Configuring Infinispan 11.0

Table of Contents

Chapter 1. Infinispan Caches

Infinispan caches provide flexible, in-memory data stores that you can configure to suit use cases such as:

- boosting application performance with high-speed local caches.
- optimizing databases by decreasing the volume of write operations.
- providing resiliency and durability for consistent data across clusters.

1.1. Cache Interface

`Cache<K,V>` is the central interface for Infinispan and extends `java.util.concurrent.ConcurrentMap`.

Cache entries are highly concurrent data structures in `key:value` format that support a wide and configurable range of data types, from simple strings to much more complex objects.

1.2. Cache Managers

Infinispan provides a `CacheManager` interface that lets you create, modify, and manage local or clustered caches. Cache Managers are the starting point for using Infinispan caches.

There are two `CacheManager` implementations:

`EmbeddedCacheManager`
: Entry point for caches when running Infinispan inside the same Java Virtual Machine (JVM) as the client application, which is also known as Library Mode.

`RemoteCacheManager`
: Entry point for caches when running Infinispan as a remote server in its own JVM. When it starts running, `RemoteCacheManager` establishes a persistent TCP connection to a Hot Rod endpoint on a Infinispan server.

> Both embedded and remote `CacheManager` implementations share some methods and properties. However, semantic differences do exist between `EmbeddedCacheManager` and `RemoteCacheManager`.

1.3. Cache Containers

Cache containers declare one or more local or clustered caches that a Cache Manager controls.

Cache container declaration

```
<cache-container name="clustered" default-cache="default">
  ...
</cache-container>
```

1.4. Cache Modes

> Infinispan Cache Managers can create and control multiple caches that use different modes. For example, you can use the same Cache Manager for local caches, distributes caches, and caches with invalidation mode.

Local Caches

Infinispan runs as a single node and never replicates read or write operations on cache entries.

Clustered Caches

Infinispan instances running on the same network can automatically discover each other and form clusters to handle cache operations.

Invalidation Mode

Rather than replicating cache entries across the cluster, Infinispan evicts stale data from all nodes whenever operations modify entries in the cache. Infinispan performs local read operations only.

Replicated Caches

Infinispan replicates each cache entry on all nodes and performs local read operations only.

Distributed Caches

Infinispan stores cache entries across a subset of nodes and assigns entries to fixed owner nodes. Infinispan requests read operations from owner nodes to ensure it returns the correct value.

Scattered Caches

Infinispan stores cache entries across a subset of nodes. By default Infinispan assigns a primary owner and a backup owner to each cache entry in scattered caches. Infinispan assigns primary owners in the same way as with distributed caches, while backup owners are always the nodes that initiate the write operations. Infinispan requests read operations from at least one owner node to ensure it returns the correct value.

1.4.1. Cache Mode Comparison

The cache mode that you should choose depends on the qualities and guarantees you need for your data.

The following table summarizes the primary differences between cache modes:

	Simple	Local	Invalidation	Replicated	Distributed	Scattered
Clustered	**No**	**No**	**Yes**	**Yes**	**Yes**	**Yes**
Read performance	**Highest** (local)	**High** (local)	**High** (local)	**High** (local)	**Medium** (owners)	**Medium** (primary)

	Simple	Local	Invalidation	Replicated	Distributed	Scattered
Write performance	**Highest** (local)	**High** (local)	**Low** (all nodes, no data)	**Lowest** (all nodes)	**Medium** (owner nodes)	**Higher** (single RPC)
Capacity	**Single node**	**Single node**	**Single node**	**Smallest node**	**Cluster** (sum_(i=1)^" nodes""node_capacity")/"owners"	**Cluster** (sum_(i=1)^" nodes""node_capacity")/"2"
Availability	**Single node**	**Single node**	**Single node**	**All nodes**	**Owner nodes**	**Owner nodes**
Features	**No TX, persistence, indexing**	**All**	**All**	**All**	**All**	**No TX**

Chapter 2. Local Caches

While Infinispan is particularly interesting in clustered mode, it also offers a very capable local mode. In this mode, it acts as a simple, in-memory data cache similar to a `ConcurrentHashMap`.

But why would one use a local cache rather than a map? Caches offer a lot of features over and above a simple map, including write-through and write-behind to a persistent store, eviction of entries to prevent running out of memory, and expiration.

Infinispan's `Cache` interface extends JDK's `ConcurrentMap` — making migration from a map to Infinispan trivial.

Infinispan caches also support transactions, either integrating with an existing transaction manager or running a separate one. Local caches transactions have two choices:

1. When to lock? **Pessimistic locking** locks keys on a write operation or when the user calls `AdvancedCache.lock(keys)` explicitly. **Optimistic locking** only locks keys during the transaction commit, and instead it throws a `WriteSkewCheckException` at commit time, if another transaction modified the same keys after the current transaction read them.
2. Isolation level. We support **read-committed** and **repeatable read**.

2.1. Simple Caches

Traditional local caches use the same architecture as clustered caches, i.e. they use the interceptor stack. That way a lot of the implementation can be reused. However, if the advanced features are not needed and performance is more important, the interceptor stack can be stripped away and simple cache can be used.

So, which features are stripped away? From the configuration perspective, simple cache does not support:

- transactions and invocation batching
- persistence (cache stores and loaders)
- custom interceptors (there's no interceptor stack!)
- indexing
- transcoding
- store as binary (which is hardly useful for local caches)

From the API perspective these features throw an exception:

- adding custom interceptors
- Distributed Executors Framework

So, what's left?

- basic map-like API

- cache listeners (local ones)
- expiration
- eviction
- security
- JMX access
- statistics (though for max performance it is recommended to switch this off using statistics-available=false)

  Declarative configuration

  ```
  <local-cache name="mySimpleCache" simple-cache="true">
      <!-- expiration, eviction, security... -->
  </local-cache>
  ```

  Programmatic configuration

  ```
  DefaultCacheManager cm = getCacheManager();
  ConfigurationBuilder builder = new ConfigurationBuilder().simpleCache(true);
  cm.defineConfiguration("mySimpleCache", builder.build());
  Cache cache = cm.getCache("mySimpleCache");
  ```

Simple cache checks against features it does not support, if you configure it to use e.g. transactions, configuration validation will throw an exception.

Chapter 3. Clustered Caches

Clustered caches store data across multiple Infinispan nodes using JGroups technology as the transport layer to pass data across the network.

3.1. Invalidation Mode

You can use Infinispan in invalidation mode to optimize systems that perform high volumes of read operations. A good example is to use invalidation to prevent lots of database writes when state changes occur.

This cache mode only makes sense if you have another, permanent store for your data such as a database and are only using Infinispan as an optimization in a read-heavy system, to prevent hitting the database for every read. If a cache is configured for invalidation, every time data is changed in a cache, other caches in the cluster receive a message informing them that their data is now stale and should be removed from memory and from any local store.

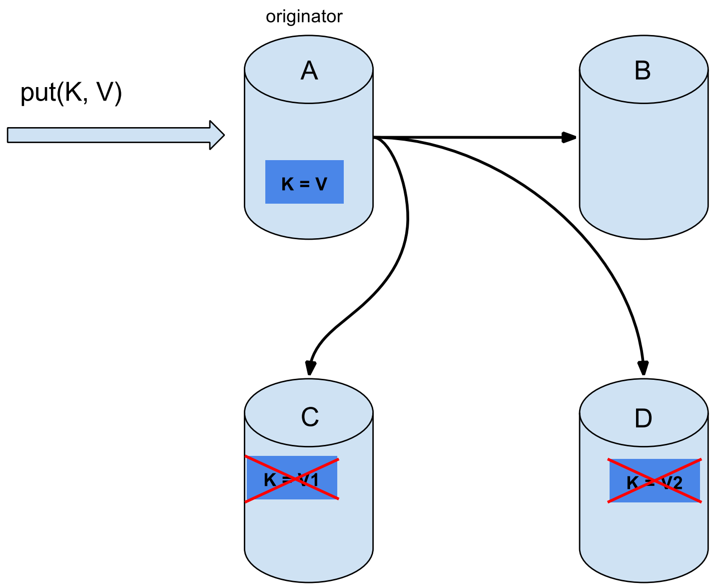

Figure 1. Invalidation mode

Sometimes the application reads a value from the external store and wants to write it to the local cache, without removing it from the other nodes. To do this, it must call `Cache.putForExternalRead(key, value)` instead of `Cache.put(key, value)`.

Invalidation mode can be used with a shared cache store. A write operation will both update the shared store, and it would remove the stale values from the other nodes' memory. The benefit of this is twofold: network traffic is minimized as invalidation messages are very small compared to replicating the entire value, and also other caches in the cluster look up modified data in a lazy manner, only when needed.

> Never use invalidation mode with a **local** store. The invalidation message will not remove entries in the local store, and some nodes will keep seeing the stale value.

An invalidation cache can also be configured with a special cache loader, `ClusterLoader`. When `ClusterLoader` is enabled, read operations that do not find the key on the local node will request it from all the other nodes first, and store it in memory locally. In certain situation it will store stale values, so only use it if you have a high tolerance for stale values.

Invalidation mode can be synchronous or asynchronous. When synchronous, a write blocks until all nodes in the cluster have evicted the stale value. When asynchronous, the originator broadcasts invalidation messages but doesn't wait for responses. That means other nodes still see the stale value for a while after the write completed on the originator.

Transactions can be used to batch the invalidation messages. Transactions acquire the key lock on the primary owner. To find more about how primary owners are assigned, please read the Key Ownership section.

- With pessimistic locking, each write triggers a lock message, which is broadcast to all the nodes. During transaction commit, the originator broadcasts a one-phase prepare message (optionally fire-and-forget) which invalidates all affected keys and releases the locks.
- With optimistic locking, the originator broadcasts a prepare message, a commit message, and an unlock message (optional). Either the one-phase prepare or the unlock message is fire-and-forget, and the last message always releases the locks.

3.2. Replicated Caches

Entries written to a replicated cache on any node will be replicated to all other nodes in the cluster, and can be retrieved locally from any node. Replicated mode provides a quick and easy way to share state across a cluster, however replication practically only performs well in small clusters (under 10 nodes), due to the number of messages needed for a write scaling linearly with the cluster size. Infinispan can be configured to use UDP multicast, which mitigates this problem to some degree.

Each key has a primary owner, which serializes data container updates in order to provide consistency. To find more about how primary owners are assigned, please read the Key Ownership section.

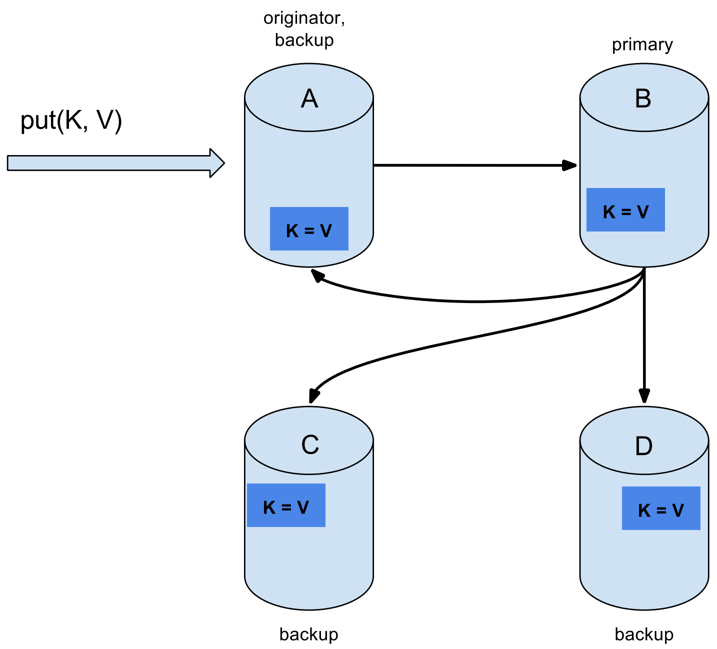

Figure 2. Replicated mode

Replicated mode can be synchronous or asynchronous.

- Synchronous replication blocks the caller (e.g. on a `cache.put(key, value)`) until the modifications have been replicated successfully to all the nodes in the cluster.
- Asynchronous replication performs replication in the background, and write operations return immediately. Asynchronous replication is not recommended, because communication errors, or errors that happen on remote nodes are not reported to the caller.

If transactions are enabled, write operations are not replicated through the primary owner.

- With pessimistic locking, each write triggers a lock message, which is broadcast to all the nodes. During transaction commit, the originator broadcasts a one-phase prepare message and an unlock message (optional). Either the one-phase prepare or the unlock message is fire-and-forget.
- With optimistic locking, the originator broadcasts a prepare message, a commit message, and an unlock message (optional). Again, either the one-phase prepare or the unlock message is fire-and-forget.

3.3. Distributed Caches

Distribution tries to keep a fixed number of copies of any entry in the cache, configured as `numOwners`. This allows the cache to scale linearly, storing more data as nodes are added to the cluster.

As nodes join and leave the cluster, there will be times when a key has more or less than numOwners copies. In particular, if numOwners nodes leave in quick succession, some entries will be lost, so we say that a distributed cache tolerates numOwners - 1 node failures.

The number of copies represents a trade-off between performance and durability of data. The more copies you maintain, the lower performance will be, but also the lower the risk of losing data due to server or network failures. Regardless of how many copies are maintained, distribution still scales linearly, and this is key to Infinispan's scalability.

The owners of a key are split into one **primary owner**, which coordinates writes to the key, and zero or more **backup owners**. To find more about how primary and backup owners are assigned, please read the Key Ownership section.

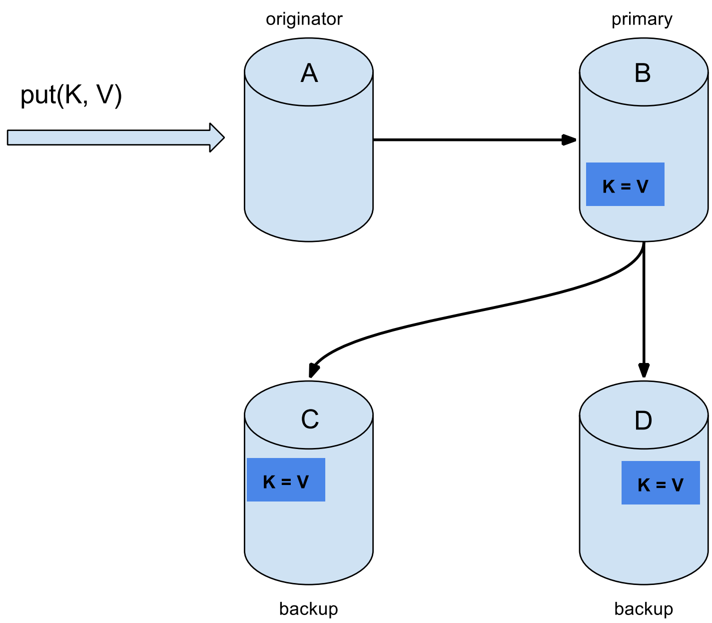

Figure 3. Distributed mode

A read operation will request the value from the primary owner, but if it doesn't respond in a reasonable amount of time, we request the value from the backup owners as well. (The infinispan.stagger.delay system property, in milliseconds, controls the delay between requests.) A read operation may require 0 messages if the key is present in the local cache, or up to 2 * numOwners messages if all the owners are slow.

A write operation will also result in at most 2 * numOwners messages: one message from the originator to the primary owner, numOwners - 1 messages from the primary to the backups, and the corresponding ACK messages.

