply the modifications to the underlying cache store. If the configured number of threads cannot consume the modifications fast enough, or if the underlying store becomes unavailable, the modification queue becomes full. In this event, the cache store becomes write-through until the queue can accept new entries.

This mode provides an advantage in that cache operations are not affected by updates to the underlying store. However, because updates happen asynchronously, there is a period of time during which data in the cache store is inconsistent with data in the cache.

The write-behind strategy is suitable for cache stores with low latency and small operational cost; for example, an unshared file-based cache store that is local to the cache itself. In this case, the time during which data is inconsistent between the cache store and the cache is reduced to the lowest possible period.

The following is an example configuration for the write-behind strategy:

```xml
<persistence passivation="false">
   <file-store fetch-state="true"
               read-only="false"
               purge="false" path="${java.io.tmpdir}">
   <!-- start write-behind configuration -->
   <write-behind modification-queue-size="123" thread-pool-size="23" />
   <!-- end write-behind configuration -->
   </file-store>
</persistence>
```

2.4.3. Segmented Stores

You can configure stores so that data resides in segments to which keys map. See Key Ownership for more information about segments and ownership.

Segmented stores increase read performance for bulk operations; for example, streaming over data (`Cache.size`, `Cache.entrySet.stream`), pre-loading the cache, and doing state transfer operations.

However, segmented stores can also result in loss of performance for write operations. This performance loss applies particularly to batch write operations that can take place with transactions or write-behind stores. For this reason, you should evaluate the overhead for write operations before you enable segmented stores. The performance gain for bulk read operations might not be acceptable if there is a significant performance loss for write operations.

> Loss of data can occur if the number of segments in a cache store are not changed gracefully. For this reason, if you change the `numSegments` setting in the store configuration, you must migrate the existing store to use the new configuration.
>
> The recommended method to migrate the cache store configuration is to perform a rolling upgrade. The store migrator supports migrating a non-segmented cache store to a segmented cache store only. The store migrator does not currently support migrating from a segmented cache store.

> Not all cache stores support segmentation. See the appropriate section for each store to determine if it supports segmentation.
>
> If you plan to convert or write a new store to support segmentation, see the following SPI section that provides more details.

2.5. Filesystem based cache stores

A filesystem-based cache store is typically used when you want to have a cache with a cache store available locally which stores data that has overflowed from memory, having exceeded size and/or time restrictions.

> Usage of filesystem-based cache stores on shared filesystems like NFS, Windows shares, etc. should be avoided as these do not implement proper file locking and can cause data corruption. File systems are inherently not transactional, so when attempting to use your cache in a transactional context, failures when writing to the file (which happens during the commit phase) cannot be recovered.

2.6. Single File Store

The single file cache store keeps all data in a single file. The way it looks up data is by keeping an in-memory index of keys and the positions of their values in this file. This results in greater performance compared to old file cache store. There is one caveat though. Since the single file based cache store keeps keys in memory, it can lead to increased memory consumption, and hence it’s not recommended for caches with big keys.

In certain use cases, this cache store suffers from fragmentation: if you store larger and larger values, the space is not reused and instead the entry is appended at the end of the file. The space (now empty) is reused only if you write another entry that can fit there. Also, when you remove all entries from the cache, the file won’t shrink, and neither will be de-fragmented.

These are the available configuration options for the single file cache store:

- `path` where data will be stored. (e.g., `path="/tmp/myDataStore"`). By default, the location is `Infinispan-SingleFileStore`.

- `max-entries` specifies the maximum number of entries to keep in this file store. As mentioned before, in order to speed up lookups, the single file cache store keeps an index of keys and their corresponding position in the file. To avoid this index resulting in memory consumption problems, this cache store can bounded by a maximum number of entries that it stores. If this limit is exceeded, entries are removed permanently using the LRU algorithm both from the in-memory index and the underlying file based cache store. So, setting a maximum limit only makes sense when Infinispan is used as a cache, whose contents can be recomputed or they can be retrieved from the authoritative data store. If this maximum limit is set when the Infinispan is used as an authoritative data store, it could lead to data loss, and hence it's not recommended for this use case. The default value is `-1` which means that the file store size is unlimited.

2.6.1. Segmentation support

The single file cache store supports segmentation and creates a separate instance per segment. Segmentation results in multiple directories under the configured directory, where each directory is a number that represents the segment to which the data maps.

2.6.2. Configuration

The following examples show single file cache store configuration:

```
<persistence>
   <file-store path="/tmp/myDataStore" max-entries="5000"/>
</persistence>
```

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
    .addSingleFileStore()
    .location("/tmp/myDataStore")
    .maxEntries(5000);
```

2.7. Soft-Index File Store

The Soft-Index File Store is an experimental local file-based. It is a pure Java implementation that tries to get around Single File Store's drawbacks by implementing a variant of B+ tree that is cached in-memory using Java's soft references - here's where the name Soft-Index File Store comes from. This B+ tree (called Index) is offloaded on filesystem to single file that does not need to be persisted - it is purged and rebuilt when the cache store restarts, its purpose is only offloading.

The data that should be persisted are stored in a set of files that are written in append-only way - that means that if you store this on conventional magnetic disk, it does not have to seek when writing a burst of entries. It is not stored in single file but set of files. When the usage of any of these files drops below 50% (the entries from the file are overwritten to another file), the file starts to be collected, moving the live entries into different file and in the end removing that file from disk.

Most of the structures in Soft Index File Store are bounded, therefore you don't have to be afraid of OOMEs. For example, you can configure the limits for concurrently open files as well.

2.7.1. Segmentation support

The Soft-Index file store supports segmentation and creates a separate instance per segment. Segmentation results in multiple directories under the configured directory, where each directory is a number that represents the segment to which the data maps.

2.7.2. Configuration

Here is an example of Soft-Index File Store configuration via XML:

```
<persistence>
    <soft-index-file-store xmlns="urn:infinispan:config:store:soft-index:8.0">
        <index path="/tmp/sifs/testCache/index" />
        <data path="/tmp/sifs/testCache/data" />
    </soft-index-file-store>
</persistence>
```

Programmatic configuration would look as follows:

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
    .addStore(SoftIndexFileStoreConfigurationBuilder.class)
        .indexLocation("/tmp/sifs/testCache/index");
        .dataLocation("/tmp/sifs/testCache/data")
```

2.7.3. Current limitations

Size of a node in the Index is limited, by default it is 4096 bytes, though it can be configured. This size also limits the key length (or rather the length of the serialized form): you can't use keys longer than size of the node - 15 bytes. Moreover, the key length is stored as 'short', limiting it to 32767 bytes. There's no way how you can use longer keys - SIFS throws an exception when the key is longer after serialization.

When entries are stored with expiration, SIFS cannot detect that some of those entries are expired. Therefore, such old file will not be compacted (method AdvancedStore.purgeExpired() is not implemented). This can lead to excessive file-system space usage.

2.8. JDBC String based Cache Store

A cache store which relies on the provided JDBC driver to load/store values in the underlying database.

Each key in the cache is stored in its own row in the database. In order to store each key in its own row, this store relies on a (pluggable) bijection that maps the each key to a String object. The bijection is defined by the Key2StringMapper interface. Infinispans ships a default implementation (smartly named DefaultTwoWayKey2StringMapper) that knows how to handle primitive types.

