Configuring Infinispan 10.1

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 cluser, 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

  ```
  CacheManager 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.

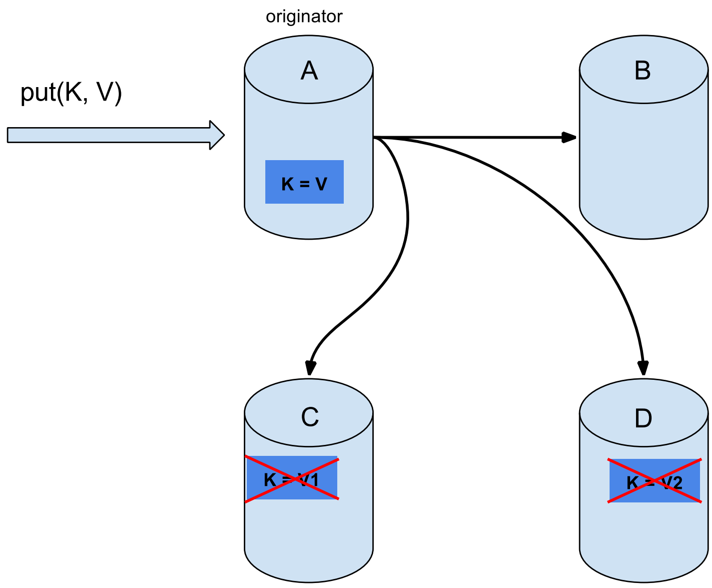

Figure 1. Invalidation mode

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.

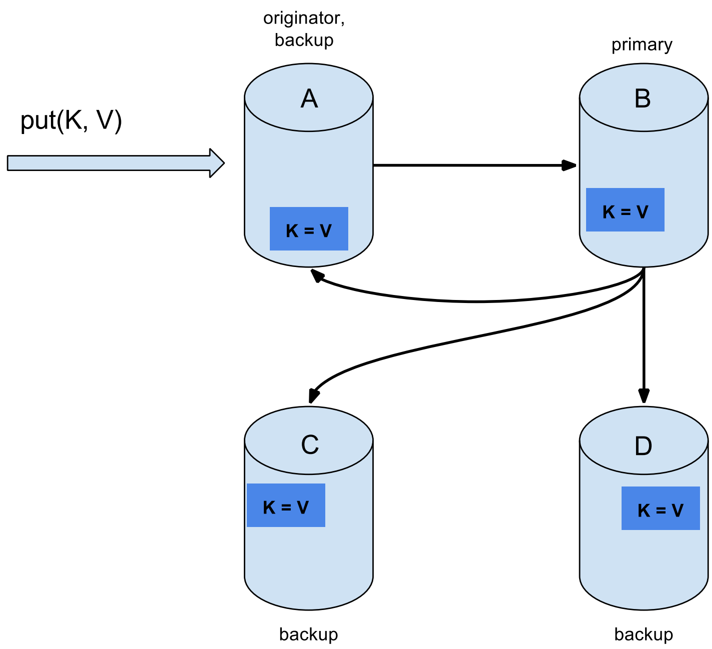

Figure 2. Replicated mode

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.

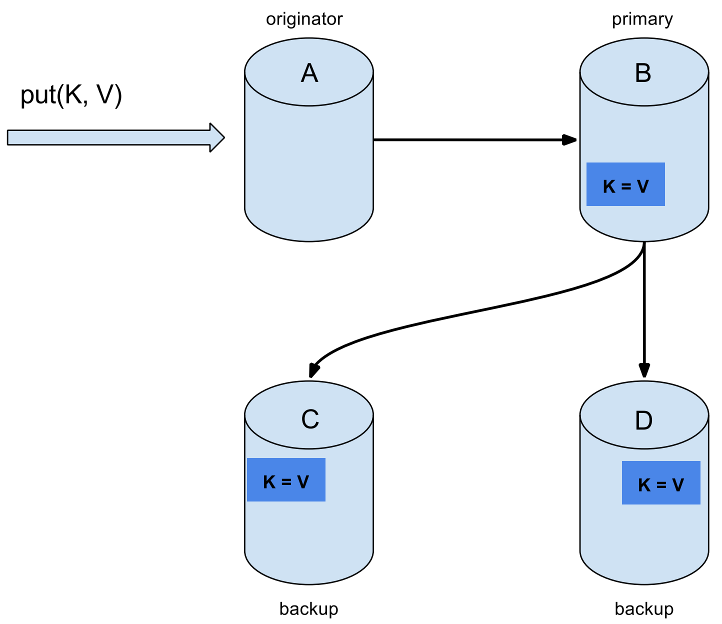

Figure 3. Distributed mode

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 are another way to customize the mapping of segments to nodes. The nodes in a cluster are not always identical. If a node has 2x the memory of a "regular" node, configuring it with a `capacityFactor` of `2` tells Infinispan to allocate 2x segments to that node. The capacity factor can be any non-negative number, and the hashing algorithm will try to assign to each node a load weighted by its capacity factor (both as a primary owner and as a backup owner).

One interesting use case is nodes with a capacity factor of `0`. This could be useful when some nodes are too short-lived to be useful as data owners, but they can't use HotRod (or other remote protocols) because they need transactions. With cross-site replication as well, the "site master" should only deal with forwarding commands between sites and shouldn't handle user requests, so it makes sense to configure it with a capacity factor of `0`.

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)
   .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.

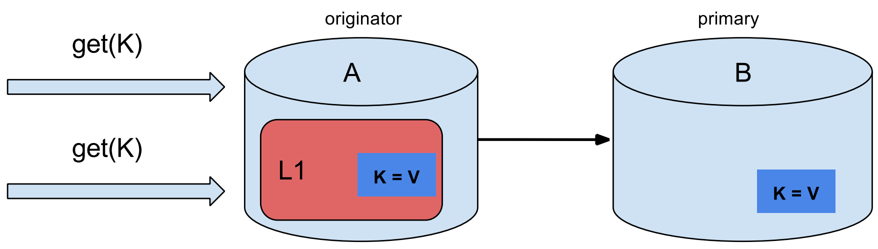

Figure 4. L1 caching

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.