> Cache topology changes may cause retries and additional messages, both for reads and for writes.

Just as replicated mode, distributed mode can also be synchronous or asynchronous. And as in replicated mode, asynchronous replication is not recommended because it can lose updates. In addition to losing updates, asynchronous distributed caches can also see a stale value when a thread writes to a key and then immediately reads the same key.

Transactional distributed caches use the same kinds of messages as transactional replicated caches, except lock/prepare/commit/unlock messages are sent only to the **affected nodes** (all the nodes that own at least one key affected by the transaction) instead of being broadcast to all the nodes in the cluster. As an optimization, if the transaction writes to a single key and the originator is the primary owner of the key, lock messages are not replicated.

3.3.1. Read consistency

Even with synchronous replication, distributed caches are not linearizable. (For transactional caches, we say they do not support serialization/snapshot isolation.) We can have one thread doing a single put:

```
cache.get(k) -> v1
cache.put(k, v2)
cache.get(k) -> v2
```

But another thread might see the values in a different order:

```
cache.get(k) -> v2
cache.get(k) -> v1
```

The reason is that read can return the value from **any** owner, depending on how fast the primary owner replies. The write is not atomic across all the owners — in fact, the primary commits the update only after it receives a confirmation from the backup. While the primary is waiting for the confirmation message from the backup, reads from the backup will see the new value, but reads from the primary will see the old one.

3.3.2. Key Ownership

Distributed caches split entries into a fixed number of segments and assign each segment to a list of owner nodes. Replicated caches do the same, with the exception that every node is an owner.

The first node in the list of owners is the **primary owner**. The other nodes in the list are **backup owners**. When the cache topology changes, because a node joins or leaves the cluster, the segment ownership table is broadcast to every node. This allows nodes to locate keys without making multicast requests or maintaining metadata for each key.

The `numSegments` property configures the number of segments available. However, the number of segments cannot change unless the cluster is restarted.

Likewise the key-to-segment mapping cannot change. Keys must always map to the same segments regardless of cluster topology changes. It is important that the key-to-segment mapping evenly distributes the number of segments allocated to each node while minimizing the number of segments that must move when the cluster topology changes.

You can customize the key-to-segment mapping by configuring a KeyPartitioner or by using the Grouping API.

However, Infinispan provides the following implementations:

SyncConsistentHashFactory

Uses an algorithm based on consistent hashing. Selected by default when server hinting is disabled.

This implementation always assigns keys to the same nodes in every cache as long as the cluster is symmetric. In other words, all caches run on all nodes. This implementation does have some negative points in that the load distribution is slightly uneven. It also moves more segments than strictly necessary on a join or leave.

TopologyAwareSyncConsistentHashFactory

Similar to `SyncConsistentHashFactory`, but adapted for Server Hinting. Selected by default when server hinting is enabled.

DefaultConsistentHashFactory

Achieves a more even distribution than `SyncConsistentHashFactory`, but with one disadvantage. The order in which nodes join the cluster determines which nodes own which segments. As a result, keys might be assigned to different nodes in different caches.

Was the default from version 5.2 to version 8.1 with server hinting disabled.

TopologyAwareConsistentHashFactory

Similar to *DefaultConsistentHashFactory*, but adapted for Server Hinting.

Was the default from version 5.2 to version 8.1 with server hinting enabled.

ReplicatedConsistentHashFactory

Used internally to implement replicated caches. You should never explicitly select this algorithm in a distributed cache.

Capacity Factors

Capacity factors allocate segment-to-node mappings based on resources available to nodes.

To configure capacity factors, you specify any non-negative number and the Infinispan hashing algorithm assigns each node a load weighted by its capacity factor (both as a primary owner and as a backup owner).

For example, nodeA has 2x the memory available than nodeB in the same Infinispan cluster. In this case, setting `capacityFactor` to a value of `2` configures Infinispan to allocate 2x the number of segments to nodeA.

Setting a capacity factor of `0` is possible but is recommended only in cases where nodes are not joined to the cluster long enough to be useful data owners.

3.3.3. Zero Capacity Node

You might need to configure a whole node where the capacity factor is `0` for every cache, user defined caches and internal caches. When defining a zero capacity node, the node won't hold any data. This is how you declare a zero capacity node:

```
<cache-container zero-capacity-node="true" />
```

```
new GlobalConfigurationBuilder().zeroCapacityNode(true);
```

However, note that this will be true for distributed caches only. If you are using replicated caches, the node will still keep a copy of the value. Use only distributed caches to make the best use of this feature.

3.3.4. Hashing Configuration

This is how you configure hashing declaratively, via XML:

```
<distributed-cache name="distributedCache" owners="2" segments="100" capacity-factor=
"2" />
```

And this is how you can configure it programmatically, in Java:

```
Configuration c = new ConfigurationBuilder()
   .clustering()
      .cacheMode(CacheMode.DIST_SYNC)
      .hash()
         .numOwners(2)
         .numSegments(100)
         .capacityFactor(2)
   .build();
```

3.3.5. Initial cluster size

Infinispan's very dynamic nature in handling topology changes (i.e. nodes being added / removed at runtime) means that, normally, a node doesn't wait for the presence of other nodes before starting. While this is very flexible, it might not be suitable for applications which require a specific number of nodes to join the cluster before caches are started. For this reason, you can specify how many nodes should have joined the cluster before proceeding with cache initialization. To do this, use the `initialClusterSize` and `initialClusterTimeout` transport properties. The declarative XML configuration:

```
<transport initial-cluster-size="4" initial-cluster-timeout="30000" />
```

The programmatic Java configuration:

```
GlobalConfiguration global = new GlobalConfigurationBuilder()
   .transport()
   .initialClusterSize(4)
   .initialClusterTimeout(30000, TimeUnit.MILLISECONDS)
   .build();
```

The above configuration will wait for *4* nodes to join the cluster before initialization. If the initial nodes do not appear within the specified timeout, the cache manager will fail to start.

3.3.6. L1 Caching

When L1 is enabled, a node will keep the result of remote reads locally for a short period of time (configurable, 10 minutes by default), and repeated lookups will return the local L1 value instead of asking the owners again.

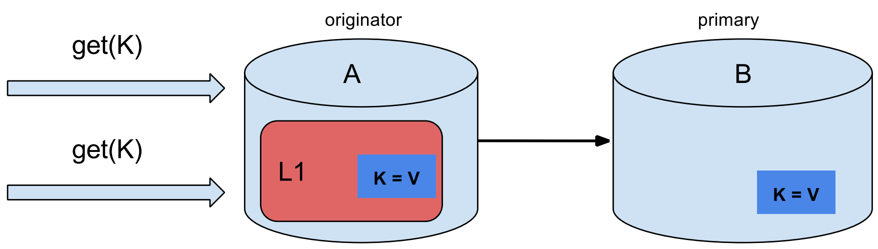

Figure 4. L1 caching

L1 caching is not free though. Enabling it comes at a cost, and this cost is that every entry update must broadcast an invalidation message to all the nodes. L1 entries can be evicted just like any other entry when the the cache is configured with a maximum size. Enabling L1 will improve performance for repeated reads of non-local keys, but it will slow down writes and it will increase memory consumption to some degree.

Is L1 caching right for you? The correct approach is to benchmark your application with and without L1 enabled and see what works best for your access pattern.

3.3.7. Server Hinting

The following topology hints can be specified:

Machine

This is probably the most useful, when multiple JVM instances run on the same node, or even when multiple virtual machines run on the same physical machine.

Rack

In larger clusters, nodes located on the same rack are more likely to experience a hardware or network failure at the same time.

Site

> Some clusters may have nodes in multiple physical locations for extra resilience. Note that Cross site replication is another alternative for clusters that need to span two or more data centres.

All of the above are optional. When provided, the distribution algorithm will try to spread the ownership of each segment across as many sites, racks, and machines (in this order) as possible.

Configuration

The hints are configured at transport level:

```
<transport
    cluster="MyCluster"
    machine="LinuxServer01"
    rack="Rack01"
    site="US-WestCoast" />
```

3.3.8. Key affinity service

In a distributed cache, a key is allocated to a list of nodes with an opaque algorithm. There is no easy way to reverse the computation and generate a key that maps to a particular node. However, we can generate a sequence of (pseudo-)random keys, see what their primary owner is, and hand them out to the application when it needs a key mapping to a particular node.

API

Following code snippet depicts how a reference to this service can be obtained and used.

```
// 1. Obtain a reference to a cache
Cache cache = ...
Address address = cache.getCacheManager().getAddress();

// 2. Create the affinity service
KeyAffinityService keyAffinityService = KeyAffinityServiceFactory
.newLocalKeyAffinityService(
      cache,
      new RndKeyGenerator(),
      Executors.newSingleThreadExecutor(),
      100);

// 3. Obtain a key for which the local node is the primary owner
Object localKey = keyAffinityService.getKeyForAddress(address);

// 4. Insert the key in the cache
cache.put(localKey, "yourValue");
```

The service is started at step 2: after this point it uses the supplied *Executor* to generate and queue keys. At step 3, we obtain a key from the service, and at step 4 we use it.

Lifecycle

`KeyAffinityService` extends `Lifecycle`, which allows stopping and (re)starting it:

```
public interface Lifecycle {
   void start();
   void stop();
}
```

The service is instantiated through `KeyAffinityServiceFactory`. All the factory methods have an `Executor` parameter, that is used for asynchronous key generation (so that it won't happen in the caller's thread). It is the user's responsibility to handle the shutdown of this `Executor`.

The `KeyAffinityService`, once started, needs to be explicitly stopped. This stops the background key generation and releases other held resources.

The only situation in which `KeyAffinityService` stops by itself is when the cache manager with which it was registered is shutdown.

Topology changes

When the cache topology changes (i.e. nodes join or leave the cluster), the ownership of the keys generated by the `KeyAffinityService` might change. The key affinity service keep tracks of these topology changes and doesn't return keys that would currently map to a different node, but it won't do anything about keys generated earlier.

As such, applications should treat `KeyAffinityService` purely as an optimization, and they should not rely on the location of a generated key for correctness.

In particular, applications should not rely on keys generated by `KeyAffinityService` for the same address to always be located together. Collocation of keys is only provided by the Grouping API.

The Grouping API

Complementary to Key affinity service, the grouping API allows you to co-locate a group of entries on the same nodes, but without being able to select the actual nodes.

How does it work?

By default, the segment of a key is computed using the key's `hashCode()`. If you use the grouping API, Infinispan will compute the segment of the group and use that as the segment of the key. See the Key Ownership section for more details on how segments are then mapped to nodes.

When the group API is in use, it is important that every node can still compute the owners of every key without contacting other nodes. For this reason, the group cannot be specified manually. The group can either be intrinsic to the entry (generated by the key class) or extrinsic (generated by an external function).

How do I use the grouping API?

First, you must enable groups. If you are configuring Infinispan programmatically, then call:

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

Or, if you are using XML:

```
<distributed-cache>
   <groups enabled="true"/>
</distributed-cache>
```

If you have control of the key class (you can alter the class definition, it's not part of an unmodifiable library), then we recommend using an intrinsic group. The intrinsic group is specified by adding the `@Group` annotation to a method. Let's take a look at an example:

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

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

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

> The group method must return a `String`

If you don't have control over the key class, or the determination of the group is an orthogonal concern to the key class, we recommend using an extrinsic group. An extrinsic group is specified by implementing the `Grouper` interface.

```java
public interface Grouper<T> {
    String computeGroup(T key, String group);

    Class<T> getKeyType();
}
```

If multiple `Grouper` classes are configured for the same key type, all of them will be called, receiving the value computed by the previous one. If the key class also has a `@Group` annotation, the first `Grouper` will receive the group computed by the annotated method. This allows you even greater control over the group when using an intrinsic group. Let's take a look at an example `Grouper` implementation:

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

   // The pattern requires a String key, of length 2, where the first character is
   // "k" and the second character is a digit. We take that digit, and perform
   // modular arithmetic on it to assign it to group "0" or group "1".
   private static Pattern kPattern = Pattern.compile("(^k)(<a>\\d</a>)$");

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

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

`Grouper` implementations must be registered explicitly in the cache configuration. If you are configuring Infinispan programmatically:

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

Or, if you are using XML:

```xml
<distributed-cache>
   <groups enabled="true">
      <grouper class="com.acme.KXGrouper" />
   </groups>
</distributed-cache>
```

Advanced Interface

`AdvancedCache` has two group-specific methods:

getGroup(groupName)
: Retrieves all keys in the cache that belong to a group.

removeGroup(groupName)
: Removes all the keys in the cache that belong to a group.

Both methods iterate over the entire data container and store (if present), so they can be slow when a cache contains lots of small groups.

3.4. Scattered Caches

Scattered mode is very similar to Distribution Mode as it allows linear scaling of the cluster. It allows single node failure by maintaining two copies of the data (as Distribution Mode with numOwners=2). Unlike Distributed, the location of data is not fixed; while we use the same Consistent Hash algorithm to locate the primary owner, the backup copy is stored on the node that wrote the data last time. When the write originates on the primary owner, backup copy is stored on any other node (the exact location of this copy is not important).

This has the advantage of single Remote Procedure Call (RPC) for any write (Distribution Mode requires one or two RPCs), but reads have to always target the primary owner. That results in faster writes but possibly slower reads, and therefore this mode is more suitable for write-intensive applications.

Storing multiple backup copies also results in slightly higher memory consumption. In order to remove out-of-date backup copies, invalidation messages are broadcast in the cluster, which generates some overhead. This makes scattered mode less performant in very big clusters (this behaviour might be optimized in the future).

When a node crashes, the primary copy may be lost. Therefore, the cluster has to reconcile the backups and find out the last written backup copy. This process results in more network traffic during state transfer.

Since the writer of data is also a backup, even if we specify machine/rack/site ids on the transport level the cluster cannot be resilient to more than one failure on the same machine/rack/site.

Currently it is not possible to use scattered mode in transactional cache. Asynchronous replication is not supported either; use asynchronous Cache API instead. Functional commands are not implemented neither but these are expected to be added soon.

The cache is configured in a similar way as the other cache modes, here is an example of declarative configuration:

```
<scattered-cache name="scatteredCache" />
```

And this is how you can configure it programmatically:

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

Scattered mode is not exposed in the server configuration as the server is usually accessed through the Hot Rod protocol. The protocol automatically selects primary owner for the writes and therefore the write (in distributed mode with two owner) requires single RPC inside the cluster, too. Therefore, scattered cache would not bring the performance benefit.

3.5. Asynchronous Communication with Clustered Caches

3.5.1. Asynchronous Communications

All clustered cache modes can be configured to use asynchronous communications with the `mode="ASYNC"` attribute on the `<replicated-cache/>`, `<distributed-cache>`, or `<invalidation-cache/>` element.

With asynchronous communications, the originator node does not receive any acknowledgement from the other nodes about the status of the operation, so there is no way to check if it succeeded on other nodes.

We do not recommend asynchronous communications in general, as they can cause inconsistencies in the data, and the results are hard to reason about. Nevertheless, sometimes speed is more important than consistency, and the option is available for those cases.

3.5.2. Asynchronous API

The Asynchronous API allows you to use synchronous communications, but without blocking the user thread.

There is one caveat: The asynchronous operations do NOT preserve the program order. If a thread calls `cache.putAsync(k, v1); cache.putAsync(k, v2)`, the final value of `k` may be either `v1` or `v2`. The advantage over using asynchronous communications is that the final value can't be `v1` on one node and `v2` on another.