> By default Infinispan shares are not stored, meaning that all nodes in the cluster will write to the underlying store upon each update. If you wish for an operation to only be written to the underlying database once, you must configure the JDBC store to be shared.

> The JDBC string-based cache store does not support segmentation. Support will be available in a future release.

2.8.1. Connection management (pooling)

In order to obtain a connection to the database the JDBC cache store relies on a ConnectionFactory implementation. The connection factory is specified programmatically using one of the connectionPool(), dataSource() or simpleConnection() methods on the JdbcStringBasedStoreConfigurationBuilder class or declaratively using one of the `<connectionPool />`, `<dataSource />` or `<simpleConnection />` elements.

Infinispan ships with three ConnectionFactory implementations:

- PooledConnectionFactoryConfigurationBuilder is a factory based on Agroal, which is configured via the PooledConnectionFactoryConfiguration or by specifying a properties file via `PooledConnectionFactoryConfiguration.propertyFile`. Properties must be specified with the prefix "org.infinispan.agroal.". An example `agroal.properties` file is shown below:

```
org.infinispan.agroal.metricsEnabled=false

org.infinispan.agroal.minSize=10
org.infinispan.agroal.maxSize=100
org.infinispan.agroal.initialSize=20
org.infinispan.agroal.acquisitionTimeout_s=1
org.infinispan.agroal.validationTimeout_m=1
org.infinispan.agroal.leakTimeout_s=10
org.infinispan.agroal.reapTimeout_m=10

org.infinispan.agroal.metricsEnabled=false
org.infinispan.agroal.autoCommit=true
org.infinispan.agroal.jdbcTransactionIsolation=READ_COMMITTED
org.infinispan.agroal.jdbcUrl=jdbc:h2:mem:PooledConnectionFactoryTest;DB_CLOSE_DELAY=-
1
org.infinispan.agroal.driverClassName=org.h2.Driver.class
org.infinispan.agroal.principal=sa
org.infinispan.agroal.credential=sa
```

- ManagedConnectionFactoryConfigurationBuilder is a connection factory that can be used within managed environments, such as application servers. It knows how to look into the JNDI tree at a certain location (configurable) and delegate connection management to the

DataSource.

- SimpleConnectionFactoryConfigurationBuilder is a factory implementation that will create database connection on a per invocation basis. Not recommended in production.

The `PooledConnectionFactory` is generally recommended for stand-alone deployments (i.e. not running within AS or servlet container). `ManagedConnectionFactory` can be used when running in a managed environment where a `DataSource` is present, so that connection pooling is performed within the `DataSource`.

2.8.2. Sample configurations

Below is a sample configuration for the JdbcStringBasedStore. For detailed description of all the parameters used refer to the JdbcStringBasedStore.

```
<persistence>
   <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:9.2" shared="true"
fetch-state="false" read-only="false" purge="false">
      <connection-pool connection-url=
"jdbc:h2:mem:infinispan_string_based;DB_CLOSE_DELAY=-1" username="sa" driver=
"org.h2.Driver"/>
      <string-keyed-table drop-on-exit="true" create-on-start="true" prefix=
"ISPN_STRING_TABLE">
         <id-column name="ID_COLUMN" type="VARCHAR(255)" />
         <data-column name="DATA_COLUMN" type="BINARY" />
         <timestamp-column name="TIMESTAMP_COLUMN" type="BIGINT" />
      </string-keyed-table>
   </string-keyed-jdbc-store>
</persistence>
```

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

Finally, below is an example of a JDBC cache store with a managed connection factory, which is chosen implicitly by specifying a datasource JNDI location:

```
<string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:9.2" shared="true"
fetch-state="false" read-only="false" purge="false">
   <data-source jndi-url="java:/StringStoreWithManagedConnectionTest/DS" />
   <string-keyed-table drop-on-exit="true" create-on-start="true" prefix=
"ISPN_STRING_TABLE">
      <id-column name="ID_COLUMN" type="VARCHAR(255)" />
      <data-column name="DATA_COLUMN" type="BINARY" />
      <timestamp-column name="TIMESTAMP_COLUMN" type="BIGINT" />
   </string-keyed-table>
</string-keyed-jdbc-store>
```

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
      .fetchPersistentState(false)
      .ignoreModifications(false)
      .purgeOnStartup(false)
      .shared(true)
      .table()
         .dropOnExit(true)
         .createOnStart(true)
         .tableNamePrefix("ISPN_STRING_TABLE")
         .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
         .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
         .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
      .dataSource()
         .jndiUrl("java:/StringStoreWithManagedConnectionTest/DS");
```

Apache Derby users

If you're connecting to an Apache Derby database, make sure you set dataColumnType to BLOB: `<data-column name="DATA_COLUMN" type="BLOB"/>`

2.9. Remote store

The `RemoteStore` is a cache loader and writer implementation that stores data in a remote Infinispan cluster. In order to communicate with the remote cluster, the `RemoteStore` uses the HotRod client/server architecture. HotRod bering the load balancing and fault tolerance of calls and the possibility to fine-tune the connection between the RemoteCacheStore and the actual cluster. Please refer to Hot Rod for more information on the protocol, client and server configuration. For a list of RemoteStore configuration refer to the javadoc . Example:

2.9.1. Segmentation support

The `RemoteStore` store supports segmentation because it can publish keys and entries by segment, allowing for more efficient bulk operations.

Segmentation is only supported when the remote server supports at least protocol version 2.3 or newer.

> ⚠ Ensure the number of segments and hash are the same between the store configured cache and the remote server otherwise bulk operations will not return correct results.

2.9.2. Sample Usage

```
<persistence>
   <remote-store xmlns="urn:infinispan:config:store:remote:8.0" cache="mycache" raw-
values="true">
      <remote-server host="one" port="12111" />
      <remote-server host="two" />
      <connection-pool max-active="10" exhausted-action="CREATE_NEW" />
      <write-behind />
   </remote-store>
</persistence>
```

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence().addStore(RemoteStoreConfigurationBuilder.class)
      .fetchPersistentState(false)
      .ignoreModifications(false)
      .purgeOnStartup(false)
      .remoteCacheName("mycache")
      .rawValues(true)
.addServer()
      .host("one").port(12111)
      .addServer()
      .host("two")
      .connectionPool()
      .maxActive(10)
      .exhaustedAction(ExhaustedAction.CREATE_NEW)
      .async().enable();
```

In this sample configuration, the remote cache store is configured to use the remote cache named "mycache" on servers "one" and "two". It also configures connection pooling and provides a custom transport executor. Additionally the cache store is asynchronous.

2.10. Cluster cache loader

The ClusterCacheLoader is a cache loader implementation that retrieves data from other cluster members.

2.10.1. ClusterCacheLoader

It is a cache loader only as it doesn't persist anything (it is not a Store), therefore features like *fetchPersistentState* (and like) are not applicable.

A cluster cache loader can be used as a non-blocking (partial) alternative to *stateTransfer* : keys not already available in the local node are fetched on-demand from other nodes in the cluster. This is a kind of lazy-loading of the cache content.