```java
// 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:

```java
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:

```java
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.

> Prior to version 9.0, the asynchronous API was emulated by borrowing a thread from an internal thread pool and running a blocking operation on that thread.

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 Caches Declaratively

Infinispan declarative configuration.

4.1. Infinispan subsystem

The Infinispan subsystem configures the cache containers and caches.

The subsystem declaration is enclosed in the following XML element:

```
<subsystem xmlns="urn:infinispan:server:core:10.1" default-cache-container="clustered
">
  ...
</subsystem>
```

4.1.1. Containers

The Infinispan subsystem can declare multiple containers. A container is declared as follows:

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

> Infinispan does not provide an implicit default cache, but lets you name a cache as the default.

If you need to declare clustered caches (distributed, replicated, invalidation), you also need to specify the `<transport/>` element which references an existing JGroups transport. This is not needed if you only intend to have local caches only.

```
<transport executor="infinispan-transport" lock-timeout="60000" stack="udp" cluster=
"my-cluster-name"/>
```

4.1.2. Cache declarations

Now you can declare your caches. Please be aware that only the caches declared in the configuration will be available to the endpoints and that attempting to access an undefined cache is an illegal operation. Contrast this with the default Infinispan library behaviour where obtaining an undefined cache will implicitly create one using the default settings. The following are example declarations for all four available types of caches:

```xml
<local-cache name="default" start="EAGER">
  ...
</local-cache>

<replicated-cache name="replcache" mode="SYNC" remote-timeout="30000" start="EAGER">
  ...
</replicated-cache>

<invalidation-cache name="invcache" mode="SYNC" remote-timeout="30000" start="EAGER">
  ...
</invalidation-cache>

<distributed-cache name="distcache" mode="SYNC" segments="20" owners="2" remote-
timeout="30000" start="EAGER">
  ...
</distributed-cache>
```