3.5.3. Return Values in Asynchronous Communication

Because the `Cache` interface extends `java.util.Map`, write methods like `put(key, value)` and `remove(key)` return the previous value by default.

In some cases, the return value may not be correct:

1. When using `AdvancedCache.withFlags()` with `Flag.IGNORE_RETURN_VALUE`, `Flag.SKIP_REMOTE_LOOKUP`, or `Flag.SKIP_CACHE_LOAD`.
2. When the cache is configured with `unreliable-return-values="true"`.
3. When using asynchronous communications.
4. When there are multiple concurrent writes to the same key, and the cache topology changes. The topology change will make Infinispan retry the write operations, and a retried operation's return value is not reliable.

Transactional caches return the correct previous value in cases 3 and 4. However, transactional caches also have a gotcha: in distributed mode, the read-committed isolation level is implemented as repeatable-read. That means this example of "double-checked locking" won't work:

```
Cache cache = ...
TransactionManager tm = ...

tm.begin();
try {
   Integer v1 = cache.get(k);
   // Increment the value
   Integer v2 = cache.put(k, v1 + 1);
   if (Objects.equals(v1, v2) {
      // success
   } else {
      // retry
   }
} finally {
  tm.commit();
}
```

The correct way to implement this is to use `cache.getAdvancedCache().withFlags(Flag.FORCE_WRITE_LOCK).get(k)`.

In caches with optimistic locking, writes can also return stale previous values. Write skew checks can avoid stale previous values.

Chapter 4. Configuring Infinispan Caches

Infinispan lets you define properties and options for caches both declaratively and programmatically.

Declarative configuration uses XML files that adhere to a Infinispan schema. Programmatic configuration, on the other hand, uses Infinispan APIs.

In most cases, you use declarative configuration as a starting point for cache definitions. At runtime you can then programmatically configure your caches to tune settings or specify additional properties. However, Infinispan provides flexibility so you can choose either declarative, programmatic, or a combination of the two.

4.1. Declarative Configuration

You configure Infinispan caches by defining properties in `infinispan.xml`.

The following example shows the basic structure of a Infinispan configuration:

```
<infinispan> ①
   <cache-container default-cache="local"> ②
      <transport stack="udp" cluster="mycluster"/> ③
      <local-cache name="local"/> ④
      <invalidation-cache name="invalidation"/> ⑤
      <replicated-cache name="replicated"/> ⑥
      <distributed-cache name="distributed"/> ⑦
   </cache-container>
</infinispan>
```

① adds the root element for the Infinispan configuration. The minimum valid configuration is `<infinispan />`; however this provides very basic capabilities with no clustering and no cache instances.

② defines properties for all caches within the container and names the default cache.

③ defines transport properties for clustered cache modes. In the preceding example, `stack="udp"` specifies the default JGroups UDP transport stack and names the Infinispan cluster.

④ local cache.

⑤ invalidation cache.

⑥ replicated cache.

⑦ distributed cache.

Reference

- Infinispan 11.0 Configuration Schema
- infinispan-config-11.0.xsd

4.1.1. Cache Configuration Templates

Infinispan lets you define configuration templates that you can apply to multiple cache definitions or use as the basis for complex configurations.

For example, the following configuration contains a configuration template for local caches:

```
<infinispan>
   <cache-container default-cache="local"> ①
      <local-cache-configuration name="local-template"> ②
         <expiration interval="10000" lifespan="10" max-idle="10"/>
      </local-cache-configuration>
      <local-cache name="local" configuration="local-template" /> ③
   </cache-container>
</infinispan>
```

① specifies the "local" cache as the default.

② defines a configuration template named "local-template" that defines an expiration policy for local caches.

③ names a local cache instance that uses the configuration template.

Inheritance with configuration templates

Configuration templates can also inherit from other templates to extend and override settings.

> Configuration template inheritance is hierarchical. For a child configuration template to inherit from a parent, you must include it after the parent template.

The following is an example of configuration template inheritance:

```
<infinispan>
   <cache-container default-cache="local">
      <local-cache-configuration name="base-template"> ①
         <expiration interval="10000" lifespan="10" max-idle="10"/>
      </local-cache-configuration>

      <local-cache-configuration name="extended-template"
                                 configuration="base-template"> ②
         <expiration lifespan="20"/>
         <memory>
            <object size="2000"/>
         </memory>
      </local-cache-configuration>

      <local-cache name="local" configuration="base-template" /> ③
      <local-cache name="local-bounded" configuration="extended-template" /> ④
   </cache-container>
</infinispan>
```

① defines a configuration template named "base-template" that defines an expiration policy for local caches. In this example, "base-template" is the parent configuration template.

② defines a configuration template named "extended-template" that inherits settings from "base-template", modifies the `lifespan` attribute for expiration, and adds a `memory` element to the configuration. In this example, "extended-template" is a child of "base-template".

③ names a local cache that uses the configuration settings in "base-template".

④ names a local cache that uses the configuration settings in "extended-template".

> Configuration template inheritance is additive for elements that have multiple values, such as `property`. Resulting child configurations merge values from parent configurations.
>
> For example, `<property value_x="foo" />` in a parent configuration merges with `<property value_y="bar" />` in a child configuration to result in `<property value_x="foo" value_y="bar" />`.

4.1.2. Cache Configuration Wildcards

You can use wildcards to match cache definitions to configuration templates.

```
<infinispan>
    <cache-container>
        <local-cache-configuration name="basecache*"> ①
            <expiration interval="10500" lifespan="11" max-idle="11"/>
        </local-cache-configuration>
        <local-cache name="basecache-1"/> ②
        <local-cache name="basecache-2"/> ③
    </cache-container>
</infinispan>
```

① uses the `*` wildcard to match any cache names that start with "basecache".

② names a local cache "basecache-1" that uses the "basecache*" configuration template.

③ names a local cache "basecache-2" that uses the "basecache*" configuration template.

> Infinispan throws exceptions if cache names match more than one wildcard.

4.1.3. Multiple Configuration Files

Infinispan supports XML inclusions (XInclude) that allow you to split configuration across multiple files.

For example, the following configuration uses an XInclude:

```
<infinispan xmlns:xi="http://www.w3.org/2001/XInclude">
    <cache-container default-cache="cache-1">
        <xi:include href="local.xml" /> ①
    </cache-container>
</infinispan>
```

① includes an `local.xml` file that contains the following cache definition:

```
<local-cache name="mycache"/>
```

If you want to use a schema for your included fragments, use the `infinispan-config-fragment-11.0.xsd` schema:

include-with-schema.xml

```
<local-cache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="urn:infinispan:config:11.0
https://infinispan.org/schemas/infinispan-config-fragment-11.0.xsd"
             xmlns="urn:infinispan:config:11.0"
             name="mycache"/>
```

Infinispan configurations provides only minimal support for the XInclude specification. For example, you cannot use the `xpointer` attribute, the `xi:fallback` element, text processing, or content negotiation.

Reference

XInclude specification

4.2. Infinispan Configuration API

Configure Infinispan programmatically.

Global configuration

Use the `GlobalConfiguration` class to apply configuration to all caches under the Cache Manager.

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .cacheContainer().statistics(true) ①
  .metrics().gauges(true).histograms(true) ②
  .jmx().enable() ③
  .build();
```

① Enables Cache Manager statistics.

② Exports statistics through the `metrics` endpoint.

③ Exports statistics via JMX MBeans.

References:

- org.infinispan.configuration.global.GlobalConfiguration.
- Global Configuration API.

Cache configuration

Use the `ConfigurationBuilder` class to configure caches.

```
ConfigurationBuilder builder = new ConfigurationBuilder();
     builder.clustering() ①
            .cacheMode(CacheMode.DIST_SYNC) ②
            .l1().lifespan(25000L) ③
            .hash().numOwners(3) ④
            .statistics().enable(); ⑤
     Configuration cfg = builder.build();
```

① Enables cache clustering.

② Uses the distributed, synchronous cache mode.

③ Configures maximum lifespan for entries in the L1 cache.

④ Configures three cluster-wide replicas for each cache entry.

⑤ Enables cache statistics.

References:

- org.infinispan.configuration.cache.ConfigurationBuilder.

4.3. Configuring Caches Programmatically

Define cache configurations with the Cache Manager.

> The examples in this section use `EmbeddedCacheManager`, which is a Cache Manager that runs in the same JVM as the client.
>
> To configure caches remotely with HotRod clients, you use `RemoteCacheManager`. Refer to the HotRod documentation for more information.

Configure new cache instances

The following example configures a new cache instance:

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-prod.xml");
Cache defaultCache = manager.getCache();
Configuration c = new ConfigurationBuilder().clustering() ①
  .cacheMode(CacheMode.REPL_SYNC) ②
  .build();

String newCacheName = "replicatedCache";
manager.defineConfiguration(newCacheName, c); ③
Cache<String, String> cache = manager.getCache(newCacheName);
```

① Creates a new Configuration object.

② Specifies distributed, synchronous cache mode.

③ Defines a new cache named "replicatedCache" with the Configuration object.

Create new caches from existing configurations

The following examples create new cache configurations from existing ones:

```
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-prod.xml");
Configuration dcc = manager.getDefaultCacheConfiguration(); ①
Configuration c = new ConfigurationBuilder().read(dcc) ②
  .clustering()
  .cacheMode(CacheMode.DIST_SYNC) ③
  .l1()
  .lifespan(60000L) ④
  .build();

String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c); ⑤
Cache<String, String> cache = manager.getCache(newCacheName);
```

① Returns the default cache configuration from the Cache Manager. In this example, `infinispan-prod.xml` defines a replicated cache as the default.

② Creates a new Configuration object that uses the default cache configuration as a base.

③ Specifies distributed, synchronous cache mode.

④ Adds an L1 lifespan configuration.

⑤ Defines a new cache named "distributedWithL1" with the Configuration object.

```java
EmbeddedCacheManager manager = new DefaultCacheManager("infinispan-prod.xml");
Configuration rc = manager.getCacheConfiguration("replicatedCache"); ①
Configuration c = new ConfigurationBuilder().read(rc)
  .clustering()
  .cacheMode(CacheMode.DIST_SYNC)
  .l1()
  .lifespan(60000L)
  .build();

String newCacheName = "distributedWithL1";
manager.defineConfiguration(newCacheName, c);
Cache<String, String> cache = manager.getCache(newCacheName);
```

① Uses a cache configuration named "replicatedCache" as a base.

Reference

- CacheManager package summary
- org.infinispan.configuration.cache.ConfigurationBuilder
- org.infinispan.manager.EmbeddedCacheManager
- HotRod Java Client Guide
- org.infinispan.client.hotrod.configuration.ConfigurationBuilder
- org.infinispan.client.hotrod.RemoteCacheManager

Chapter 5. Configuring the Data Container

Configure the data container where Infinispan stores entries. Specify the encoding for cache entries, store data in off-heap memory, and use eviction or expiration policies to keep only active entries in memory.

5.1. Configuring Encoding for Infinispan Caches

Define the MediaType that Infinispan uses to encode your data when writing and reading to and from the cache.

> When you define a MediaType, you specify the format of your data to Infinispan.
>
> If you want to use the Infinispan Console, Hot Rod clients, and REST clients interchangeably, specify the `application/x-protostream` MediaType so Infinispan encodes data in Protobuf format.

Procedure

- Specify a MediaType for key and values in your Infinispan cache configuration.
  - Declaratively: Set the `encoding` attribute.
  - Programmatically: Use the `encoding()` method.

Declarative examples

- Use the same encoding for keys and values:

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

- Use a different encoding for keys and values:

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

Programmatic examples

- Use the same encoding for keys and values:

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

cfg
  .encoding()
    .mediaType("application/x-protostream")
  .build());
```

- Use a different encoding for keys and values:

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

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

5.1.1. Storing Data in Protobuf Format

Storing data in the cache as Protobuf encoded entries provides a platform independent configuration that enables you to perform cache operations from any client.

> When you configure indexing for Infinispan Search, Infinispan automatically stores keys and values with the `application/x-protostream` media type.

Procedure

1. Specify `application/x-protostream` as the MediaType for keys and values as follows:

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

2. Configure your clients.

Hot Rod clients must register Protocol Buffers schema definitions that describe entities and client marshallers.

Infinispan converts between `application/x-protostream` and `application/json` so REST clients only need to send the following headers to read and write JSON formatted data:

- `Accept: application/json` for read operations.
- `Content-Type: application/json` for write operations.

Reference

- Storage Formats and Client Interoperability

- Using the ProtoStream Marshaller
- Marshalling Custom Java Objects with ProtoStream

5.2. Configuring Infinispan to Store Cache Entries Off Heap

Infinispan can use either JVM heap memory or off-heap native memory as the data container for cache entries. By default Infinispan stores cache entries in JVM heap memory.

Infinispan can use off-heap storage with eviction based on the total number of entries or maximum amount of memory. When using off-heap, Infinispan stores cache entries in Protobuf format with the `application/x-protostream` encoding.

Procedure

To store cache entries in off-heap memory, configure Infinispan in one of the following ways:

- Declaratively: Add the `storage="OFF_HEAP"` attribute to the `memory` element.
- Programmatically: Call the `storage(OFF_HEAP)` method in the `MemoryConfigurationBuilder` class.

Declarative example

```
<local-cache name="off_heap">
  <encoding media-type="application/x-protostream"/>
  <memory storage="OFF_HEAP" max-size="1.5GB" when-full="REMOVE"/>
</local-cache>
```

Programmatic example

```
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg
  .encoding()
    .mediaType("application/x-protostream")
  .memory()
    .storage(StorageType.OFF_HEAP)
    .maxCount(500)
    .whenFull(EvictionStrategy.REMOVE)
  .build());
```

- Infinispan Configuration Schema Reference
- org.infinispan.configuration.cache.MemoryConfigurationBuilder

5.3. Configuring Eviction and Expiration

Eviction and expiration are two strategies for cleaning the data container by removing old, unused entries. Although eviction and expiration are similar, they have some important differences.

- ☑ Eviction lets Infinispan control the size of the data container by removing entries when the container becomes larger than a configured threshold.
- ☑ Expiration limits the amount of time entries can exist. Infinispan uses a scheduler to periodically remove expired entries. Entries that are expired but not yet removed are immediately removed on access; in this case `get()` calls for expired entries return "null" values.
- ☑ Eviction is local to Infinispan nodes.
- ☑ Expiration takes place across Infinispan clusters.
- ☑ You can use eviction and expiration together or independently of each other.
- ☑ You can configure eviction and expiration declaratively in `infinispan.xml` to apply cache-wide defaults for entries.
- ☑ You can explicitly define expiration settings for specific entries but you cannot define eviction on a per-entry basis.
- ☑ You can manually evict entries and manually trigger expiration.

5.3.1. Eviction

Eviction lets you control the size of the data container by removing cache entries to make space when adding new entries.

> Eviction removes entries from memory but not from persistent cache stores. To ensure that entries remain available after Infinispan evicts them, you should configure a persistent cache store.

Infinispan eviction relies on two configurations:

- Maximum size of the data container.
- Strategy for removing entries.

Data container size

Infinispan lets you store entries either in the Java heap or in native memory (off-heap) and set a maximum size for the data container.

You configure the maximum size of the data container in one of two ways:

- Total number of entries (`max-count`).
- Maximum amount of memory (`max-size`).

  To perform eviction based on the amount of memory, you define a maximum size in bytes. For this reason, you must encode entries with a binary storage format such as `application/x-protostream`.

Evicting cache entries

When you configure `memory`, Infinispan approximates the current memory usage of the data container. When entries are added or modified, Infinispan compares the current memory usage of the data container to the maximum size. If the size exceeds the maximum, Infinispan performs eviction.

Eviction happens immediately in the thread that adds an entry that exceeds the maximum size.

Consider the following configuration as an example:

```
<memory max-count="50"/>
```

In this case, the cache can have a total of 50 entries. After the cache reaches the total number of entries, write operations trigger Infinispan to perform eviction.

Eviction strategies

Strategies control how Infinispan performs eviction. You can either perform eviction manually or configure Infinispan to do one of the following:

- Remove old entries to make space for new ones.
- Throw `ContainerFullException` and prevent new entries from being created.

  The exception eviction strategy works only with transactional caches that use 2 phase commits; not with 1 phase commits or synchronization optimizations.

> Infinispan includes the Caffeine caching library that implements a variation of the Least Frequently Used (LFU) cache replacement algorithm known as TinyLFU. For off-heap storage, Infinispan uses a custom implementation of the Least Recently Used (LRU) algorithm.

References

- Caffeine
- Setting Up Persistent Storage

Configuring the Total Number of Entries for Infinispan Caches

Limit the size of the data container for cache entries to a total number of entries.

Procedure

1. Configure your Infinispan cache encoding with an appropriate storage format.
2. Specify the total number of entries that caches can contain before Infinispan performs eviction.
   - Declaratively: Set the `max-count` attribute.
   - Programmatically: Call the `maxCount()` method.
3. Configure an eviction strategy to control how Infinispan removes entries.
   - Declaratively: Set the `when-full` attribute.
   - Programmatically: Call the `whenFull()` method.

Declarative example

```
<local-cache name="maximum_count">
  <encoding media-type="application/x-protostream"/>
  <memory max-count="500" when-full="REMOVE"/>
</local-cache>
```

Programmatic example

```
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg
  .encoding()
    .mediaType("application/x-protostream")
  .memory()
    .maxCount(500)
    .whenFull(EvictionStrategy.REMOVE)
  .build());
```

Reference

- Infinispan Configuration Schema Reference
- org.infinispan.configuration.cache.MemoryConfigurationBuilder

Configuring the Maximum Amount of Memory for Infinispan Caches

Limit the size of the data container for cache entries to a maximum amount of memory.

Procedure

1. Configure your Infinispan cache to use a storage format that supports binary encoding.

   You must use a binary storage format to perform eviction based on the maximum amount of memory.

2. Configure the maximum amount of memory, in bytes, that caches can use before Infinispan performs eviction.
   - Declaratively: Set the `max-size` attribute.
   - Programmatically: Use the `maxSize()` method.
3. Optionally specify a byte unit of measurement. The default is B (bytes). Refer to the configuration schema for supported units.
4. Configure an eviction strategy to control how Infinispan removes entries.
   - Declaratively: Set the `when-full` attribute.
   - Programmatically: Use the `whenFull()` method.

Declarative example

```
<local-cache name="maximum_size">
  <encoding media-type="application/x-protostream"/>
  <memory max-size="1.5GB" when-full="REMOVE"/>
</local-cache>
```

Programmatic example

```
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg
  .encoding()
    .mediaType("application/x-protostream")
  .memory()
    .maxSize("1.5GB")
    .whenFull(EvictionStrategy.REMOVE)
  .build());
```

Reference

- Infinispan Configuration Schema Reference
- org.infinispan.configuration.cache.EncodingConfiguration
- org.infinispan.configuration.cache.MemoryConfigurationBuilder
- MediaType Configuration

Eviction Examples

You configure eviction as part of your cache definition.

Default memory configuration

Eviction is not enabled, which is the default configuration. Infinispan stores cache entries as objects in the JVM heap.

```
<memory />
```

Eviction based on the total number of entries

Infinispan stores cache entries as objects in the JVM heap. Eviction happens when there are 100 entries in the data container and Infinispan gets a request to create a new entry:

```
<memory max-count="100"/>
```

Eviction based maximum size in bytes

Infinispan stores cache entries as `byte[]` arrays if you encode entries with binary storage formats, for example: `application/x-protostream` format.

In the following example, Infinispan performs eviction when the size of the data container reaches 500 MB (megabytes) in size and it gets a request to create a new entry:

```
<encoding media-type="application/x-protostream"/> ①
<memory max-size="500 MB"/> ②
```

① Specifies a binary format for entries in the cache.

② Defines the maximum size of the data container as MB (megabytes).

Off-heap storage

Infinispan stores cache entries as bytes in native memory. Eviction happens when there are 100 entries in the data container and Infinispan gets a request to create a new entry:

```
<memory storage="OFF_HEAP" max-count="100"/>
```

Off-heap storage with the exception strategy

Infinispan stores cache entries as bytes in native memory. When there are 100 entries in the data container, and Infinispan gets a request to create a new entry, it throws an exception and does not allow the new entry:

```
<transaction mode="NONE"/>

<memory storage="OFF_HEAP" max-count="100" when-full="EXCEPTION"/>
```

Manual eviction

Infinispan stores cache entries as objects in the JVM heap. Eviction is not enabled but performed manually using the `evict()` method.

> This configuration prevents a warning message when you enable passivation but do not configure eviction.

```
<memory when-full="MANUAL"/>
```

Passivation with eviction

Passivation persists data to cache stores when Infinispan evicts entries. You should always enable eviction if you enable passivation, for example:

```
<persistence passivation="true">
  ...
</persistence>

<memory max-count="100"/>
```

References

- Passivation

Custom Classes with Memory-Based Eviction

You must use binary or off-heap storage memory based eviction, as in the following examples:

Declarative configuration

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

Programmatic configuration

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

5.3.2. Expiration

Expiration removes entries from caches when they reach one of the following time limits:

Lifespan

Sets the maximum amount of time that entries can exist.

Maximum idle

Specifies how long entries can remain idle. If operations do not occur for entries, they become idle.

> Maximum idle expiration does not currently support cache configurations with persistent cache stores.
>
> When using expiration with an exception-based eviction policy, entries that are expired but not yet removed from the cache count towards the size of the data container.

How Expiration Works

When you configure expiration, Infinispan stores keys with metadata that determines when entries expire.

- Lifespan uses a `creation` timestamp and the value for the `lifespan` configuration property.
- Maximum idle uses a `last used` timestamp and the value for the `max-idle` configuration

property.

Infinispan checks if lifespan or maximum idle metadata is set and then compares the values with the current time.

If `(creation + lifespan > currentTime)` or `(lastUsed + maxIdle > currentTime)` then Infinispan detects that the entry is expired.

Expiration occurs whenever entries are accessed or found by the expiration reaper.

For example, `k1` reaches the maximum idle time and a client makes a `Cache.get(k1)` request. In this case, Infinispan detects that the entry is expired and removes it from the data container. The `Cache.get()` returns `null`.

Infinispan also expires entries from cache stores, but only with lifespan expiration. Maximum idle expiration does not work with cache stores. In the case of cache loaders, Infinispan cannot expire entries because loaders can only read from external storage.

> Infinispan adds expiration metadata as `long` primitive data types to cache entries. This can increase the size of keys by as much as 32 bytes.

Expiration Reaper

Infinispan uses a reaper thread that runs periodically to detect and remove expired entries. The expiration reaper ensures that expired entries that are no longer accessed are removed.

The Infinispan `ExpirationManager` interface handles the expiration reaper and exposes the `processExpiration()` method.

In some cases, you can disable the expiration reaper and manually expire entries by calling `processExpiration()`; for instance, if you are using local cache mode with a custom application where a maintenance thread runs periodically.

> If you use clustered cache modes, you should never disable the expiration reaper.
>
> Infinispan always uses the expiration reaper when using cache stores. In this case you cannot disable it.

Reference

- org.infinispan.configuration.cache.ExpirationConfigurationBuilder
- org.infinispan.expiration.ExpirationManager

Maximum Idle and Clustered Caches

Because maximum idle expiration relies on the last access time for cache entries, it has some limitations with clustered cache modes.

With lifespan expiration, the creation time for cache entries provides a value that is consistent across clustered caches. For example, the creation time for `k1` is always the same on all nodes.

For maximum idle expiration with clustered caches, last access time for entries is not always the same on all nodes. To ensure that entries have the same relative access times across clusters, Infinispan sends touch commands to all owners when keys are accessed.

The touch commands that Infinispan send have the following considerations:

- `Cache.get()` requests do not return until all touch commands complete. This synchronous behavior increases latency of client requests.
- The touch command also updates the "recently accessed" metadata for cache entries on all owners, which Infinispan uses for eviction.
- With scattered cache mode, Infinispan sends touch commands to all nodes, not just primary and backup owners.

Additional information

- Maximum idle expiration does not work with invalidation mode.
- Iteration across a clustered cache can return expired entries that have exceeded the maximum idle time limit. This behavior ensures performance because no remote invocations are performed during the iteration. Also note that iteration does not refresh any expired entries.

Expiration Examples

When you configure Infinispan to expire entries, you can set lifespan and maximum idle times for:

- All entries in a cache (cache-wide). You can configure cache-wide expiration in `infinispan.xml` or programmatically using the `ConfigurationBuilder`.
- Per entry, which takes priority over cache-wide expiration values. You configure expiration for specific entries when you create them.

> When you explicitly define lifespan and maximum idle time values for cache entries, Infinispan replicates those values across the cluster along with the cache entries. Likewise, Infinispan persists expiration values along with the entries if you configure cache stores.

Configuring expiration for all cache entries

Expire all cache entries after 2 seconds:

```
<expiration lifespan="2000" />
```

Expire all cache entries 1 second after last access time:

```
<expiration max-idle="1000" />
```

Disable the expiration reaper with the `interval` attribute and manually expire entries 1 second after last access time:

```
<expiration max-idle="1000" interval="-1" />
```

Expire all cache entries after 5 seconds or 1 second after the last access time, whichever happens first:

```
<expiration lifespan="5000" max-idle="1000" />
```

Configuring expiration when creating cache entries

The following example shows how to configure lifespan and maximum idle values when creating cache entries:

```
// Use the cache-wide expiration configuration.
cache.put("pinot noir", pinotNoirPrice); ①

// Define a lifespan value of 2.
cache.put("chardonnay", chardonnayPrice, 2, TimeUnit.SECONDS); ②

// Define a lifespan value of -1 (disabled) and a max-idle value of 1.
cache.put("pinot grigio", pinotGrigioPrice,
          -1, TimeUnit.SECONDS, 1, TimeUnit.SECONDS); ③

// Define a lifespan value of 5 and a max-idle value of 1.
cache.put("riesling", rieslingPrice,
          5, TimeUnit.SECONDS, 1, TimeUnit.SECONDS); ④
```

If the Infinispan configuration defines a lifespan value of `1000` for all entries, the preceding `Cache.put()` requests cause the entries to expire:

① After 1 second.

② After 2 seconds.

③ 1 second after last access time.

④ After 5 seconds or 1 second after the last access time, whichever happens first.

Reference

- Infinispan Configuration Schema
- org.infinispan.configuration.cache.ExpirationConfigurationBuilder
- org.infinispan.expiration.ExpirationManager

Chapter 6. Configuring Statistics, Metrics, and JMX

Enable statistics that Infinispan exports to a metrics endpoint or via JMX MBeans. You can also register JMX MBeans to perform management operations.

6.1. Enabling Infinispan Statistics

Infinispan lets you enable statistics for Cache Managers and caches. However, enabling statistics for a Cache Manager does not enable statistics for the caches that it controls. You must explicitly enable statistics for your caches.

> Infinispan server enables statistics for Cache Managers by default.

Procedure

- Enable statistics declaratively or programmatically.

Declaratively

```
<cache-container statistics="true"> ①
  <local-cache name="mycache" statistics="true"/> ②
</cache-container>
```

① Enables statistics for the Cache Manager.

② Enables statistics for the named cache.

Programmatically

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .cacheContainer().statistics(true) ①
  .build();

 ...

Configuration config = new ConfigurationBuilder()
  .statistics().enable() ②
  .build();
```

① Enables statistics for the Cache Manager.

② Enables statistics for the named cache.

6.2. Enabling Infinispan Metrics

Configure Infinispan to export gauges and histograms.

Procedure

- Configure metrics declaratively or programmatically.

Declaratively

```
<cache-container statistics="true"> ①
  <metrics gauges="true" histograms="true" /> ②
</cache-container>
```

① Computes and collects statistics about the Cache Manager.

② Exports collected statistics as gauge and histogram metrics.

Programmatically

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .statistics().enable() ①
  .metrics().gauges(true).histograms(true) ②
  .build();
```

① Computes and collects statistics about the Cache Manager.

② Exports collected statistics as gauge and histogram metrics.

6.2.1. Infinispan Metrics

Infinispan is compatible with the Eclipse MicroProfile Metrics API and can generate gauge and histogram metrics.

- Infinispan metrics are provided at the `vendor` scope. Metrics related to the JVM are provided in the `base` scope for Infinispan server.
- Gauges provide values such as the average number of nanoseconds for write operations or JVM uptime. Gauges are enabled by default. If you enable statistics, Infinispan automatically generates gauges.
- Histograms provide details about operation execution times such as read, write, and remove times. Infinispan does not enable histograms by default because they require additional computation.

Reference

- Eclipse MicroProfile Metrics

6.3. Configuring Infinispan to Register JMX MBeans

Infinispan can register JMX MBeans that you can use to collect statistics and perform administrative operations. However, you must enable statistics separately to JMX otherwise Infinispan provides `0` values for all statistic attributes.

Procedure

- Enable JMX declaratively or programmatically.

Declaratively

```
<cache-container>
  <jmx enabled="true" /> ①
</cache-container>
```

① Registers Infinispan JMX MBeans.

Programmatically

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .jmx().enable() ①
  .build();
```

① Registers Infinispan JMX MBeans.

6.3.1. Naming Multiple Cache Managers

In cases where multiple Infinispan Cache Managers run on the same JVM, you should uniquely identify each Cache Manager to prevent conflicts.

Procedure

- Uniquely identify each cache manager in your environment.

For example, the following examples specify "Hibernate2LC" as the cache manager name, which results in a JMX MBean named `org.infinispan:type=CacheManager,name="Hibernate2LC"`.

Declaratively

```
<cache-container name="Hibernate2LC">
  <jmx enabled="true" />
  ...
</cache-container>
```

Programmatically

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .cacheManagerName("Hibernate2LC")
  .jmx().enable()
  .build();
```

Reference

- GlobalConfigurationBuilder
- Infinispan Configuration Schema

6.3.2. Registering MBeans In Custom MBean Servers

Infinispan includes an `MBeanServerLookup` interface that you can use to register MBeans in custom

MBeanServer instances.

Procedure

1. Create an implementation of `MBeanServerLookup` so that the `getMBeanServer()` method returns the custom MBeanServer instance.
2. Configure Infinispan with the fully qualified name of your class, as in the following example:

Declaratively

```xml
<cache-container>
   <jmx enabled="true" mbean-server-lookup="com.acme.MyMBeanServerLookup" />
</cache-container>
```

Programmatically

```java
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .jmx().enable().mBeanServerLookup(new com.acme.MyMBeanServerLookup())
  .build();
```

Reference

- Infinispan Configuration Schema
- MBeanServerLookup

6.3.3. Infinispan MBeans

Infinispan exposes JMX MBeans that represent manageable resources.

`org.infinispan:type=Cache`
: Attributes and operations available for cache instances.

`org.infinispan:type=CacheManager`
: Attributes and operations available for cache managers, including Infinispan cache and cluster health statistics.

For a complete list of available JMX MBeans along with descriptions and available operations and attributes, see the *Infinispan JMX Components* documentation.

Reference

Infinispan JMX Components

Chapter 7. Configuring Infinispan Clustering

Infinispan uses the JGroups protocol so nodes can discover each other on the network and form clusters. When Infinispan nodes form clusters, they also use JGroups for inter-cluster and intra-cluster communication.

7.1. Setting Up Cluster Transport

Infinispan nodes rely on a transport layer to join and leave clusters as well as to replicate data across the network.

Infinispan uses JGroups technology to handle cluster transport. You configure cluster transport with JGroups stacks, which define properties for either UDP or TCP protocols.

7.1.1. Getting Started with Default Stacks

Use default JGroups stacks with recommended settings as a starting point for your cluster transport layer.

Declaratively

- Specify default JGroups stacks with the `stack` attribute.