The cluster cache loader does not support segmentation.

```
<persistence>
   <cluster-loader remote-timeout="500"/>
</persistence>
```

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
    .addClusterLoader()
    .remoteCallTimeout(500);
```

For a list of ClusterCacheLoader configuration refer to the javadoc .

The ClusterCacheLoader does not support preloading (`preload=true`). It also won't provide state if `fetchPersistentSate=true`.

2.11. Command-Line Interface cache loader

The Command-Line Interface (CLI) cache loader is a cache loader implementation that retrieves data from another Infinispan node using the CLI. The node to which the CLI connects to could be a standalone node, or could be a node that it's part of a cluster. This cache loader is read-only, so it will only be used to retrieve data, and hence, won't be used when persisting data.

2.11.1. CLI Cache Loader

The CLI cache loader is configured with a connection URL pointing to the Infinispan node to which connect to. Here is an example:

The Command-Line Interface (CLI) cache loader does not support segmentation.

```
<persistence>
   <cli-loader connection="jmx://1.2.3.4:4444/MyCacheManager/myCache" />
</persistence>
```

```java
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
    .addStore(CLInterfaceLoaderConfigurationBuilder.class)
    .connectionString("jmx://192.0.2.0:4444/MyCacheManager/myCache");
```

2.12. RocksDB Cache Store

Infinispan supports using RocksDB as a cache store.

2.12.1. Introduction

RocksDB is a fast key-value filesystem-based storage from Facebook. It started as a fork of Google's LevelDB, but provides superior performance and reliability, especially in highly concurrent scenarios.

2.12.2. Segmentation support

The RocksDB cache store supports segmentation and creates a separate column family per segment, which substantially improves lookup performance and iteration. However, write operations are a little slower when the cache store is segmented.

> You should not configure more than a few hundred segments. RocksDB is not designed to have an unlimited number of column families. Too many segments also significantly increases startup time for the cache store.

Sample Usage

The RocksDB cache store requires 2 filesystem directories to be configured - each directory contains a RocksDB database: one location is used to store non-expired data, while the second location is used to store expired keys pending purge.

```java
Configuration cacheConfig = new ConfigurationBuilder().persistence()
                .addStore(RocksDBStoreConfigurationBuilder.class)
                .build();
EmbeddedCacheManager cacheManager = new DefaultCacheManager(cacheConfig);

Cache<String, User> usersCache = cacheManager.getCache("usersCache");
usersCache.put("raytsang", new User(...));
```

2.12.3. Configuration

It is also possible to configure the underlying rocks db instance. This can be done via properties in the store configuration. Any property that is prefixed with the name `database` will configure the rocks db database. Data is now stored in column families, these can be configured independently of the database by setting a property prefixed with the name `data`.

Note that you do not have to supply properties and this is entirely optional.

Sample Programatic Configuration

```
Properties props = new Properties();
props.put("database.max_background_compactions", "2");
props.put("data.write_buffer_size", "512MB");

Configuration cacheConfig = new ConfigurationBuilder().persistence()
                .addStore(RocksDBStoreConfigurationBuilder.class)
                .location("/tmp/rocksdb/data")
                .expiredLocation("/tmp/rocksdb/expired")
        .properties(props)
                .build();
```

Parameter	Description
location	Directory to use for RocksDB to store primary cache store data. The directory will be auto-created if it does not exit.
expiredLocation	Directory to use for RocksDB to store expiring data pending to be purged permanently. The directory will be auto-created if it does not exit.
expiryQueueSize	Size of the in-memory queue to hold expiring entries before it gets flushed into expired RocksDB store
clearThreshold	There are two methods to clear all entries in RocksDB. One method is to iterate through all entries and remove each entry individually. The other method is to delete the database and re-init. For smaller databases, deleting individual entries is faster than the latter method. This configuration sets the max number of entries allowed before using the latter method
compressionType	Configuration for RocksDB for data compression, see CompressionType enum for options
blockSize	Configuration for RocksDB - see documentation for performance tuning
cacheSize	Configuration for RocksDB - see documentation for performance tuning

Sample XML Configuration

infinispan.xml

```xml
<local-cache name="vehicleCache">
   <persistence>
      <rocksdb-store path="/tmp/rocksdb/data">
         <expiration path="/tmp/rocksdb/expired"/>
         <property name="database.max_background_compactions">2</property>
         <property name="data.write_buffer_size">512MB</property>
      </rocksdb-store>
   </persistence>
</local-cache>
```

2.12.4. Additional References

Refer to the test case for code samples in action.

Refer to test configurations for configuration samples.

2.13. JPA Cache Store

The implementation depends on JPA 2.0 specification to access entity meta model.

In normal use cases, it's recommended to leverage Infinispan for JPA second level cache and/or query cache. However, if you'd like to use only Infinispan API and you want Infinispan to persist into a cache store using a common format (e.g., a database with well defined schema), then JPA Cache Store could be right for you.

Things to note

- When using JPA Cache Store, the key should be the ID of the entity, while the value should be the entity object.
- Only a single `@Id` or `@EmbeddedId` annotated property is allowed.
- Auto-generated ID is not supported.
- Lastly, all entries will be stored as immortal entries.

> The JPA cache store does not support segmentation.

2.13.1. Sample Usage

For example, given a persistence unit "myPersistenceUnit", and a JPA entity User:

persistence.xml

```xml
<persistence-unit name="myPersistenceUnit">
    ...
</persistence-unit>
```

User entity class

User.java

```
@Entity
public class User implements Serializable {
    @Id
    private String username;
    private String firstName;
    private String lastName;

    ...
}
```

Then you can configure a cache "usersCache" to use JPA Cache Store, so that when you put data into the cache, the data would be persisted into the database based on JPA configuration.

```
EmbeddedCacheManager cacheManager = ...;

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

Cache<String, User> usersCache = cacheManager.getCache("usersCache");
usersCache.put("raytsang", new User(...));
```

Normally a single Infinispan cache can store multiple types of key/value pairs, for example:

```
Cache<String, User> usersCache = cacheManager.getCache("myCache");
usersCache.put("raytsang", new User());
Cache<Integer, Teacher> teachersCache = cacheManager.getCache("myCache");
teachersCache.put(1, new Teacher());
```

It's important to note that, when a cache is configured to use a JPA Cache Store, that cache would only be able to store ONE type of data.

```
Cache<String, User> usersCache = cacheManager.getCache("myJPACache"); // configured
for User entity class
usersCache.put("raytsang", new User());
Cache<Integer, Teacher> teachersCache = cacheManager.getCache("myJPACache"); // cannot
do this when this cache is configured to use a JPA cache store
teachersCache.put(1, new Teacher());
```

Use of `@EmbeddedId` is supported so that you can also use composite keys.