4.2. Locking

To define the locking configuration for a cache, add the `<locking/>` element as follows:

```xml
<locking isolation="REPEATABLE_READ" acquire-timeout="30000" concurrency-level="1000"
striping="false"/>
```

The possible attributes for the locking element are:

- *isolation* sets the cache locking isolation level. Can be NONE, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE. Defaults to REPEATABLE_READ
- *striping* if true, a pool of shared locks is maintained for all entries that need to be locked. Otherwise, a lock is created per entry in the cache. Lock striping helps control memory footprint but may reduce concurrency in the system.
- *acquire-timeout* maximum time to attempt a particular lock acquisition.
- *concurrency-level* concurrency level for lock containers. Adjust this value according to the number of concurrent threads interacting with Infinispan.
- *concurrent-updates* for non-transactional caches only: if set to true(default value) the cache keeps data consistent in the case of concurrent updates. For clustered caches this comes at the cost of an additional RPC, so if you don't expect your application to write data concurrently, disabling this flag increases performance.

4.3. Loaders and Stores

Loaders and stores can be defined in server mode in almost the same way as in embedded mode.

However, in server mode it is no longer necessary to define the `<persistence>…</persistence>` tag. Instead, a store's attributes are now defined on the store type element. For example, to configure the H2 database with a distributed cache in domain mode we define the "default" cache as follows in our domain.xml configuration:

```
<subsystem xmlns="urn:infinispan:server:core:10.1">
  <cache-container name="clustered" default-cache="default" statistics="true">
    <transport lock-timeout="60000"/>
    <global-state/>
    <distributed-cache name="default">
      <string-keyed-jdbc-store datasource="java:jboss/datasources/ExampleDS" fetch-
state="true" shared="true">
        <string-keyed-table prefix="ISPN">
          <id-column name="id" type="VARCHAR"/>
          <data-column name="datum" type="BINARY"/>
          <timestamp-column name="version" type="BIGINT"/>
        </string-keyed-table>
        <write-behind modification-queue-size="20"/>
      </string-keyed-jdbc-store>
    </distributed-cache>
  </cache-container>
</subsystem>
```

Another important thing to note in this example, is that we use the "ExampleDS" datasource which is defined in the datasources subsystem in our domain.xml configuration as follows:

```
<subsystem xmlns="urn:jboss:domain:datasources:4.0">
  <datasources>
    <datasource jndi-name="java:jboss/datasources/ExampleDS" pool-name="ExampleDS"
enabled="true" use-java-context="true">
      <connection-url>jdbc:h2:mem:test;DB_CLOSE_DELAY=-
1;DB_CLOSE_ON_EXIT=FALSE</connection-url>
      <driver>h2</driver>
      <security>
        <user-name>sa</user-name>
        <password>sa</password>
      </security>
    </datasource>
  </datasources>
</subsystem>
```

For additional examples of store configurations, please view the configuration templates in the default "domain.xml" file provided with in the server distribution at `./domain/configuration/domain.xml`.

4.4. State Transfer

To define the state transfer configuration for a distributed or replicated cache, add the `<state-transfer/>` element as follows:

```
<state-transfer enabled="true" timeout="240000" chunk-size="512" await-initial-
transfer="true" />
```

The possible attributes for the state-transfer element are:

- *enabled* if true, this will cause the cache to ask neighboring caches for state when it starts up, so the cache starts 'warm', although it will impact startup time. Defaults to true.
- *timeout* the maximum amount of time (ms) to wait for state from neighboring caches, before throwing an exception and aborting startup. Defaults to 240000 (4 minutes).
- *chunk-size* the number of cache entries to batch in each transfer. Defaults to 512.
- *await-initial-transfer* if true, this will cause the cache to wait for initial state transfer to complete before responding to requests. Defaults to true.