```
<infinispan>
  <cache-container default-cache="replicatedCache">
    <transport stack="udp" /> ①
    ...
  </cache-container>
</infinispan>
```

① uses the `default-jgroups-udp.xml` stack for cluster transport.

Programmatically

- Specify default JGroups stacks with the `addProperty()` method.

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .defaultTransport()
        .clusterName("qa-cluster")
        .addProperty("configurationFile", "default-jgroups-udp.xml") ①
        .build();
```

① uses the `default-jgroups-udp.xml` stack for cluster transport.

Use the `cluster-stack` argument with the Infinispan server startup script.

```
$ bin/server.sh --cluster-stack=tcp
```

Default JGroups Stacks

Default JGroups stacks are included in `infinispan-core.jar` and on the classpath. You can locate the default JGroups stacks in the `default-configs` directory.

File name	Stack name	Description
`default-jgroups-udp.xml`	`udp`	Uses UDP for transport and UDP multicast for discovery. Suitable for larger clusters (over 100 nodes) or if you are using replicated caches or invalidation mode. Minimizes the number of open sockets.
`default-jgroups-tcp.xml`	`tcp`	Uses TCP for transport and the `MPING` protocol for discovery, which uses `UDP` multicast. Suitable for smaller clusters (under 100 nodes) *only if* you are using distributed caches because TCP is more efficient than UDP as a point-to-point protocol.
`default-jgroups-ec2.xml`	`ec2`	Uses TCP for transport and `S3_PING` for discovery. Suitable for Amazon EC2 nodes where UDP multicast is not available.
`default-jgroups-kubernetes.xml`	`kubernetes`	Uses TCP for transport and `DNS_PING` for discovery. Suitable for Kubernetes and Red Hat OpenShift nodes where UDP multicast is not always available.
`default-jgroups-google.xml`	`google`	Uses TCP for transport and `GOOGLE_PING2` for discovery. Suitable for Google Cloud Platform nodes where UDP multicast is not available.
`default-jgroups-azure.xml`	`azure`	Uses TCP for transport and `AZURE_PING` for discovery. Suitable for Microsoft Azure nodes where UDP multicast is not available.

Reference

- JGroups Protocol
- JGroups Discovery Protocols

7.1.2. Using Inline JGroups Stacks

Custom JGroups stacks can help you optimize network performance for Infinispan clusters compared to using the default stacks.

Procedure

- Embed your custom JGroups stack definitions in `infinispan.xml` as in the following example:

```
<infinispan> ①
  <jgroups> ②
    <stack name="prod"> ③
      <TCP bind_port="7800" port_range="30" recv_buf_size="20000000" send_buf_size
="640000"/>
      <MPING bind_addr="127.0.0.1" break_on_coord_rsp="true"
             mcast_addr="${jgroups.mping.mcast_addr:228.2.4.6}"
             mcast_port="${jgroups.mping.mcast_port:43366}"
             num_discovery_runs="3"
             ip_ttl="${jgroups.udp.ip_ttl:2}"/>
      <MERGE3 />
      <FD_SOCK />
      <FD_ALL timeout="3000" interval="1000" timeout_check_interval="1000" />
      <VERIFY_SUSPECT timeout="1000" />
      <pbcast.NAKACK2 use_mcast_xmit="false" xmit_interval="100"
xmit_table_num_rows="50"
                      xmit_table_msgs_per_row="1024"
xmit_table_max_compaction_time="30000" />
      <UNICAST3 xmit_interval="100" xmit_table_num_rows="50"
xmit_table_msgs_per_row="1024"
                xmit_table_max_compaction_time="30000" />
      <pbcast.STABLE stability_delay="200" desired_avg_gossip="2000" max_bytes="1M"
/>
      <pbcast.GMS print_local_addr="false" join_timeout=
"${jgroups.join_timeout:2000}" />
      <UFC max_credits="4m" min_threshold="0.40" />
      <MFC max_credits="4m" min_threshold="0.40" />
      <FRAG3 />
    </stack>
  </jgroups>
  <cache-container default-cache="replicatedCache"> ④
    <transport stack="prod" /> ⑤
    ...
  </cache-container>
</infinispan>
```

① root element of `infinispan.xml`.

② contains JGroups stack definitions.

③ defines a JGroups stack named "prod".

④ configures a Infinispan Cache Manager and names the "replicatedCache" cache definition as the default.

⑤ uses the "prod" JGroups stack for cluster transport.

> Use inheritance with inline JGroups stacks to tune and customize specific transport properties.

Reference

- Adjusting and Tuning JGroups Stacks
- Infinispan Configuration Schema

7.1.3. Adjusting and Tuning JGroups Stacks

Use inheritance to combine, extend, remove, and replace specific properties in the default JGroups stacks or custom configurations.

Procedure

1. Add a new JGroups stack declaration.
2. Name a parent stack with the `extends` attribute.

   > Use the default JGroups stacks as parents. For example, `extends="tcp"` to tunes the default TCP stack.

3. Modify transport properties with the `stack.combine` attribute.

Stack Combine Attribute

`stack.combine` modifies inherited JGroups properties.

`stack.position` identifies protocols to modify. If you do not specify `stack.position`, Infinispan defaults to the same protocol as the inherited configuration, which resets all non-specified attributes to the default values.

Value	Description
`COMBINE`	Overrides existing protocol attributes.
`REPLACE`	Replaces existing protocols.
`INSERT_AFTER`	Inserts protocols into the JGroups stack.
`REMOVE`	Removes protocols from the inherited configuration.

For example, evaluate a Gossip router for cluster discovery using a TCP stack configuration named "prod":

```
<jgroups>
  ...
  <stack name="gossip-prod" extends="prod"> ①
    <TCPGOSSIP initial_hosts="${jgroups.tunnel.gossip_router_hosts:localhost[12001]}"
             stack.combine="REPLACE" stack.position="MPING" /> ②
    <FD_SOCK stack.combine="REMOVE"/> ③
    <VERIFY_SUSPECT timeout="2000"/> ④
    <SYM_ENCRYPT sym_algorithm="AES"
                 key_store_name="defaultStore.keystore"
                 store_password="changeit"
                 alias="myKey" stack.combine="INSERT_AFTER"
                 stack.position="pbcast.NAKACK2" /> ⑤
  </stack>
  ...
</jgroups>
```

① creates a new stack named "gossip-prod" that inherits from "prod".

② uses `TCPGOSSIP` discovery instead of `MPING`.

③ removes `FD_SOCK` from the "gossip-prod" stack.

④ increases the `VERIFY_SUSPECT` timeout.

⑤ adds `SYM_ENCRYPT` for cluster security to the stack after the `pbcast.NAKACK2` protocol.

7.1.4. Using JGroups Stacks in External Files

You can use custom JGroups transport configuration from external files.

Infinispan looks for JGroups configuration files on your classpath first and then for absolute path names.

Declaratively

- Specify your JGroups transport configuration with the `stack-file` element.

```
<infinispan>
  <jgroups>
     <stack-file name="prod-tcp" path="prod-jgroups-tcp.xml"/> ①
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <transport stack="prod-tcp" /> ②
    <replicated-cache name="replicatedCache"/>
  </cache-container>
  ...
</infinispan>
```

① adds the "prod-jgroups-tcp.xml" stack definition.

② configures Infinispan cluster to use the stack.

Programmatically

- Specify your JGroups transport configuration with the `addProperty()` method.

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .defaultTransport()
        .clusterName("prod-cluster")
        .addProperty("configurationFile", "prod-jgroups-tcp.xml") ①
        .build();
```

① adds the "prod-jgroups-tcp.xml" stack definition as the default for the cluster transport.

Reference

- GlobalConfigurationBuilder.transport()
- TransportConfigurationBuilder

7.1.5. Tuning JGroups Stacks with System Properties

Pass system properties to the JVM at startup to tune JGroups stacks.

For example, to change the `TCP` port and IP address do the following:

```
$ java -cp ... -Djgroups.tcp.port=1234 -Djgroups.tcp.address=192.0.2.0
```

System Properties for Default JGroups Stacks

Use system properties with default JGroups stacks.

Common Properties

The following system properties apply to all JGroups stacks, including `default-jgroups-udp.xml` and `default-jgroups-tcp.xml`:

System Property	Description	Default Value	Required/Optional
`jgroups.bind.address`	Bind address for cluster transport.	`SITE_LOCAL`	Optional
`jgroups.bind.port`	Bind port for the socket.	`7800`	Optional
`jgroups.mcast_addr`	IP address for multicast, both discovery and inter-cluster communication. The IP address must be a valid "class D" address that is suitable for IP multicast.	`228.6.7.8`	Optional
`jgroups.mcast_port`	Port for the multicast socket.	`46655`	Optional

System Property	Description	Default Value	Required/Optional
`jgroups.ip_ttl`	Time-to-live (TTL) for IP multicast packets. The value defines the number of network hops a packet can make before it is dropped.	2	Optional
`jgroups.thread_pool.min_threads`	Minimum number of threads for the thread pool.	0	Optional
`jgroups.thread_pool.max_threads`	Maximum number of threads for the thread pool.	200	Optional
`jgroups.join_timeout`	Maximum number of milliseconds to wait for join requests to succeed.	2000	Optional

Amazon EC3

The following system properties apply to `default-jgroups-ec2.xml`:

System Property	Description	Default Value	Required/Optional
`jgroups.s3.access_key`	Amazon S3 access key for an S3 bucket.	No default value.	Optional
`jgroups.s3.secret_access_key`	Amazon S3 secret key used for an S3 bucket.	No default value.	Optional
`jgroups.s3.bucket`	Name of the Amazon S3 bucket. The name must exist and be unique.	No default value.	Optional

Kubernetes

The following system properties apply to `default-jgroups-kubernetes.xml`:

System Property	Description	Default Value	Required/Optional
`jgroups.dns.query`	Sets the DNS record that returns cluster members.	N/A	Required if you do not set the `dns_query` parameter.

Google Cloud Platform

The following system properties apply to `default-jgroups-google.xml`:

System Property	Description	Default Value	Required/Optional
`jgroups.google.bucket_name`	Name of the Google Compute Engine bucket. The name must exist and be unique.	N/A	Required if you do not set the `dns_query` parameter.

Reference

- JGroups System Properties
- JGroups Protocol List

7.1.6. Using Custom JChannels

Construct custom JGroups JChannels as in the following example:

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
JChannel jchannel = new JChannel();
// Configure the jchannel to your needs.
JGroupsTransport transport = new JGroupsTransport(jchannel);
global.transport().transport(transport);
new DefaultCacheManager(global.build());
```

Infinispan cannot use custom JChannels that are already connected.

Reference

JGroups JChannel

7.1.7. TCP and UDP Ports for Cluster Traffic

Infinispan uses the following ports by default:

Default Port	Protocol	Description
`7800`	TCP/UDP	JGroups cluster bind port
`46655`	UDP	JGroups multicast
`7200`	TCP	JGroups RELAY2 for cross-site replication

Reference

Setting Up Cluster Transport

7.2. Configuring Cluster Discovery

Running Infinispan on hosted services requires using discovery mechanisms that are adapted to network constraints that individual cloud providers impose. For instance, Amazon EC2 does not allow UDP multicast.

Infinispan can use the following cloud discovery mechanisms:

- Generic discovery protocols (`TCPPING` and `TCPGOSSIP`)
- JGroups PING protocols (`KUBE_PING` and `DNS_PING`)
- Cloud-specific PING protocols

> Embedded Infinispan requires cloud provider dependencies.

7.2.1. TCPPING

`TCPPING` is a generic JGroups discovery mechanism that uses a static list of IP addresses for cluster members.

To use `TCPPING`, you must add the list of static IP addresses to the JGroups configuration file for each Infinispan node. However, the drawback to `TCPPING` is that it does not allow nodes to dynamically join Infinispan clusters.

TCPPING configuration example

```
<config>
      <TCP bind_port="7800" />
      <TCPPING timeout="3000"
           initial_hosts=
"${jgroups.tcpping.initial_hosts:localhost[7800],localhost[7801]}"
           port_range="1"
           num_initial_members="3"/>
...
...
</config>
```

Reference

JGroups TCPPING

7.2.2. Gossip Router

Gossip routers provide a centralized location on the network from which your Infinispan cluster can retrieve addresses of other nodes.

You inject the address (`IP:PORT`) of the Gossip router into Infinispan nodes as follows:

1. Pass the address as a system property to the JVM; for example, `-DGossipRouterAddress="10.10.2.4[12001]"`.
2. Reference that system property in the JGroups configuration file.

Gossip router configuration example

```
<config>
    <TCP bind_port="7800" />
    <TCPGOSSIP timeout="3000" initial_hosts="${GossipRouterAddress}"
num_initial_members="3" />
...
...
</config>
```

Reference

JGroups Gossip Router

7.2.3. DNS_PING

JGroups `DNS_PING` queries DNS servers to discover Infinispan cluster members in Kubernetes environments such as OKD and Red Hat OpenShift.

DNS_PING configuration example

```
<stack name="dns-ping">
...
    <dns.DNS_PING
      dns_query="myservice.myproject.svc.cluster.local" />
...
</stack>
```

Reference

- JGroups DNS_PING
- DNS for Services and Pods (Kubernetes documentation for adding DNS entries)

7.2.4. KUBE_PING

JGroups `Kube_PING` uses a Kubernetes API to discover Infinispan cluster members in environments such as OKD and Red Hat OpenShift.

KUBE_PING configuration example

```
<config>
    <TCP bind_addr="${match-interface:eth.*}" />
    <kubernetes.KUBE_PING />
...
...
</config>
```

KUBE_PING configuration requirements

- Your `KUBE_PING` configuration must bind the JGroups stack to the `eth0` network interface. Docker containers use eth0 for communication.

- KUBE_PING uses environment variables inside containers for configuration. The KUBERNETES_NAMESPACE environment variable must specify a valid namespace. You can either hardcode it or populate it via the Kubernetes Downward API.
- KUBE_PING requires additional privileges on Red Hat OpenShift. Assuming that oc project -q returns the current namespace and default is the service account name, you can run:

```
$ oc policy add-role-to-user view system:serviceaccount:$(oc project -q):default -n
$(oc project -q)
```

Reference

- JGroups Kube_PING
- Kubernetes Downward API
- Docker Networking

7.2.5. NATIVE_S3_PING

On Amazon Web Service (AWS), use the S3_PING protocol for discovery.

You can configure JGroups to use shared storage to exchange the details of Infinispan nodes. NATIVE_S3_PING allows Amazon S3 as the shared storage but requires both Amazon S3 and EC2 subscriptions.