```java
@Entity
public class Vehicle implements Serializable {
    @EmbeddedId
    private VehicleId id;
    private String color;   ...
}

@Embeddable
public class VehicleId implements Serializable
{
    private String state;
    private String licensePlate;
    ...
}
```

Lastly, auto-generated IDs (e.g., `@GeneratedValue`) is not supported. When putting things into the cache with a JPA cache store, the key should be the ID value!

2.13.2. Configuration

Sample Programmatic Configuration

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

Parameter	Description
persistenceUnitName	JPA persistence unit name in JPA configuration (persistence.xml) that contains the JPA entity class
entityClass	JPA entity class that is expected to be stored in this cache. Only one class is allowed.

Sample XML Configuration

```xml
<local-cache name="vehicleCache">
   <persistence passivation="false">
      <jpa-store xmlns="urn:infinispan:config:store:jpa:7.0"
         persistence-unit="org.infinispan.persistence.jpa.configurationTest"
         entity-class="org.infinispan.persistence.jpa.entity.Vehicle">
      />
   </persistence>
</local-cache>
```

Parameter	Description
persistence-unit	JPA persistence unit name in JPA configuration (persistence.xml) that contains the JPA entity class
entity-class	Fully qualified JPA entity class name that is expected to be stored in this cache. Only one class is allowed.

2.13.3. Additional References

Refer to the test case for code samples in action.

Refer to test configurations for configuration samples.

2.14. Custom Cache Stores

If the provided cache stores do not fulfill all of your requirements, it is possible for you to implement your own store. The steps required to create your own store are as follows:

1. Write your custom store by implementing one of the following interfaces:
   - `org.infinispan.persistence.spi.AdvancedCacheWriter`
   - `org.infinispan.persistence.spi.AdvancedCacheLoader`
   - `org.infinispan.persistence.spi.CacheLoader`
   - `org.infinispan.persistence.spi.CacheWriter`
   - `org.infinispan.persistence.spi.ExternalStore`
   - `org.infinispan.persistence.spi.AdvancedLoadWriteStore`
   - `org.infinispan.persistence.spi.TransactionalCacheWriter`
   - `org.infinispan.persistence.spi.SegmentedAdvancedLoadWriteStore`
2. Annotate your store class with the `@Store` annotation and specify the properties relevant to your store, e.g. is it possible for the store to be shared in Replicated or Distributed mode: `@Store(shared = true)`.
3. Create a custom cache store configuration and builder. This requires extending `AbstractStoreConfiguration` and `AbstractStoreConfigurationBuilder`. As an optional step, you should add the following annotations to your configuration - `@ConfigurationFor`, `@BuiltBy` as well as adding `@ConfiguredBy` to your store implementation class. These additional annotations will ensure that your custom configuration builder is used to parse your store configuration from xml. If these annotations are not added, then the `CustomStoreConfigurationBuilder` will be used to parse the common store attributes defined in `AbstractStoreConfiguration` and any additional elements will be ignored. If a store and its configuration do not declare the `@Store` and `@ConfigurationFor` annotations respectively, a warning message will be logged upon cache initialisation.

   If you wish for your store to be segmented, where it will craete a different store instance per segment, instead of extending `AbstractStoreConfiguration` you should extend `AbstractSegmentedStoreConfiguration`.

4. Add your custom store to your cache's configuration:
   a. Add your custom store to the ConfigurationBuilder, for example:

```
Configuration config = new ConfigurationBuilder()
            .persistence()
            .addStore(CustomStoreConfigurationBuilder.class)
            .build();
```

   b. Define your custom store via xml:

```
<local-cache name="customStoreExample">
  <persistence>
    <store class="org.infinispan.persistence.dummy.DummyInMemoryStore" />
  </persistence>
</local-cache>
```

2.14.1. HotRod Deployment

A Custom Cache Store can be packaged into a separate JAR file and deployed in a HotRod server using the following steps:

1. Follow Custom Cache Stores, steps 1-3>> in the previous section and package your implementations in a JAR file (or use a Custom Cache Store Archetype).
2. In your Jar create a proper file under `META-INF/services/`, which contains the fully qualified class name of your store implementation. The name of this service file should reflect the interface that your store implements. For example, if your store implements the `AdvancedCacheWriter` interface than you need to create the following file:
   - `/META-INF/services/org.infinispan.persistence.spi.AdvancedCacheWriter`
3. Deploy the JAR file in the Infinispan Server.

2.15. Store Migrator

Infinispan 9.0 introduced changes to internal marshalling functionality that are not backwardly compatible with previous versions of Infinispan. As a result, Infinispan 9.x and later cannot read cache stores created in earlier versions of Infinispan. Additionally, Infinispan no longer provides some store implementations such as JDBC Mixed and Binary stores.

You can use `StoreMigrator.java` to migrate cache stores. This migration tool reads data from cache stores in previous versions and rewrites the content for compatibility with the current marshalling implementation.

2.15.1. Migrating Cache Stores

To perform a migration with `StoreMigrator`,

1. Put `infinispan-tools-10.0.jar` and dependencies for your source and target databases, such as JDBC drivers, on your classpath.
2. Create a `.properties` file that contains configuration properties for the source and target cache stores.

   You can find an example properties file that contains all applicable configuration options in migrator.properties.
3. Specify `.properties` file as an argument for `StoreMigrator`.
4. Run `mvn exec:java` to execute the migrator.

See the following example Maven `pom.xml` for `StoreMigrator`:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.infinispan.example</groupId>
    <artifactId>jdbc-migrator-example</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.infinispan</groupId>
            <artifactId>infinispan-tools</artifactId>
            <!-- Replace ${version.infinispan} with the
            version of Infinispan that you're using. -->
            <version>${version.infinispan}</version>
        </dependency>

        <!-- ADD YOUR REQUIRED DEPENDENCIES HERE -->
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>1.2.1</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>java</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <mainClass>StoreMigrator</mainClass>
                    <arguments>
                        <argument><!-- PATH TO YOUR MIGRATOR.PROPERTIES FILE --
></argument>
                    </arguments>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

2.15.2. Store Migrator Properties

All migrator properties are configured within the context of a source or target store. Each property must start with either `source.` or `target.`.

All properties in the following sections apply to both source and target stores, except for `table.binary.*` properties because it is not possible to migrate to a binary table.

Common Properties

Property	Description	Example value	Required
type	JDBC_STRING | JDBC_BINARY | JDBC_MIXED | LEVELDB | ROCKSDB | SINGLE_FILE_STORE | SOFT_INDEX_FILE_STORE	JDBC_MIXED	TRUE
cache_name	The name of the cache associated with the store	persistentMixedCache	TRUE
segment_count	How many segments this store will be created with. If not provided store will not be segmented. (supported as target only - JDBC not yet supported)	null	FALSE

It should be noted that the **segment_count** property should match how many segments your cache will be using. That is that it should match the `clustering.hash.numSegments` config value. If these do not match, data will not be properly read when running the cache.

JDBC Properties

Property	Description	Example value	Required
dialect	The dialect of the underlying database	POSTGRES	TRUE

<table>
<tr><th>Property</th><th>Description</th><th>Example value</th><th>Required</th></tr>
<tr><td>marshaller.type</td><td>The marshaller to use for the store. Possible values are:

- `LEGACY` Infinispan 8.2.x marshaller. Valid for source stores only.

- `CURRENT` Infinispan 9.x marshaller.

- `CUSTOM` Custom marshaller.</td><td>CURRENT</td><td>TRUE</td></tr>
<tr><td>marshaller.class</td><td>The class of the marshaller if type=CUSTOM</td><td>org.example.CustomMarshaller</td><td></td></tr>
<tr><td>marshaller.externalizers</td><td>A comma-separated list of custom AdvancedExternalizer implementations to load `[id]:<Externalizer class>`</td><td>`25:Externalizer1,org.example.Externalizer2`</td><td></td></tr>
<tr><td>connection_pool.connection_url</td><td>The JDBC connection url</td><td>`jdbc:postgresql:postgres`</td><td>TRUE</td></tr>
<tr><td>connection_pool.driver_class</td><td>The class of the JDBC driver</td><td>org.postrgesql.Driver</td><td>TRUE</td></tr>
<tr><td>connection_pool.username</td><td>Database username</td><td></td><td>TRUE</td></tr>
<tr><td>connection_pool.password</td><td>Database password</td><td></td><td>TRUE</td></tr>
<tr><td>db.major_version</td><td>Database major version</td><td>9</td><td></td></tr>
<tr><td>db.minor_version</td><td>Database minor version</td><td>5</td><td></td></tr>
<tr><td>db.disable_upsert</td><td>Disable db upsert</td><td>false</td><td></td></tr>
<tr><td>db.disable_indexing</td><td>Prevent table index being created</td><td>false</td><td></td></tr>
<tr><td>table.`<binary|string>`.table_name_prefix</td><td>Additional prefix for table name</td><td>tablePrefix</td><td></td></tr>
<tr><td>table.`<binary|string>`.`<id|data|timestamp>`.name</td><td>Name of the column</td><td>id_column</td><td>TRUE</td></tr>
<tr><td>table.`<binary|string>`.`<id|data|timestamp>`.type</td><td>Type of the column</td><td>VARCHAR</td><td>TRUE</td></tr>
</table>

Property	Description	Example value	Required
key_to_string_mapper	TwoWayKey2StringMapper Class	`org.infinispan.persistence.keymappers.DefaultTwoWayKey2StringMapper`	

LevelDB/RocksDB Properties

Property	Description	Example value	Required
location	The location of the db directory	/some/example/dir	TRUE
compression	The compression type to be used	SNAPPY	

SingleFileStore Properties

Property	Description	Example value	Required
location	The directory containing the store's .dat file	/some/example/dir	TRUE

SoftIndexFileStore Properties

Property	Description	Example value	Required
location	The location of the db directory	/some/example/dir	TRUE
index_location	The location of the db's index	/some/example/dir-index	Target Only

2.16. SPI

The following class diagram presents the main SPI interfaces of the persistence API:

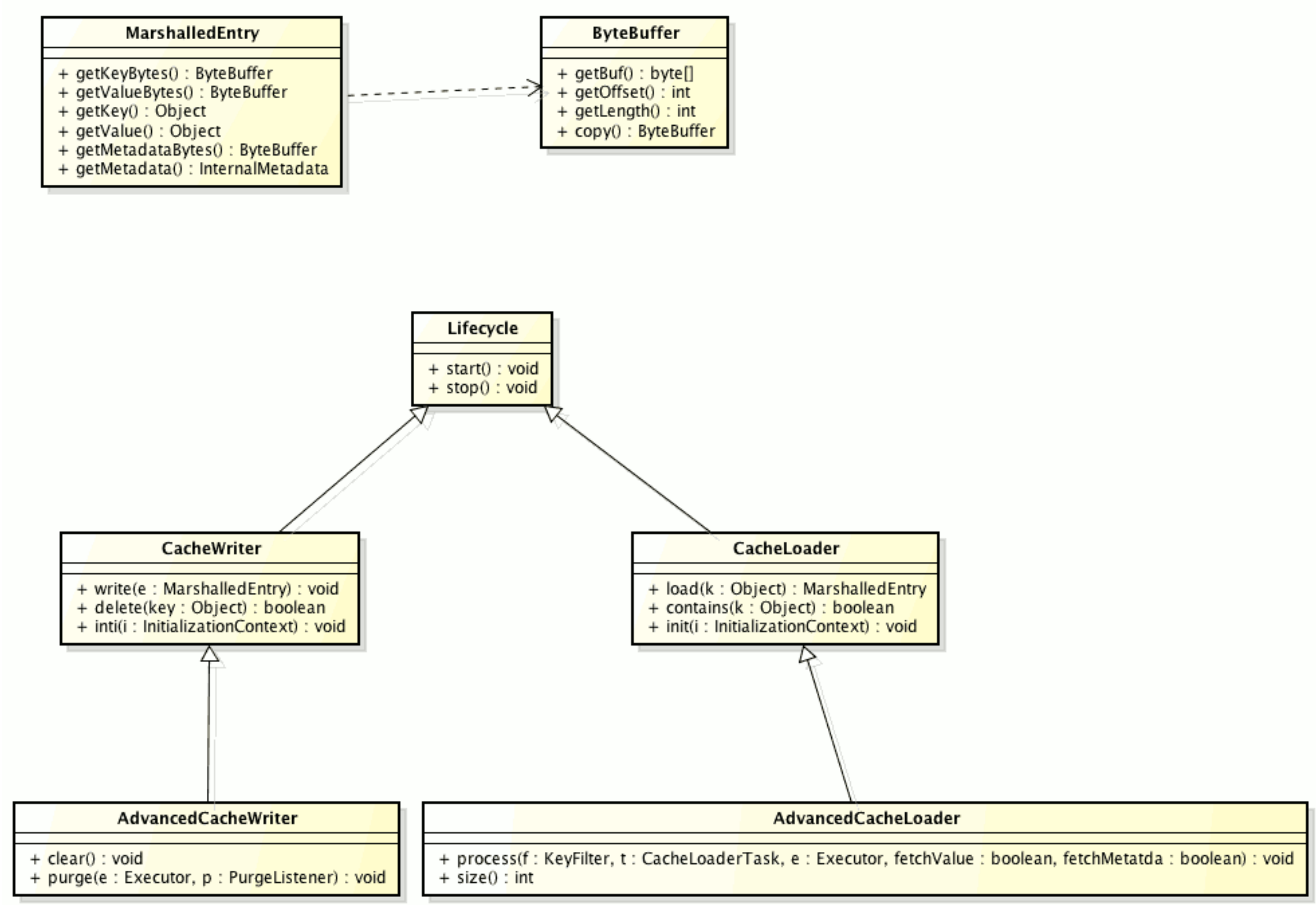

Figure 1. Persistence SPI

Some notes about the classes:

- ByteBuffer - abstracts the serialized form of an object
- MarshalledEntry - abstracts the information held within a persistent store corresponding to a key-value added to the cache. Provides method for reading this information both in serialized (ByteBuffer) and deserialized (Object) format. Normally data read from the store is kept in serialized format and lazily deserialized on demand, within the MarshalledEntry implementation
- CacheWriter and CacheLoader provide basic methods for reading and writing to a store
- AdvancedCacheLoader and AdvancedCacheWriter provide operations to manipulate the underlaying storage in bulk: parallel iteration and purging of expired entries, clear and size.
- SegmentedAdvancedLoadWriteStore provide all the various operations that deal with segments.

A cache store can be segmented if it does one of the following:

- Implements the SegmentedAdvancedLoadWriteStore interface. In this case only a single store instance is used per cache.
- Has a configuration that extends the AbstractSegmentedConfiguration abstract class. Doing this requires you to implement the `newConfigurationFrom` method where it is expected that a new `StoreConfiguration` instance is created per invocation. This creates a store instance per segment to which a node can write. Stores might start and stop as data is moved between nodes.

A provider might choose to only implement a subset of these interfaces:

- Not implementing the AdvancedCacheWriter makes the given writer not usable for purging expired entries or clear
- If a loader does not implement the AdvancedCacheLoader inteface, then it will not participate in preloading nor in cache iteration (required also for stream operations).

If you’re looking at migrating your existing store to the new API or to write a new store implementation, the SingleFileStore might be a good starting point/example.

2.16.1. More implementations

Many more cache loader and cache store implementations exist. Visit this website for more details.

Chapter 3. Setting Up Partition Handling

3.1. Partition handling

An Infinispan cluster is built out of a number of nodes where data is stored. In order not to lose data in the presence of node failures, Infinispan copies the same data — cache entry in Infinispan parlance — over multiple nodes. This level of data redundancy is configured through the `numOwners` configuration attribute and ensures that as long as fewer than `numOwners` nodes crash simultaneously, Infinispan has a copy of the data available.

However, there might be catastrophic situations in which more than `numOwners` nodes disappear from the cluster:

Split brain

Caused e.g. by a router crash, this splits the cluster in two or more partitions, or sub-clusters that operate independently. In these circumstances, multiple clients reading/writing from different partitions see different versions of the same cache entry, which for many application is problematic. Note there are ways to alleviate the possibility for the split brain to happen, such as redundant networks or IP bonding. These only reduce the window of time for the problem to occur, though.

`numOwners` **nodes crash in sequence**

When at least `numOwners` nodes crash in rapid succession and Infinispan does not have the time to properly rebalance its state between crashes, the result is partial data loss.

The partition handling functionality discussed in this section allows the user to configure what operations can be performed on a cache in the event of a split brain occurring. Infinispan provides multiple partition handling strategies, which in terms of Brewer's CAP theorem determine whether availability or consistency is sacrificed in the presence of partition(s). Below is a list of the provided strategies:

Strategy	Description	CAP
DENY_READ_WRITES	If the partition does not have all owners for a given segment, both reads and writes are denied for all keys in that segment.	Consistency
ALLOW_READS	Allows reads for a given key if it exists in this partition, but only allows writes if this partition contains all owners of a segment. This is still a consistent approach because some entries are readable if available in this partition, but from a client application perspective it is not deterministic.	Consistency

Strategy	Description	CAP
ALLOW_READ_WRITES	Allow entries on each partition to diverge, with conflict resolution attempted upon the partitions merging.	Availability