4.5. Declarative Cache Configuration

Declarative configuration comes in a form of XML document that adheres to a provided Infinispan configuration XML schema.

Every aspect of Infinispan that can be configured declaratively can also be configured programmatically. In fact, declarative configuration, behind the scenes, invokes the programmatic configuration API as the XML configuration file is being processed. One can even use a combination of these approaches. For example, you can read static XML configuration files and at runtime programmatically tune that same configuration. Or you can use a certain static configuration defined in XML as a starting point or template for defining additional configurations in runtime.

There are two main configuration abstractions in Infinispan: `global` and `cache`.

Global configuration

Global configuration defines global settings shared among all cache instances created by a single EmbeddedCacheManager. Shared resources like thread pools, serialization/marshalling settings, transport and network settings, JMX domains are all part of global configuration.

Cache configuration

Cache configuration is specific to the actual caching domain itself: it specifies eviction, locking, transaction, clustering, persistence etc. You can specify as many named cache configurations as you need. One of these caches can be indicated as the `default` cache, which is the cache returned by the `CacheManager.getCache()` API, whereas other named caches are retrieved via the `CacheManager.getCache(String name)` API.

Whenever they are specified, named caches inherit settings from the default cache while additional behavior can be specified or overridden. Infinispan also provides a very flexible inheritance mechanism, where you can define a hierarchy of configuration templates, allowing multiple caches to share the same settings, or overriding specific parameters as necessary.

One of the major goals of Infinispan is to aim for zero configuration. A simple XML configuration file containing nothing more than a single infinispan element is enough to get you started. The configuration file listed below provides sensible defaults and is perfectly valid.

infinispan.xml

```
<infinispan />
```

However, that would only give you the most basic, local mode, non-clustered cache manager with no caches. Non-basic configurations are very likely to use customized global and default cache elements.

Declarative configuration is the most common approach to configuring Infinispan cache instances. In order to read XML configuration files one would typically construct an instance of DefaultCacheManager by pointing to an XML file containing Infinispan configuration. Once the configuration file is read you can obtain reference to the default cache instance.

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-file.xml");
Cache defaultCache = manager.getCache();
```

or any other named instance specified in `my-config-file.xml`.

```
Cache someNamedCache = manager.getCache("someNamedCache");
```

The name of the default cache is defined in the `<cache-container>` element of the XML configuration file, and additional caches can be configured using the `<local-cache>`,`<distributed-cache>` ,`<invalidation-cache>` or `<replicated-cache>` elements.

The following example shows the simplest possible configuration for each of the cache types supported by Infinispan:

```
<infinispan>
   <cache-container default-cache="local">
      <transport cluster="mycluster"/>
      <local-cache name="local"/>
      <invalidation-cache name="invalidation" mode="SYNC"/>
      <replicated-cache name="repl-sync" mode="SYNC"/>
      <distributed-cache name="dist-sync" mode="SYNC"/>
   </cache-container>
</infinispan>
```

4.6. Cache configuration templates

As mentioned above, Infinispan supports the notion of *configuration templates*. These are full or partial configuration declarations which can be shared among multiple caches or as the basis for more complex configurations.

The following example shows how a configuration named `local-template` is used to define a cache named `local`.

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

      <!-- cache definitions -->
      <local-cache name="local" configuration="local-template" />
   </cache-container>
</infinispan>
```

Templates can inherit from previously defined templates, augmenting and/or overriding some or all of the configuration elements:

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

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

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

In the above example, `base-template` defines a local cache with a specific *expiration* configuration. The `extended-template` configuration inherits from `base-template`, overriding just a single parameter of the *expiration* element (all other attributes are inherited) and adds a *memory* element. Finally, two caches are defined: `local` which uses the `base-template` configuration and `local-bounded` which uses the `extended-template` configuration.