NATIVE_S3_PING configuration example

```
<config>
    <TCP bind_port="7800" />
    <org.jgroups.aws.s3.NATIVE_S3_PING
            region_name="replace this with your region (e.g. eu-west-1)"
            bucket_name="replace this with your bucket name"
            bucket_prefix="replace this with a prefix to use for entries in the bucket
(optional)" />
</config>
```

NATIVE_S3_PING dependencies for embedded Infinispan

```
<dependency>
  <groupId>org.jgroups.aws.s3</groupId>
  <artifactId>native-s3-ping</artifactId>
  <version>${version.jgroups.native_s3_ping}</version>
</dependency>
```

7.2.6. JDBC_PING

JDBC_PING uses JDBC connections to shared databases, such as Amazon RDS on EC2, to store information about Infinispan nodes.

Reference

JDBC_PING Wiki

7.2.7. AZURE_PING

On Microsoft Azure, use a generic discovery protocol or `AZURE_PING`, which uses shared Azure Blob Storage to store discovery information.

AZURE_PING configuration example

```
<azure.AZURE_PING
    storage_account_name="replace this with your account name"
    storage_access_key="replace this with your access key"
    container="replace this with your container name"
/>
```

AZURE_PING dependencies for embedded Infinispan

```
<dependency>
  <groupId>org.jgroups.azure</groupId>
  <artifactId>jgroups-azure</artifactId>
  <version>${version.jgroups.azure}</version>
</dependency>
```

7.2.8. GOOGLE2_PING

On Google Compute Engine (GCE), use a generic discovery protocol or `GOOGLE2_PING`, which uses Google Cloud Storage (GCS) to store information about the cluster members.

GOOGLE2_PING configuration example

```
<org.jgroups.protocols.google.GOOGLE_PING2 location="${jgroups.google.bucket_name}" />
```

GOOGLE2_PING dependencies for embedded Infinispan

```
<dependency>
  <groupId>org.jgroups.google</groupId>
  <artifactId>jgroups-google</artifactId>
  <version>${version.jgroups.google}</version>
</dependency>
```

7.3. Encrypting Cluster Transport

Secure cluster transport so that nodes communicate with encrypted messages. You can also configure Infinispan clusters to perform certificate authentication so that only nodes with valid identities can join.

7.3.1. Infinispan Cluster Security

To secure cluster traffic, you configure Infinispan nodes to encrypt JGroups message payloads with secret keys.

Infinispan nodes can obtain secret keys from either:

- The coordinator node (asymmetric encryption).
- A shared keystore (symmetric encryption).

Retrieving secret keys from coordinator nodes

You configure asymmetric encryption by adding the `ASYM_ENCRYPT` protocol to a JGroups stack in your Infinispan configuration. This allows Infinispan clusters to generate and distribute secret keys.

> **!** When using asymmetric encryption, you should also provide keystores so that nodes can perform certificate authentication and securely exchange secret keys. This protects your cluster from man-in-the-middle (MitM) attacks.

Asymmetric encryption secures cluster traffic as follows:

1. The first node in the Infinispan cluster, the coordinator node, generates a secret key.
2. A joining node performs certificate authentication with the coordinator to mutually verify identity.
3. The joining node requests the secret key from the coordinator node. That request includes the public key for the joining node.
4. The coordinator node encrypts the secret key with the public key and returns it to the joining node.
5. The joining node decrypts and installs the secret key.
6. The node joins the cluster, encrypting and decrypting messages with the secret key.

Retrieving secret keys from shared keystores

You configure symmetric encryption by adding the `SYM_ENCRYPT` protocol to a JGroups stack in your Infinispan configuration. This allows Infinispan clusters to obtain secret keys from keystores that you provide.

1. Nodes install the secret key from a keystore on the Infinispan classpath at startup.
2. Node join clusters, encrypting and decrypting messages with the secret key.

Comparison of asymmetric and symmetric encryption

`ASYM_ENCRYPT` with certificate authentication provides an additional layer of encryption in comparison with `SYM_ENCRYPT`. You provide keystores that encrypt the requests to coordinator nodes for the secret key. Infinispan automatically generates that secret key and handles cluster traffic, while letting you specify when to generate secret keys. For example, you can configure clusters to generate new secret keys when nodes leave. This ensures that nodes cannot bypass certificate authentication and join with old keys.

SYM_ENCRYPT, on the other hand, is faster than ASYM_ENCRYPT because nodes do not need to exchange keys with the cluster coordinator. A potential drawback to SYM_ENCRYPT is that there is no configuration to automatically generate new secret keys when cluster membership changes. Users are responsible for generating and distributing the secret keys that nodes use to encrypt cluster traffic.

7.3.2. Configuring Cluster Transport with Asymmetric Encryption

Configure Infinispan clusters to generate and distribute secret keys that encrypt JGroups messages.

Procedure

1. Create a keystore with certificate chains that enables Infinispan to verify node identity.
2. Place the keystore on the classpath for each node in the cluster.

   For Infinispan Server, you put the keystore in the $ISPN_HOME directory.

3. Add the SSL_KEY_EXCHANGE and ASYM_ENCRYPT protocols to a JGroups stack in your Infinispan configuration, as in the following example:

```
<infinispan>
    <jgroups>
        <stack name="encrypt-tcp" extends="tcp"> ①
          <SSL_KEY_EXCHANGE keystore_name="mykeystore.jks" ②
                            keystore_password="changeit" ③
                            stack.combine="INSERT_AFTER"
                            stack.position="VERIFY_SUSPECT"/> ④
          <ASYM_ENCRYPT asym_keylength="2048" ⑤
                        asym_algorithm="RSA" ⑥
                        change_key_on_coord_leave = "false" ⑦
                        change_key_on_leave = "false" ⑧
                        use_external_key_exchange = "true" ⑨
                        stack.combine="INSERT_BEFORE"
                        stack.position="pbcast.NAKACK2"/> ⑩
        </stack>
    </jgroups>
    <cache-container name="default" statistics="true">
      <transport cluster="${infinispan.cluster.name}"
                 stack="encrypt-tcp" ⑪
                 node-name="${infinispan.node.name:}"/>
   </cache-container>
</infinispan>
```

① Creates a secure JGroups stack named "encrypt-tcp" that extends the default TCP stack for Infinispan.

② Names the keystore that nodes use to perform certificate authentication.

③ Specifies the keystore password.

④ Uses the stack.combine and stack.position attributes to insert SSL_KEY_EXCHANGE into the default TCP stack after the VERIFY_SUSPECT protocol.

⑤ Specifies the length of the secret key that the coordinator node generates. The default value is 2048.

⑥ Specifies the cipher engine the coordinator node uses to generate secret keys. The default value is RSA.

⑦ Configures Infinispan to generate and distribute a new secret key when the coordinator node changes.

⑧ Configures Infinispan to generate and distribute a new secret key when nodes leave.

⑨ Configures Infinispan nodes to use the SSL_KEY_EXCHANGE protocol for certificate authentication.

⑩ Uses the stack.combine and stack.position attributes to insert ASYM_ENCRYPT into the default TCP stack before the pbcast.NAKACK2 protocol.

⑪ Configures the Infinispan cluster to use the secure JGroups stack.

Verification

When you start your Infinispan cluster, the following log message indicates that the cluster is using the secure JGroups stack:

```
[org.infinispan.CLUSTER] ISPN000078: Starting JGroups channel cluster with stack
<encrypted_stack_name>
```

Infinispan nodes can join the cluster only if they use ASYM_ENCRYPT and can obtain the secret key from the coordinator node. Otherwise the following message is written to Infinispan logs:

```
[org.jgroups.protocols.ASYM_ENCRYPT] <hostname>: received message without encrypt
header from <hostname>; dropping it
```

Reference

The example ASYM_ENCRYPT configuration in this procedure shows commonly used parameters. Refer to JGroups documentation for the full set of available parameters.

- JGroups 4 Manual
- JGroups 4.2 Schema

7.3.3. Configuring Cluster Transport with Symmetric Encryption

Configure Infinispan clusters to encrypt JGroups messages with secret keys from keystores that you provide.

Procedure

1. Create a keystore that contains a secret key.

2. Place the keystore on the classpath for each node in the cluster.

   For Infinispan Server, you put the keystore in the $ISPN_HOME directory.

3. Add the `SYM_ENCRYPT` protocol to a JGroups stack in your Infinispan configuration, as in the following example:

```
<infinispan>
    <jgroups>
         <stack name="encrypt-tcp" extends="tcp"> ①
           <SYM_ENCRYPT keystore_name="myKeystore.p12" ②
                        keystore_type="PKCS12" ③
                        store_password="changeit" ④
                        key_password="changeit" ⑤
                        alias="myKey" ⑥
                        stack.combine="INSERT_AFTER"
                        stack.position="VERIFY_SUSPECT"/> ⑦
         </stack>
    </jgroups>
    <cache-container name="default" statistics="true">
      <transport cluster="${infinispan.cluster.name}"
                 stack="encrypt-tcp" ⑧
                 node-name="${infinispan.node.name:}"/>
   </cache-container>
</infinispan>
```

① Creates a secure JGroups stack named "encrypt-tcp" that extends the default TCP stack for Infinispan.

② Names the keystore from which nodes obtain secret keys.

③ Specifies the keystore type. JGroups uses JCEKS by default.

④ Specifies the keystore password.

⑤ Specifies the secret key password.

⑥ Specifies the secret key alias.

⑦ Uses the `stack.combine` and `stack.position` attributes to insert `SYM_ENCRYPT` into the default TCP stack after the `VERIFY_SUSPECT` protocol.

⑧ Configures the Infinispan cluster to use the secure JGroups stack.

Verification

When you start your Infinispan cluster, the following log message indicates that the cluster is using the secure JGroups stack:

```
[org.infinispan.CLUSTER] ISPN000078: Starting JGroups channel cluster with stack
<encrypted_stack_name>
```

Infinispan nodes can join the cluster only if they use `SYM_ENCRYPT` and can obtain the secret key from the shared keystore. Otherwise the following message is written to Infinispan logs:

```
[org.jgroups.protocols.SYM_ENCRYPT] <hostname>: received message without encrypt
header from <hostname>; dropping it
```

Reference

The example SYM_ENCRYPT configuration in this procedure shows commonly used parameters. Refer to JGroups documentation for the full set of available parameters.

- JGroups 4 Manual
- JGroups 4.2 Schema

Chapter 8. Setting Up Persistent Storage

Infinispan can persist in-memory data to external storage, giving you additional capabilities to manage your data such as:

Durability

Adding cache stores allows you to persist data to non-volatile storage so it survives restarts.

Write-through caching

Configuring Infinispan as a caching layer in front of persistent storage simplifies data access for applications because Infinispan handles all interactions with the external storage.

Data overflow

Using eviction and passivation techniques ensures that Infinispan keeps only frequently used data in-memory and writes older entries to persistent storage.

8.1. Infinispan Cache Stores

Cache stores connect Infinispan to persistent data sources and implement the `NonBlockingStore` interface.

8.1.1. Configuring Cache Stores

Add cache stores to Infinispan caches in a chain either declaratively or programmatically. Cache read operations check each cache store in the configured order until they locate a valid non-null element of data. Write operations affect all cache stores except for those that you configure as read only.

Procedure

1. Use the `persistence` parameter to configure the persistence layer for caches.
2. Configure whether cache stores are local to the node or shared across the cluster.

   Use either the `shared` attribute declaratively or the `shared(false)` method programmatically.

3. Configure other cache stores properties as appropriate. Custom cache stores can also include `property` parameters.

> Configuring cache stores as shared or not shared (local only) determines which parameters you should set. In some cases, using the wrong combination of parameters in your cache store configuration can lead to data loss or performance issues.
>
> For example, if the cache store is local to a node then it makes sense to fetch state and purge on startup. However, if the cache store is shared, then you should not fetch state or purge on startup.

Local (non-shared) file store

```
<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="true"
           path="${java.io.tmpdir}">
      <write-behind modification-queue-size="123" />
   </file-store>
</persistence>
```

Shared custom cache 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" />

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

Single file store

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

Reference

- Infinispan Configuration Schema
- Infinispan Cache Store Implementations

- Creating Custom Cache Stores

8.1.2. Setting a Global Persistent Location for File-Based Cache Stores

Infinispan uses a global filesystem location for saving data to persistent storage.

> The global persistent location must be unique to each Infinispan instance. To share data between multiple instances, use a shared persistent location.

Infinispan servers use the `$ISPN_HOME/server/data` directory as the global persistent location.

If you are using Infinispan as a library embedded in custom applications and global-state is enabled, the global persistent location defaults to the `user.dir` system property. This system property typically uses the directory where your application starts. You should configure a global persistent location to use a suitable location.

Declarative configuration

```
<cache-container default-cache="myCache">
   <global-state>
      <persistent-location path="example" relative-to="my.data"/>
   </global-state>
   ...
</cache-container>
```

```
new GlobalConfigurationBuilder().globalState().enable().persistentLocation("example",
"my.data");
```

File-Based Cache Stores and Global Persistent Location

When using file-based cache stores, you can optionally specify filesystem directories for storage. Unless absolute paths are declared, directories are always relative to the global persistent location.

For example, you configure your global persistent location as follows:

```
<global-state>
   <persistent-location path="/tmp/example" relative-to="my.data"/>
</global-state>
```

You then configure a Single File cache store that uses a path named `myDataStore` as follows:

```
<file-store path="myDataStore"/>
```

In this case, the configuration results in a Single File cache store in `/tmp/example/myDataStore/myCache.dat`

If you attempt to set an absolute path that resides outside the global persistent location and global-state is enabled, Infinispan throws the following exception:

```
ISPN000558: "The store location 'foo' is not a child of the global persistent location
'bar'"
```

Reference

- Infinispan configuration schema
- org.infinispan.configuration.global.GlobalStateConfiguration

8.1.3. Passivation

Passivation configures Infinispan to write entries to cache stores when it evicts those entries from memory. In this way, passivation ensures that only a single copy of an entry is maintained, either in-memory or in a cache store, which prevents unnecessary and potentially expensive writes to persistent storage.

Activation is the process of restoring entries to memory from the cache store when there is an attempt to access passivated entries. For this reason, when you enable passivation, you must configure cache stores that implement both `CacheWriter` and `CacheLoader` interfaces so they can write and load entries from persistent storage.

When Infinispan evicts an entry from the cache, it notifies cache listeners that the entry is passivated then stores the entry in the cache store. When Infinispan gets an access request for an evicted entry, it lazily loads the entry from the cache store into memory and then notifies cache listeners that the entry is activated.

- Passivation uses the first cache loader in the Infinispan configuration and ignores all others.
- Passivation is not supported with:
  - Transactional stores. Passivation writes and removes entries from the store outside the scope of the actual Infinispan commit boundaries.
  - Shared stores. Shared cache stores require entries to always exist in the store for other owners. For this reason, passivation is not supported because entries cannot be removed.

If you enable passivation with transactional stores or shared stores, Infinispan throws an exception.

Passivation and Cache Stores

Passivation disabled

Writes to data in memory result in writes to persistent storage.

If Infinispan evicts data from memory, then data in persistent storage includes entries that are evicted from memory. In this way persistent storage is a superset of the in-memory cache.

If you do not configure eviction, then data in persistent storage provides a copy of data in memory.

Passivation enabled

Infinispan adds data to persistent storage only when it evicts data from memory.