The requirements of your application should determine which strategy is appropriate. For example, DENY_READ_WRITES is more appropriate for applications that have high consistency requirements; i.e. when the data read from the system must be accurate. Whereas if Infinispan is used as a best-effort cache, partitions maybe perfectly tolerable and the ALLOW_READ_WRITES might be more appropriate as it favours availability over consistency.

The following sections describe how Infinispan handles split brain and successive failures for each of the partition handling strategies. This is followed by a section describing how Infinispan allows for automatic conflict resolution upon partition merges via merge policies. Finally, we provide a section describing how to configure partition handling strategies and merge policies.

3.1.1. Split brain

In a split brain situation, each network partition will install its own JGroups view, removing the nodes from the other partition(s). We don't have a direct way of determining whether the has been split into two or more partitions, since the partitions are unaware of each other. Instead, we assume the cluster has split when one or more nodes disappear from the JGroups cluster without sending an explicit leave message.

Split Strategies

In this section, we detail how each partition handling strategy behaves in the event of split brain occurring.

ALLOW_READ_WRITES

Each partition continues to function as an independent cluster, with all partitions remaining in AVAILABLE mode. This means that each partition may only see a part of the data, and each partition could write conflicting updates in the cache. During a partition merge these conflicts are automatically resolved by utilising the ConflictManager and the configured EntryMergePolicy.

DENY_READ_WRITES

When a split is detected each partition does not start a rebalance immediately, but first it checks whether it should enter **DEGRADED** mode instead:

- If at least one segment has lost all its owners (meaning at least *numOwners* nodes left since the last rebalance ended), the partition enters DEGRADED mode.
- If the partition does not contain a simple majority of the nodes (floor(numNodes/2) + 1) in the *latest stable topology*, the partition also enters DEGRADED mode.
- Otherwise the partition keeps functioning normally, and it starts a rebalance.

The *stable topology* is updated every time a rebalance operation ends and the coordinator determines that another rebalance is not necessary.

These rules ensures that at most one partition stays in AVAILABLE mode, and the other partitions enter DEGRADED mode.

When a partition is in DEGRADED mode, it only allows access to the keys that are wholly owned:

- Requests (reads and writes) for entries that have all the copies on nodes within this partition are honoured.
- Requests for entries that are partially or totally owned by nodes that disappeared are rejected with an `AvailabilityException`.

This guarantees that partitions cannot write different values for the same key (cache is consistent), and also that one partition can not read keys that have been updated in the other partitions (no stale data).

To exemplify, consider the initial cluster `M = {A, B, C, D}`, configured in distributed mode with `numOwners = 2`. Further on, consider three keys `k1`, `k2` and `k3` (that might exist in the cache or not) such that `owners(k1) = {A,B}`, `owners(k2) = {B,C}` and `owners(k3) = {C,D}`. Then the network splits in two partitions, `N1 = {A, B}` and `N2 = {C, D}`, they enter DEGRADED mode and behave like this:

- on `N1`, `k1` is available for read/write, `k2` (partially owned) and `k3` (not owned) are not available and accessing them results in an `AvailabilityException`
- on `N2`, `k1` and `k2` are not available for read/write, `k3` is available

A relevant aspect of the partition handling process is the fact that when a split brain happens, the resulting partitions rely on the original segment mapping (the one that existed before the split brain) in order to calculate key ownership. So it doesn't matter if `k1`, `k2`, or `k3` already existed cache or not, their availability is the same.

If at a further point in time the network heals and `N1` and `N2` partitions merge back together into the initial cluster `M`, then `M` exits the degraded mode and becomes fully available again. During this merge operation, because `M` has once again become fully available, the ConflictManager and the configured EntryMergePolicy are used to check for any conflicts that may have occurred in the interim period between the split brain occurring and being detected.

As another example, the cluster could split in two partitions `O1 = {A, B, C}` and `O2 = {D}`, partition `O1` will stay fully available (rebalancing cache entries on the remaining members). Partition `O2`, however, will detect a split and enter the degraded mode. Since it doesn't have any fully owned keys, it will reject any read or write operation with an `AvailabilityException`.

If afterwards partitions `O1` and `O2` merge back into `M`, then the ConflictManager attempts to resolve any conflicts and `D` once again becomes fully available.

ALLOW_READS

Partitions are handled in the same manner as DENY_READ_WRITES, except that when a partition is in DEGRADED mode read operations on a partially owned key WILL not throw an AvailabilityException.

Current limitations

Two partitions could start up isolated, and as long as they don't merge they can read and write inconsistent data. In the future, we will allow custom availability strategies (e.g. check that a certain node is part of the cluster, or check that an external machine is accessible) that could handle that situation as well.

3.1.2. Successive nodes stopped

As mentioned in the previous section, Infinispan can't detect whether a node left the JGroups view because of a process/machine crash, or because of a network failure: whenever a node leaves the JGroups cluster abruptly, it is assumed to be because of a network problem.

If the configured number of copies (`numOwners`) is greater than 1, the cluster can remain available and will try to make new replicas of the data on the crashed node. However, other nodes might crash during the rebalance process. If more than `numOwners` nodes crash in a short interval of time, there is a chance that some cache entries have disappeared from the cluster altogether. In this case, with the DENY_READ_WRITES or ALLOW_READS strategy enabled, Infinispan assumes (incorrectly) that there is a split brain and enters DEGRADED mode as described in the split-brain section.

The administrator can also shut down more than `numOwners` nodes in rapid succession, causing the loss of the data stored only on those nodes. When the administrator shuts down a node gracefully, Infinispan knows that the node can't come back. However, the cluster doesn't keep track of how each node left, and the cache still enters DEGRADED mode as if those nodes had crashed.

At this stage there is no way for the cluster to recover its state, except stopping it and repopulating it on restart with the data from an external source. Clusters are expected to be configured with an appropriate `numOwners` in order to avoid `numOwners` successive node failures, so this situation should be pretty rare. If the application can handle losing some of the data in the cache, the administrator can force the availability mode back to AVAILABLE via JMX.

3.1.3. Conflict Manager

The conflict manager is a tool that allows users to retrieve all stored replica values for a given key. In addition to allowing users to process a stream of cache entries whose stored replicas have conflicting values. Furthermore, by utilising implementations of the EntryMergePolicy interface it is possible for said conflicts to be resolved automatically.

Detecting Conflicts

Conflicts are detected by retrieving each of the stored values for a given key. The conflict manager retrieves the value stored from each of the key's write owners defined by the current consistent hash. The .equals method of the stored values is then used to determine whether all values are equal. If all values are equal then no conflicts exist for the key, otherwise a conflict has occurred. Note that null values are returned if no entry exists on a given node, therefore we deem a conflict to have occurred if both a null and non-null value exists for a given key.

Merge Policies

In the event of conflicts arising between one or more replicas of a given CacheEntry, it is necessary for a conflict resolution algorithm to be defined, therefore we provide the EntryMergePolicy interface. This interface consists of a single method, "merge", whose returned CacheEntry is utilised as the "resolved" entry for a given key. When a non-null CacheEntry is returned, this entries value is "put" to all replicas in the cache. However when the merge implementation returns a null value, all replicas associated with the conflicting key are removed from the cache.

The merge method takes two parameters: the "preferredEntry" and "otherEntries". In the context of a partition merge, the preferredEntry is the primary replica of a CacheEntry stored in the partition that contains the most nodes or if partitions are equal the one with the largest topologyId. In the event of overlapping partitions, i.e. a node A is present in the topology of both partitions {A}, {A,B,C}, we pick {A} as the preferred partition as it will have the higher topologId as the other partition’s topology is behind. When a partition merge is not occurring, the "preferredEntry" is simply the primary replica of the CacheEntry. The second parameter, "otherEntries" is simply a list of all other entries associated with the key for which a conflict was detected.

> EntryMergePolicy::merge is only called when a conflict has been detected, it is not called if all CacheEntrys are the same.

Currently Infinispan provides the following implementations of EntryMergePolicy:

Policy	Description
MergePolicy.NONE (default)	No attempt is made to resolve conflicts. Entries hosted on the minority partition are removed and the nodes in this partition do not hold any data until the rebalance starts. Note, this behaviour is equivalent to prior Infinispan versions which did not support conflict resolution. Note, in this case all changes made to entries hosted on the minority partition are lost, but once the rebalance has finished all entries will be consistent.

Policy	Description
MergePolicy.PREFERRED_ALWAYS	Always utilise the "preferredEntry". MergePolicy.NONE is almost equivalent to PREFERRED_ALWAYS, albeit without the performance impact of performing conflict resolution, therefore MergePolicy.NONE should be chosen unless the following scenario is a concern. When utilising the DENY_READ_WRITES or DENY_READ strategy, it is possible for a write operation to only partially complete when the partitions enter DEGRADED mode, resulting in replicas containing inconsistent values. MergePolicy.PREFERRED_ALWAYS will detect said inconsistency and resolve it, whereas with MergePolicy.NONE the CacheEntry replicas will remain inconsistent after the cluster has rebalanced.
MergePolicy.PREFERRED_NON_NULL	Utilise the "preferredEntry" if it is non-null, otherwise utilise the first entry from "otherEntries".
MergePolicy.REMOVE_ALL	Always remove a key from the cache when a conflict is detected.
Fully qualified class name	The custom implementation for merge will be used Custom merge policy

3.1.4. Usage

During a partition merge the ConflictManager automatically attempts to resolve conflicts utilising the configured EntryMergePolicy, however it is also possible to manually search for/resolve conflicts as required by your application.

The code below shows how to retrieve an EmbeddedCacheManager's ConflictManager, how to retrieve all versions of a given key and how to check for conflicts across a given cache.