> Be aware that for multi-valued elements (such as `properties`) the inheritance is additive, i.e. the child configuration will be the result of merging the properties from the parent and its own.

4.7. Cache configuration wildcards

An alternative way to apply templates to caches is to use wildcards in the template name, e.g. basecache*. Any cache whose name matches the template wildcard will inherit that configuration.

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

Above, caches basecache-1 and basecache-2 will use the basecache* configuration. The configuration will also be applied when retrieving undefined caches programmatically.

> If a cache name matches multiple wildcards, i.e. it is ambiguous, an exception will be thrown.

4.8. XInclude support

The configuration parser supports XInclude which means you can split your XML configuration across multiple files:

infinispan.xml

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

included.xml

```
<local-cache name="cache-1"/>
```

> the parser supports a minimal subset of the XInclude spec (no support for XPointer, fallback, text processing and content negotiation).

4.9. Declarative configuration reference

For more details on the declarative configuration schema, refer to the configuration reference.

If you are using XML editing tools for configuration writing you can use the provided Infinispan schema to assist you.

Chapter 5. Configuring Caches Programmatically

Infinispan programmatic configuration.

5.1. CacheManager and ConfigurationBuilder API

Programmatic Infinispan configuration is centered around the CacheManager and ConfigurationBuilder API. Although every single aspect of Infinispan configuration could be set programmatically, the most usual approach is to create a starting point in a form of XML configuration file and then in runtime, if needed, programmatically tune a specific configuration to suit the use case best.

```
EmbeddedCacheManager manager = new DefaultCacheManager("my-config-file.xml");
Cache defaultCache = manager.getCache();
```

Let’s assume that a new synchronously replicated cache is to be configured programmatically. First, a fresh instance of Configuration object is created using ConfigurationBuilder helper object, and the cache mode is set to synchronous replication. Finally, the configuration is defined/registered with a manager.

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

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

The default cache configuration (or any other cache configuration) can be used as a starting point for creation of a new cache. For example, lets say that `infinispan-config-file.xml` specifies a replicated cache as a default and that a distributed cache is desired with a specific L1 lifespan while at the same time retaining all other aspects of a default cache. Therefore, the starting point would be to read an instance of a default Configuration object and use `ConfigurationBuilder` to construct and modify cache mode and L1 lifespan on a new `Configuration` object. As a final step the configuration is defined/registered with a manager.

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

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

As long as the base configuration is the default named cache, the previous code works perfectly fine. However, other times the base configuration might be another named cache. So, how can new configurations be defined based on other defined caches? Take the previous example and imagine that instead of taking the default cache as base, a named cache called "replicatedCache" is used as base. The code would look something like this:

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

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

Refer to CacheManager , ConfigurationBuilder , Configuration , and GlobalConfiguration javadocs for more details.

5.2. ConfigurationBuilder Programmatic Configuration API

While the above paragraph shows how to combine declarative and programmatic configuration, starting from an XML configuration is completely optional. The ConfigurationBuilder fluent interface style allows for easier to write and more readable programmatic configuration. This approach can be used for both the global and the cache level configuration. GlobalConfiguration objects are constructed using GlobalConfigurationBuilder while Configuration objects are built using ConfigurationBuilder. Let's look at some examples on configuring both global and cache level options with this API:

5.2.1. Enabling JMX MBeans and statistics

Sometimes you might also want to enable collection of global JMX statistics at cache manager level or get information about the transport. To enable global JMX statistics simply do:

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

Please note that by not enabling (or by explicitly disabling) global JMX statistics your are just turning off statistics collection. The corresponding MBean is still registered and can be used to manage the cache manager in general, but the statistics attributes do not return meaningful values.