When Infinispan activates entries, it restores data in memory and deletes data from persistent storage. In this way, data in memory and data in persistent storage form separate subsets of the entire data set, with no intersection between the two.

> Entries in persistent storage can become stale when using shared cache stores. This occurs because Infinispan does not delete passivated entries from shared cache stores when they are activated.
>
> Values are updated in memory but previously passivated entries remain in persistent storage with out of date values.

The following table shows data in memory and in persistent storage after a series of operations:

Operation	Passivation disabled	Passivation enabled	Passivation enabled with shared cache store
Insert k1.	**Memory:** k1 **Disk:** k1	**Memory:** k1 **Disk:** -	**Memory:** k1 **Disk:** -
Insert k2.	**Memory:** k1, k2 **Disk:** k1, k2	**Memory:** k1, k2 **Disk:** -	**Memory:** k1, k2 **Disk:** -
Eviction thread runs and evicts k1.	**Memory:** k2 **Disk:** k1, k2	**Memory:** k2 **Disk:** k1	**Memory:** k2 **Disk:** k1
Read k1.	**Memory:** k1, k2 **Disk:** k1, k2	**Memory:** k1, k2 **Disk:** -	**Memory:** k1, k2 **Disk:** k1
Eviction thread runs and evicts k2.	**Memory:** k1 **Disk:** k1, k2	**Memory:** k1 **Disk:** k2	**Memory:** k1 **Disk:** k1, k2
Remove k2.	**Memory:** k1 **Disk:** k1	**Memory:** k1 **Disk:** -	**Memory:** k1 **Disk:** k1

8.1.4. Cache Loaders and Transactional Caches

Only JDBC String-Based cache stores support transactional operations. If you configure caches as transactional, you should set `transactional=true` to keep data in persistent storage synchronized with data in memory.

For all other cache stores, Infinispan does not enlist cache loaders in transactional operations. This can result in data inconsistency if transactions succeed in modifying data in memory but do not completely apply changes to data in the cache store. In this case manual recovery does not work with cache stores.

Reference

- JDBC String-Based Cache Stores

8.1.5. Segmented Cache Stores

Cache stores can organize data into hash space segments to which keys map.

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.

> The number of segments you configure for cache stores must match the number of segments you define in the Infinispan configuration with the `clustering.hash.numSegments` parameter.
>
> If you change the `numSegments` parameter in the configuration after you add a segmented cache store, Infinispan cannot read data from that cache store.

Reference

Key Ownership

8.1.6. Filesystem-Based Cache Stores

In most cases, filesystem-based cache stores are appropriate for local cache stores for data that overflows from memory because it exceeds size and/or time restrictions.

> You should not use filesystem-based cache stores on shared file systems such as an NFS, Microsoft Windows, or Samba share. Shared file systems do not provide file locking capabilities, which can lead to data corruption.
>
> Likewise, shared file systems are not transactional. If you attempt to use transactional caches with shared file systems, unrecoverable failures can happen when writing to files during the commit phase.

8.1.7. Write-Through

Write-Through is a cache writing mode where writes to memory and writes to cache stores are synchronous. When a client application updates a cache entry, in most cases by invoking `Cache.put()`, Infinispan does not return the call until it updates the cache store. This cache writing mode results in updates to the cache store concluding within the boundaries of the client thread.

The primary advantage of Write-Through mode is that the cache and cache store are updated simultaneously, which ensures that the cache store is always consistent with the cache.

However, Write-Through mode can potentially decrease performance because the need to access and update cache stores directly adds latency to cache operations.

Infinispan defaults to Write-Through mode unless you explicitly configure Write-Behind mode on cache stores.

Write-through configuration

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

Reference

Write-Behind

8.1.8. Write-Behind

Write-Behind is a cache writing mode where writes to memory are synchronous and writes to cache stores are asynchronous.

When clients send write requests, Infinispan adds those operations to a modification queue. Infinispan processes operations as they join the queue so that the calling thread is not blocked and the operation completes immediately.

If the number of write operations in the modification queue increases beyond the size of the queue, Infinispan adds those additional operations to the queue. However, those operations do not complete until Infinispan processes operations that are already in the queue.

For example, calling `Cache.putAsync` returns immediately and the Stage also completes immediately if the modification queue is not full. If the modification queue is full, or if Infinispan is currently processing a batch of write operations, then `Cache.putAsync` returns immediately and the Stage completes later.

Write-Behind mode provides a performance advantage over Write-Through mode because cache operations do not need to wait for updates to the underlying cache store to complete. However, data in the cache store remains inconsistent with data in the cache until the modification queue is processed. For this reason, Write-Behind mode is suitable for cache stores with low latency, such as unshared and local filesystem-based cache stores, where the time between the write to the cache and the write to the cache store is as small as possible.

Write-behind configuration

```
<persistence passivation="false">
   <file-store fetch-state="true"
               read-only="false"
               purge="false" path="${java.io.tmpdir}">
   <write-behind modification-queue-size="123"
                 fail-silently="true"/>
   </file-store>
</persistence>
```

The preceding configuration example uses the `fail-silently` parameter to control what happens when either the cache store is unavailable or the modification queue is full.

- If `fail-silently="true"` then Infinispan logs WARN messages and rejects write operations.
- If `fail-silently="false"` then Infinispan throws exceptions if it detects the cache store is unavailable during a write operation. Likewise if the modification queue becomes full, Infinispan throws an exception.

  In some cases, data loss can occur if Infinispan restarts and write operations exist in the modification queue. For example the cache store goes offline but, during the time it takes to detect that the cache store is unavailable, write operations are added to the modification queue because it is not full. If Infinispan restarts or otherwise becomes unavailable before the cache store comes back online, then the write operations in the modification queue are lost because they were not persisted.

Reference

Write-Through

8.2. Cache Store Implementations

Infinispan provides several cache store implementations that you can use. Alternatively you can provide custom cache stores.

8.2.1. Cluster Cache Loaders

`ClusterCacheLoader` retrieves data from other Infinispan cluster members but does not persist data. In other words, `ClusterCacheLoader` is not a cache store.

`ClusterCacheLoader` provides a non-blocking partial alternative to state transfer. `ClusterCacheLoader` fetches keys from other nodes on demand if those keys are not available on the local node, which is similar to lazily loading cache content.

The following points also apply to `ClusterCacheLoader`:

- Preloading does not take effect (`preload=true`).
- Fetching persistent state is not supported (`fetch-state=true`).

- Segmentation is not supported.

> **Warning:** The `ClusterLoader` has been deprecated and will be removed in a future release.

Declarative configuration

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

Programmatic configuration

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

Reference

- Infinispan configuration schema
- ClusterLoader
- ClusterLoaderConfiguration

8.2.2. Single File Cache Stores

Single File cache stores, `SingleFileStore`, persist data to file. Infinispan also maintains an in-memory index of keys while keys and values are stored in the file. By default, Single File cache stores are segmented, which means that Infinispan creates a separate file for each segment.

Because `SingleFileStore` keeps an in-memory index of keys and the location of values, it requires additional memory, depending on the key size and the number of keys. For this reason, `SingleFileStore` is not recommended for use cases where the keys have a large size.

In some cases, `SingleFileStore` can also become fragmented. If the size of values continually increases, available space in the single file is not used but the entry is appended to the end of the file. Available space in the file is used only if an entry can fit within it. Likewise, if you remove all entries from memory, the single file store does not decrease in size or become defragmented.

Declarative configuration

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

Programmatic configuration

- For embedded deployments, do the following:

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
    .addSingleFileStore()
    .maxEntries(5000);
```

- For server deployments, do the following:

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.configuration.NearCacheMode;
...

ConfigurationBuilder builder = new ConfigurationBuilder();
builder
  .remoteCache("mycache")
    .configuration("<infinispan><cache-container><distributed-cache name=\"mycache
\"><persistence><file-store max-entries=\"5000\"/></persistence></distributed-
cache></cache-container></infinispan>");
```

Segmentation

Single File cache stores support segmentation and create a separate instance per segment, which results in multiple directories in the path you configure. Each directory is a number that represents the segment to which the data maps.

Reference

- Setting a Global Persistent Location
- Infinispan configuration schema
- SingleFileStore

8.2.3. JDBC String-Based Cache Stores

JDBC String-Based cache stores, `JdbcStringBasedStore`, use JDBC drivers to load and store values in the underlying database.

`JdbcStringBasedStore` stores each entry in its own row in the table to increase throughput for concurrent loads. `JdbcStringBasedStore` also uses a simple one-to-one mapping that maps each key to a `String` object using the `key-to-string-mapper` interface.

Infinispan provides a default implementation, `DefaultTwoWayKey2StringMapper`, that handles primitive types.

> By default Infinispan shares are not stored, which means that all nodes in the cluster write to the underlying store on each update. If you want operations to write to the underlying database once only, you must configure JDBC store as shared.

Segmentation

`JdbcStringBasedStore` uses segmentation by default and requires a column in the database table to represent the segments to which entries belong.

Connection Factories

`JdbcStringBasedStore` relies on a `ConnectionFactory` implementation to connection to a database.

Infinispan provides the following `ConnectionFactory` implementations:

`PooledConnectionFactoryConfigurationBuilder`
A connection factory based on Agroal that you configure via `PooledConnectionFactoryConfiguration`.

Alternatively, you can specify configuration properties prefixed with `org.infinispan.agroal.` as in the following example:

```
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
```

You then configure Infinispan to use your properties file via `PooledConnectionFactoryConfiguration.propertyFile`.

> You should use `PooledConnectionFactory` with standalone deployments, rather than deployments in servlet containers.

`ManagedConnectionFactoryConfigurationBuilder`
A connection factory that you can can use with managed environments such as application servers. This connection factory can explore a configurable location in the JNDI tree and delegate connection management to the `DataSource`.

`SimpleConnectionFactoryConfigurationBuilder`
A connection factory that creates database connections on a per invocation basis. You should use this connection factory for test or development environments only.

Reference

- Agroal
- ConnectionFactoryConfigurationBuilder
- PooledConnectionFactoryConfigurationBuilder
- ManagedConnectionFactoryConfigurationBuilder
- SimpleConnectionFactoryConfigurationBuilder

JDBC String-Based Cache Store Configuration

You can configure `JdbcStringBasedStore` programmatically or declaratively.

Declarative configuration

- Using `PooledConnectionFactory`

```
<persistence>
   <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:11.0" shared="
true">
      <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"
                          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" />
         <segment-column name="SEGMENT_COLUMN" type="INT" />
      </string-keyed-table>
   </string-keyed-jdbc-store>
</persistence>
```

- Using `ManagedConnectionFactory`

```
<persistence>
  <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:11.0" shared="true
">
    <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" />
        <segment-column name="SEGMENT_COLUMN" type="INT"/>
    </string-keyed-table>
  </string-keyed-jdbc-store>
</persistence>
```

Programmatic configuration

- Using `PooledConnectionFactory`

```
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")
         .segmentColumnName("SEGMENT_COLUMN").segmentColumnType("INT")
      .connectionPool()
         .connectionUrl("jdbc:h2:mem:infinispan_string_based;DB_CLOSE_DELAY=-1")
         .username("sa")
         .driverClass("org.h2.Driver");
```

- Using `ManagedConnectionFactory`

```
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")
         .segmentColumnName("SEGMENT_COLUMN").segmentColumnType("INT")
      .dataSource()
         .jndiUrl("java:/StringStoreWithManagedConnectionTest/DS");
```

Reference

- JDBC cache store configuration schema
- JdbcStringBasedStore
- JdbcStringBasedStoreConfiguration

8.2.4. JPA Cache Stores

JPA (Java Persistence API) cache stores, `JpaStore`, use formal schema to persist data. Other applications can then read from persistent storage to load data from Infinispan. However, other applications should not use persistent storage concurrently with Infinispan.

When using `JpaStore`, you should take the following into consideration:

- Keys should be the ID of the entity. Values should be the entity object.
- Only a single `@Id` or `@EmbeddedId` annotation is allowed.
- Auto-generated IDs with the `@GeneratedValue` annotation are not supported.
- All entries are stored as immortal.
- `JpaStore` does not support segmentation.

Declarative configuration

```
<local-cache name="vehicleCache">
   <persistence passivation="false">
      <jpa-store xmlns="urn:infinispan:config:store:jpa:11.0"
         persistence-unit="org.infinispan.persistence.jpa.configurationTest"
         entity-class="org.infinispan.persistence.jpa.entity.Vehicle">
        />
   </persistence>
</local-cache>
```

Parameter	Description
`persistence-unit`	Specifies the JPA persistence unit name in the JPA configuration file, `persistence.xml`, that contains the JPA entity class.
`entity-class`	Specifies the fully qualified JPA entity class name that is expected to be stored in this cache. Only one class is allowed.

Programmatic configuration

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

Parameter	Description
`persistenceUnitName`	Specifies the JPA persistence unit name in the JPA configuration file, `persistence.xml`, that contains the JPA entity class.

Parameter	Description
entityClass	Specifies the fully qualified JPA entity class name that is expected to be stored in this cache. Only one class is allowed.

Reference

- JPA cache store configuration schema
- JpaStore
- JpaStoreConfiguration
- JPA Cache Store test
- JPA Cache Store test configuration

JPA Cache Store Usage Example

This section provides an example for using JPA cache stores.

Prerequistes

- Configure Infinispan to marshall your JPA entities. By default, Infinispan uses ProtoStream for marshalling Java objects. To marshall JPA entities, you must create a `SerializationContextInitializer` implementation that registers a `.proto` schema and marshaller with a `SerializationContext`.

Procedure

1. Define a persistence unit "myPersistenceUnit" in `persistence.xml`.

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

2. Create a user entity class.

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

    ...
}
```

3. Configure a cache named "usersCache" with a JPA cache store.

   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.