```java
EmbeddedCacheManager manager = new DefaultCacheManager("example-config.xml");
Cache<Integer, String> cache = manager.getCache("testCache");
ConflictManager<Integer, String> crm = ConflictManagerFactory.get(cache
.getAdvancedCache());

// Get All Versions of Key
Map<Address, InternalCacheValue<String>> versions = crm.getAllVersions(1);

// Process conflicts stream and perform some operation on the cache
Stream<Map<Address, InternalCacheEntry<Integer, String>>> stream = crm.getConflicts();
stream.forEach(map -> {
   CacheEntry<Object, Object> entry = map.values().iterator().next();
   Object conflictKey = entry.getKey();
   cache.remove(conflictKey);
});

// Detect and then resolve conflicts using the configured EntryMergePolicy
crm.resolveConflicts();

// Detect and then resolve conflicts using the passed EntryMergePolicy instance
crm.resolveConflicts((preferredEntry, otherEntries) -> preferredEntry);
```

Although the `ConflictManager::getConflicts` stream is processed per entry, the underlying spliterator is in fact lazily-loading cache entries on a per segment basis.

3.1.5. Configuring partition handling

Unless the cache is distributed or replicated, partition handling configuration is ignored. The default partition handling strategy is ALLOW_READ_WRITES and the default EntryMergePolicy is MergePolicies::PREFERRED_ALWAYS.