```java
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(...));
```

- Caches that use a JPA cache store can store one type of data only, as in the following example:

```java
Cache<String, User> usersCache = cacheManager.getCache("myJPACache");
// Cache is configured for the User entity class
usersCache.put("username", new User());
// Cannot configure caches to use another entity class with JPA cache stores
Cache<Integer, Teacher> teachersCache = cacheManager.getCache("myJPACache");
teachersCache.put(1, new Teacher());
// The put request does not work for the Teacher entity class
```

- The `@EmbeddedId` annotation allows you to use composite keys, as in the following example:

```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;
    ...
}
```

References

- Using the ProtoStream Marshaller
- Marshalling Custom Java Objects with ProtoStream

8.2.5. Remote Cache Stores

Remote cache stores, `RemoteStore`, use the Hot Rod protocol to store data on Infinispan clusters.

The following is an example `RemoteStore` configuration that stores data in a cache named "mycache" on two Infinispan Server instances, named "one" and "two":

> If you configure remote cache stores as shared you cannot preload data. In other words if `shared="true"` in your configuration then you must set `preload="false"`.

Declarative configuration

```
<persistence>
   <remote-store xmlns="urn:infinispan:config:store:remote:11.0" 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>
```

Programmatic configuration

```
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();
```

Segmentation

`RemoteStore` supports segmentation and can publish keys and entries by segment, which makes bulk operations more efficient. However, segmentation is available only with Infinispan Hot Rod protocol version 2.3 or later.

> ⚠ When you enable segmentation for `RemoteStore`, it uses the number of segments that you define in your Infinispan server configuration.
>
> If the source cache is segmented and uses a different number of segments than `RemoteStore`, then incorrect values are returned for bulk operations. In this case, you should disable segmentation for `RemoteStore`.

Reference

- Remote cache store configuration schema
- RemoteStore
- RemoteStoreConfigurationBuilder

8.2.6. RocksDB Cache Stores

RocksDB provides key-value filesystem-based storage with high performance and reliability for highly concurrent environments.

RocksDB cache stores, `RocksDBStore`, use two databases. One database provides a primary cache store for data in memory; the other database holds entries that Infinispan expires from memory.

Declarative configuration

```
<local-cache name="vehicleCache">
   <persistence>
      <rocksdb-store xmlns="urn:infinispan:config:store:rocksdb:11.0" path=
"rocksdb/data">
         <expiration path="rocksdb/expired"/>
      </rocksdb-store>
   </persistence>
</local-cache>
```

Programmatic configuration

```
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(...));
```

```java
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("rocksdb/data")
                .expiredLocation("rocksdb/expired")
        .properties(props)
                .build();
```

Table 1. `RocksDBStore` configuration parameters

Parameter	Description
`location`	Specifies the path to the RocksDB database that provides the primary cache store. If you do not set the location, it is automatically created. Note that the path must be relative to the global persistent location.
`expiredLocation`	Specifies the path to the RocksDB database that provides the cache store for expired data. If you do not set the location, it is automatically created. Note that the path must be relative to the global persistent location.
`expiryQueueSize`	Sets the size of the in-memory queue for expiring entries. When the queue reaches the size, Infinispan flushes the expired into the RocksDB cache store.
`clearThreshold`	Sets the maximum number of entries before deleting and re-initializing (**re-init**) the RocksDB database. For smaller size cache stores, iterating through all entries and removing each one individually can provide a faster method.

RocksDB tuning parameters

You can also specify the following RocksDB tuning parameters:

- `compressionType`
- `blockSize`
- `cacheSize`

RocksDB configuration properties

Optionally set properties in the configuration as follows:

- Prefix properties with `database` to adjust and tune RocksDB databases.
- Prefix properties with `data` to configure the column families in which RocksDB stores your data.

```
<property name="database.max_background_compactions">2</property>
<property name="data.write_buffer_size">64MB</property>
<property
name="data.compression_per_level">kNoCompression:kNoCompression:kNoCompression:kSnappy
Compression:kZSTD:kZSTD</property>
```

Segmentation

`RocksDBStore` supports segmentation and creates a separate column family per segment. Segmented RocksDB cache stores improve lookup performance and iteration but slightly lower performance of write operations.

> 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 cache store start time.

Reference

- RocksDB cache store configuration schema
- RocksDBStore
- RocksDBStoreConfiguration
- rocksdb.org
- RocksDB Tuning Guide
- RocksDB Cache Store test
- RocksDB Cache Store test configuration

8.2.7. Soft-Index File Stores

Soft-Index File cache stores, `SoftIndexFileStore`, provide local file-based storage.

`SoftIndexFileStore` is a Java implementation that uses a variant of **B+ Tree** that is cached in-memory using Java soft references. The **B+ Tree**, called `Index` is offloaded on the file system to a single file that is purged and rebuilt each time the cache store restarts.

`SoftIndexFileStore` stores data in a set of files rather than a single file. When usage of any file drops below 50%, the entries in the file are overwritten to another file and the file is then deleted.

`SoftIndexFileStore` persists data in a set of files that are written in an append-only method. For this reason, if you use `SoftIndexFileStore` on conventional magnetic disk, it does not need to seek when writing a burst of entries.

Most structures in `SoftIndexFileStore` are bounded, so out-of-memory exceptions do not pose a risk. You can also configure limits for concurrently open files.

By default the size of a node in the `Index` is limited to 4096 bytes. This size also limits the key length; more precisely the length of serialized keys. For this reason, you cannot use keys longer than the size of the node, 15 bytes. Additionally, key length is stored as "short", which limits key length to

32767 bytes. `SoftIndexFileStore` throws an exception if keys are longer after serialization occurs.

`SoftIndexFileStore` cannot detect expired entries, which can lead to excessive usage of space on the file system .

> `AdvancedStore.purgeExpired()` is not implemented in `SoftIndexFileStore`.

Declarative configuration

```
<persistence>
    <soft-index-file-store xmlns="urn:infinispan:config:store:soft-index:11.0">
        <index path="testCache/index" />
        <data path="testCache/data" />
    </soft-index-file-store>
</persistence>
```

Programmatic configuration

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

Segmentation

Soft-Index File cache stores support segmentation and create a separate instance per segment, which results in multiple directories in the path you configure. Each directory is a number that represents the segment to which the data maps.

Reference

- Soft-Index File cache store configuration schema
- SoftIndexFileStore
- SoftIndexFileStoreConfiguration

8.2.8. Implementing Custom Cache Stores

You can create custom cache stores through the Infinispan persistent SPI.

Infinispan Persistence SPI

The Infinispan Service Provider Interface (SPI) enables read and write operations to external storage through the `NonBlockingStore` interface and has the following features:

Portability across JCache-compliant vendors

Infinispan maintains compatibility between the `NonBlockingStore` interface and the `JSR-107` JCache specification by using an adapter that handles blocking code.

Simplified transaction integration

Infinispan automatically handles locking so your implementations do not need to coordinate concurrent access to persistent stores. Depending on the locking mode you use, concurrent writes to the same key generally do not occur. However, you should expect operations on the persistent storage to originate from multiple threads and create implementations to tolerate this behavior.

Parallel iteration

Infinispan lets you iterate over entries in persistent stores with multiple threads in parallel.

Reduced serialization resulting in less CPU usage

Infinispan exposes stored entries in a serialized format that can be transmitted remotely. For this reason, Infinispan does not need to deserialize entries that it retrieves from persistent storage and then serialize again when writing to the wire.

Reference

- Persistence SPI
- NonBlockingStore
- JSR-107

Creating Cache Stores

Create custom cache stores by implementing the `NonBlockingStore` interface.

1. Implement the appropriate Infinispan persistent SPIs.
2. Annotate your store class with the `@ConfiguredBy` annotation if it has a custom configuration.
3. Create a custom cache store configuration and builder if desired.
   a. Extend `AbstractStoreConfiguration` and `AbstractStoreConfigurationBuilder`.
   b. Optionally add the following annotations to your store Configuration class to ensure that your custom configuration builder parses your cache store configuration from XML:
      - `@ConfigurationFor`
      - `@BuiltBy`

      If you do not add these annotations, then `CustomStoreConfigurationBuilder` parses the common store attributes defined in `AbstractStoreConfiguration` and any additional elements are ignored.

      > If a configuration does not declare the `@ConfigurationFor` annotation, a warning message is logged when Infinispan initializes the cache.

Configuring Infinispan to Use Custom Stores

After you create your custom cache store implementation, configure Infinispan to use it.

Declarative configuration

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

Programmatic configuration

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

Deploying Custom Cache Stores

You can package custom cache stores into JAR files and deploy them to Infinispan servers as follows:

1. Package your custom cache store implementation in a JAR file.
2. Add your JAR file to the `server/lib` directory of your Infinispan server.

8.3. Migrating Between Cache Stores

Infinispan provides a utility to migrate data from one cache store to another.

8.3.1. Cache Store Migrator

Infinispan provides the `StoreMigrator.java` utility that recreates data for the latest Infinispan cache store implementations.

`StoreMigrator` takes a cache store from a previous version of Infinispan as source and uses a cache store implementation as target.

When you run `StoreMigrator`, it creates the target cache with the cache store type that you define using the `EmbeddedCacheManager` interface. `StoreMigrator` then loads entries from the source store into memory and then puts them into the target cache.

`StoreMigrator` also lets you migrate data from one type of cache store to another. For example, you can migrate from a JDBC String-Based cache store to a Single File cache store.

> `StoreMigrator` cannot migrate data from segmented cache stores to:
>
> - Non-segmented cache store.
> - Segmented cache stores that have a different number of segments.

8.3.2. Getting the Store Migrator

`StoreMigrator` is available as part of the Infinispan tools library, `infinispan-tools`, and is included in the Maven repository.

Procedure

- Configure your `pom.xml` for `StoreMigrator` as follows:

```xml
<?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>
      </dependency>
      <!-- Additional dependencies -->
    </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>
org.infinispan.tools.store.migrator.StoreMigrator</mainClass>
            <arguments>
              <argument>path/to/migrator.properties</argument>
            </arguments>
          </configuration>
        </plugin>
      </plugins>
    </build>
</project>
```

8.3.3. Configuring the Store Migrator

Set properties for source and target cache stores in a `migrator.properties` file.

Procedure

1. Create a `migrator.properties` file.
2. Configure the source cache store in `migrator.properties`.
   a. Prepend all configuration properties with `source.` as in the following example:

```
source.type=SOFT_INDEX_FILE_STORE
source.cache_name=myCache
source.location=/path/to/source/sifs
```

3. Configure the target cache store in `migrator.properties`.
   a. Prepend all configuration properties with `target.` as in the following example:

```
target.type=SINGLE_FILE_STORE
target.cache_name=myCache
target.location=/path/to/target/sfs.dat
```

Store Migrator Properties

Configure source and target cache stores in a `StoreMigrator` properties.

Table 2. Cache Store Type Property

Property	Description	Required/Optional

`type`	Specifies the type of cache store type for a source or target.
	
	`.type=JDBC_STRING`
	
	`.type=JDBC_BINARY`
	
	`.type=JDBC_MIXED`
	
	`.type=LEVELDB`
	
	`.type=ROCKSDB`
	
	`.type=SINGLE_FILE_STORE`
	
	`.type=SOFT_INDEX_FILE_STORE`
	
	`.type=JDBC_MIXED`
		Required

Table 3. Common Properties

Property	Description	Example Value	Required/Optional
`cache_name`	Names the cache that the store backs.	`.cache_name=myCache`	Required
`segment_count`	Specifies the number of segments for target cache stores that can use segmentation. The number of segments must match `clustering.hash.numSegments` in the Infinispan configuration. In other words, the number of segments for a cache store must match the number of segments for the corresponding cache. If the number of segments is not the same, Infinispan cannot read data from the cache store.	`.segment_count=256`	Optional

Table 4. JDBC Properties

Property	Description	Required/Optional
`dialect`	Specifies the dialect of the underlying database.	Required
`version`	Specifies the marshaller version for source cache stores. Set one of the following values: * `8` for Infinispan 8.x * `9` for Infinispan 9.x * `10` Infinispan 10.x	Required for source stores only. For example: `source.version=9`
`marshaller.class`	Specifies a custom marshaller class.	Required if using custom marshallers.

<table>
<tr><th>Property</th><th>Description</th><th>Required/Optional</th></tr>
<tr><td>marshaller.externalizers</td><td>Specifies a comma-separated list of custom AdvancedExternalizer implementations to load in this format: [id]:<Externalizer class></td><td>Optional</td></tr>
<tr><td>connection_pool.connection_url</td><td>Specifies the JDBC connection URL.</td><td>Required</td></tr>
<tr><td>connection_pool.driver_class</td><td>Specifies the class of the JDBC driver.</td><td>Required</td></tr>
<tr><td>connection_pool.username</td><td>Specifies a database username.</td><td>Required</td></tr>
<tr><td>connection_pool.password</td><td>Specifies a password for the database username.</td><td>Required</td></tr>
<tr><td>db.major_version</td><td>Sets the database major version.</td><td>Optional</td></tr>
<tr><td>db.minor_version</td><td>Sets the database minor version.</td><td>Optional</td></tr>
<tr><td>db.disable_upsert</td><td>Disables database upsert.</td><td>Optional</td></tr>
<tr><td>db.disable_indexing</td><td>Specifies if table indexes are created.</td><td>Optional</td></tr>
<tr><td>table.string.table_name_prefix</td><td>Specifies additional prefixes for the table name.</td><td>Optional</td></tr>
<tr><td>table.string.<id|data|timestamp>.name</td><td>Specifies the column name.</td><td>Required</td></tr>
<tr><td>table.string.<id|data|timestamp>.type</td><td>Specifies the column type.</td><td>Required</td></tr>
<tr><td>key_to_string_mapper</td><td>Specifies the TwoWayKey2StringMapper class.</td><td>Optional</td></tr>
</table>

To migrate from Binary cache stores in older Infinispan versions, change `table.string.*` to `table.binary.\*` in the following properties:

- `source.table.binary.table_name_prefix`
- `source.table.binary.<id\|data\|timestamp>.name`
- `source.table.binary.<id\|data\|timestamp>.type`

```
# Example configuration for migrating to a JDBC String-Based cache store
target.type=STRING
target.cache_name=myCache
target.dialect=POSTGRES
target.marshaller.class=org.example.CustomMarshaller
target.marshaller.externalizers=25:Externalizer1,org.example.Externalizer2
target.connection_pool.connection_url=jdbc:postgresql:postgres
target.connection_pool.driver_class=org.postrgesql.Driver
target.connection_pool.username=postgres
target.connection_pool.password=redhat
target.db.major_version=9
target.db.minor_version=5
target.db.disable_upsert=false
target.db.disable_indexing=false
target.table.string.table_name_prefix=tablePrefix
target.table.string.id.name=id_column
target.table.string.data.name=datum_column
target.table.string.timestamp.name=timestamp_column
target.table.string.id.type=VARCHAR
target.table.string.data.type=bytea
target.table.string.timestamp.type=BIGINT
target.key_to_string_mapper=org.infinispan.persistence.keymappers.
DefaultTwoWayKey2StringMapper
```

Table 5. RocksDB Properties

Property	Description	Required/Optional
`location`	Sets the database directory.	Required
`compression`	Specifies the compression type to use.	Optional

```
# Example configuration for migrating from a RocksDB cache store.
source.type=ROCKSDB
source.cache_name=myCache
source.location=/path/to/rocksdb/database
source.compression=SNAPPY
```

Table 6. SingleFileStore Properties

Property	Description	Required/Optional
`location`	Sets the directory that contains the cache store `.dat` file.	Required

```
# Example configuration for migrating to a Single File cache store.
target.type=SINGLE_FILE_STORE
target.cache_name=myCache
target.location=/path/to/sfs.dat
```

Table 7. SoftIndexFileStore Properties

Property	Description	Value
Required/Optional	`location`	Sets the database directory.
Required	`index_location`	Sets the database index directory.

```
# Example configuration for migrating to a Soft-Index File cache store.
target.type=SOFT_INDEX_FILE_STORE
target.cache_name=myCache
target.location=path/to/sifs/database
target.location=path/to/sifs/index
```

8.3.4. Migrating Cache Stores

Run `StoreMigrator` to migrate data from one cache store to another.

Prerequisites

- Get `infinispan-tools.jar`.
- Create a `migrator.properties` file that configures the source and target cache stores.

Procedure

- If you build `infinispan-tools.jar` from source, do the following:
  1. Add `infinispan-tools.jar` and dependencies for your source and target databases, such as JDBC drivers, to your classpath.
  2. Specify `migrator.properties` file as an argument for `StoreMigrator`.
- If you pull `infinispan-tools.jar` from the Maven repository, run the following command:

  `mvn exec:java`

Chapter 9. Setting Up Partition Handling

9.1. 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

Strategy	Description	CAP
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
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.

9.1.1. 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.

9.1.2. 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.

9.1.3. 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

9.1.4. 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.

```
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, CacheEntry<Integer, String>>> conflicts = crm.getConflicts();
conflicts.forEach(map -> {
   CacheEntry<Integer, String> 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.

9.1.5. 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.

```
<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:

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

Implement a custom merge policy

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

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

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

```java
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.

9.1.6. 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);
}
```