```xml
<distributed-cache name="the-default-cache">
   <partition-handling when-split="ALLOW_READ_WRITES" merge-policy="
PREFERRED_NON_NULL"/>
</distributed-cache>
```

The same can be achieved programmatically:

```java
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
                    .whenSplit(PartitionHandling.ALLOW_READ_WRITES)
                    .mergePolicy(MergePolicies.PREFERRED_ALWAYS);
```

Implement a custom merge policy

It’s also possible to provide custom implementations of the EntryMergePolicy

```
<distributed-cache name="the-default-cache">
   <partition-handling when-split="ALLOW_READ_WRITES" merge-policy=
"org.example.CustomMergePolicy"/>
</distributed-cache>
```

```
ConfigurationBuilder dcc = new ConfigurationBuilder();
dcc.clustering().partitionHandling()
                    .whenSplit(PartitionHandling.ALLOW_READ_WRITES)
                    .mergePolicy(new CustomMergePolicy());
```

```
public class CustomMergePolicy implements EntryMergePolicy<String, String> {

   @Override
   public CacheEntry<String, String> merge(CacheEntry<String, String> preferredEntry,
List<CacheEntry<String, String>> otherEntries) {
      // decide which entry should be used

      return the_solved_CacheEntry;
   }
```

Deploy custom merge policies to a Infinispan server instance

To utilise a custom EntryMergePolicy implementation on the server, it's necessary for the implementation class(es) to be deployed to the server. This is accomplished by utilising the java service-provider convention and packaging the class files in a jar which has a META-INF/services/org.infinispan.conflict.EntryMergePolicy file containing the fully qualified class name of the EntryMergePolicy implementation.

```
# list all necessary implementations of EntryMergePolicy with the full qualified name
org.example.CustomMergePolicy
```

In order for a Custom merge policy to be utilised on the server, you should enable object storage, if your policies semantics require access to the stored Key/Value objects. This is because cache entries in the server may be stored in a marshalled format and the Key/Value objects returned to your policy would be instances of WrappedByteArray. However, if the custom policy only depends on the metadata associated with a cache entry, then object storage is not required and should be avoided (unless needed for other reasons) due to the additional performance cost of marshalling data per request. Finally, object storage is never required if one of the provided merge policies is used.

3.1.6. Monitoring and administration

The availability mode of a cache is exposed in JMX as an attribute in the Cache MBean. The attribute is writable, allowing an administrator to forcefully migrate a cache from DEGRADED mode back to AVAILABLE (at the cost of consistency).

The availability mode is also accessible via the AdvancedCache interface:

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

// Read the availability
boolean available = ac.getAvailability() == AvailabilityMode.AVAILABLE;

// Change the availability
if (!available) {
   ac.setAvailability(AvailabilityMode.AVAILABLE);
}
```

Chapter 4. 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.

4.1. 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`, `ModuleCommandInitializer` 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

4.1.1. 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
```

For a full, working example of a sample module that makes use of custom commands and visitors, check out Infinispan Sample Module .

4.1.2. 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

4.2. 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.

Chapter 5. Custom Interceptors

It is possible to add custom interceptors to Infinispan, both declaratively and programatically. 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. For a detailed list refer to CommandInterceptor API.

5.1. 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>
```

5.2. 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.

5.3. Custom interceptor design

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

- Custom interceptors must extend BaseCustomInterceptor
- 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 6. Rolling Upgrades and Updates with Kubernetes and OpenShift

Pods running in Kubernetes and OpenShift are immutable. The only way you can alter the configuration is to roll out a new deployment.

Upgrades and updates sound similar but are distinct processes for rolling out new deployments.

6.1. Performing Rolling Updates on Kubernetes

Rolling updates replace existing pods with new ones.

- Rolling Updates
- When to Use a Rolling Deployment

Example DeploymentConfiguration for Rolling Updates

```yaml
- apiVersion: v1
  kind: DeploymentConfig
  metadata:
    name: infinispan-cluster
  spec:
    replicas: 3
    strategy:
      type: Rolling
      rollingParams:
        updatePeriodSeconds: 10
        intervalSeconds: 20
        timeoutSeconds: 600
        maxUnavailable: 1
        maxSurge: 1
    template:
      spec:
        containers:
        - args:
          - -Djboss.default.jgroups.stack=kubernetes
          image: jboss/infinispan-server:latest
          name: infinispan-server
          ports:
          - containerPort: 8181
            protocol: TCP
          - containerPort: 9990
            protocol: TCP
          - containerPort: 11211
            protocol: TCP
          - containerPort: 11222
            protocol: TCP
          - containerPort: 57600
```

```
            protocol: TCP
          - containerPort: 7600
            protocol: TCP
          - containerPort: 8080
            protocol: TCP
          env:
          - name: KUBERNETES_NAMESPACE
            valueFrom: {fieldRef: {apiVersion: v1, fieldPath: metadata.namespace}}
          terminationMessagePath: /dev/termination-log
          terminationGracePeriodSeconds: 90
          livenessProbe:
            exec:
              command:
              - /usr/local/bin/is_running.sh
            initialDelaySeconds: 10
            timeoutSeconds: 80
            periodSeconds: 60
            successThreshold: 1
            failureThreshold: 5
          readinessProbe:
             exec:
                command:
                - /usr/local/bin/is_healthy.sh
             initialDelaySeconds: 10
             timeoutSeconds: 40
             periodSeconds: 30
             successThreshold: 2
             failureThreshold: 5
```

Kubernetes uses very similar concept to Deployment Configurations called `Deployment`.

It is also highly recommended to adjust the JGroups stack to discover new nodes (or leaves) more quickly. One should at least adjust the value of `FD_ALL` timeout and adjust it to the longest GC Pause.

Other hints for tuning configuration parameters are:

- OpenShift should replace running nodes one by one. This can be achieved by adjusting `rollingParams` (`maxUnavailable: 1` and `maxSurge: 1`).
- Depending on the cluster size, one needs to adjust `updatePeriodSeconds` and `intervalSeconds`. The bigger cluster size is, the bigger those values should be used.
- When using Initial State Transfer, the `initialDelaySeconds` value for both probes should be set to higher value.
- During Initial State Transfer nodes might not respond to probes. The best results are achieved with higher values of `failureThreshold` and `successThreshold` values.

6.2. Performing Rolling Upgrades on Kubernetes

Rolling upgrades migrate data from one Infinispan cluster to another.

For both Kubernetes and OpenShift, the rolling upgrade procedure is nearly identical to the procedure for Infinispan server rolling upgrades.

Differences in rolling upgrade procedures

- Depending on configuration, it is a good practice to use OpenShift Routes or Kubernetes Ingress API to expose services to the clients. During the upgrade the Route (or Ingress) used by the clients can be altered to point to the new cluster.
- Invoking CLI commands can be done by using Kubernetes (`kubectl exec`) or OpenShift clients (`oc exec`). Here is an example: `oc exec <POD_NAME> — '/opt/jboss/infinispan-server/bin/ispn-cli.sh' '-c' '--controller=$(hostname -i):9990' '/subsystem=datagrid-infinispan/cache-container=clustered/distributed-cache=default:disconnect-source(migrator-name=hotrod)'`

Key differences when upgrading using the library mode:

- Client application needs to expose JMX. It usually depends on application and environment type but the easiest way to do it is to add the following switches into the Java boostrap script `-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=<PORT>`.
- Connecting to the JMX can be done by forwarding ports. With OpenShift this might be achieved by using `oc port-forward` command whereas in Kubernetes by `kubectl port-forward`.

The last step in the Rolling Upgrade (removing a Remote Cache Store) needs to be performed differently. We need to use Kubernetes/OpenShift Rolling update command and replace Pods configuration with the one which does not contain Remote Cache Store.

A detailed instruction might be found in ISPN-6673 ticket.