Further options at the global JMX statistics level allows you to configure the cache manager name which comes handy when you have multiple cache managers running on the same system, or how to locate the JMX MBean Server:

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

5.2.2. Configuring thread pools

Some of the Infinispan features are powered by a group of the thread pool executors which can also be tweaked at this global level. For example:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
   .replicationQueueThreadPool()
     .threadPoolFactory(ScheduledThreadPoolExecutorFactory.create())
  .build();
```

You can not only configure global, cache manager level, options, but you can also configure cache level options such as the cluster mode:

```
Configuration config = new ConfigurationBuilder()
  .clustering()
    .cacheMode(CacheMode.DIST_SYNC)
    .sync()
    .l1().lifespan(25000L)
    .hash().numOwners(3)
  .build();
```

Or you can configure eviction and expiration settings:

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

5.2.3. Configuring transactions and locking

An application might also want to interact with an Infinispan cache within the boundaries of JTA and to do that you need to configure the transaction layer and optionally tweak the locking settings. When interacting with transactional caches, you might want to enable recovery to deal with transactions that finished with an heuristic outcome and if you do that, you will often want to enable JMX management and statistics gathering too:

```
Configuration config = new ConfigurationBuilder()
  .locking()
    .concurrencyLevel(10000).isolationLevel(IsolationLevel.REPEATABLE_READ)
    .lockAcquisitionTimeout(12000L).useLockStriping(false).writeSkewCheck(true)
    .versioning().enable().scheme(VersioningScheme.SIMPLE)
  .transaction()
    .transactionManagerLookup(new GenericTransactionManagerLookup())
    .recovery()
  .jmxStatistics()
  .build();
```

5.2.4. Configuring cache stores

Configuring Infinispan with chained cache stores is simple too:

```
Configuration config = new ConfigurationBuilder()
   .persistence().passivation(false)
   .addSingleFileStore().location("/tmp").async().enable()
   .preload(false).shared(false).threadPoolSize(20).build();
```

5.2.5. Advanced programmatic configuration

The fluent configuration can also be used to configure more advanced or exotic options, such as advanced externalizers:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  .serialization()
    .addAdvancedExternalizer(998, new PersonExternalizer())
    .addAdvancedExternalizer(999, new AddressExternalizer())
  .build();
```

Or, add custom interceptors:

```
Configuration config = new ConfigurationBuilder()
  .customInterceptors().addInterceptor()
    .interceptor(new FirstInterceptor()).position(InterceptorConfiguration.Position
.FIRST)
    .interceptor(new LastInterceptor()).position(InterceptorConfiguration.Position
.LAST)
    .interceptor(new FixPositionInterceptor()).index(8)
    .interceptor(new AfterInterceptor()).after(NonTransactionalLockingInterceptor
.class)
    .interceptor(new BeforeInterceptor()).before(CallInterceptor.class)
  .build();
```

For information on the individual configuration options, please check the configuration guide.

Chapter 6. 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.

6.1. Getting Started with Default Stacks

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

Default JGroups stacks are included in `infinispan-core.jar` and, as a result, are on the classpath.

Programmatic procedure

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

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .defaultTransport()
        .clusterName("qa-cluster")
         // Use default JGroups stacks with the addProperty() method.
        .addProperty("configurationFile", "default-jgroups-tcp.xml")
        .machineId("qa-machine").rackId("qa-rack")
        .build();
```

Declarative procedure

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

```
<infinispan>
  <cache-container default-cache="replicatedCache">
    <!-- Add default JGroups stacks to clustered caches. -->
    <transport stack="tcp" />
    ...
  </cache-container>
</infinispan>
```

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

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

6.1.1. Default JGroups Stacks

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. Minimises the number of open sockets.
`default-jgroups-tcp.xml`	`tcp`	Uses TCP for transport and UDP multicast for discovery. 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.

Next Steps

After you get up and running with the default JGroups stacks, use inheritance to combine, extend, remove, and replace stack properties. See Adjusting and Tuning JGroups Stacks.

6.1.2. Default JGroups Stacks

Infinispan uses the following JGroups `TCP` and `UDP` stacks by default:

```xml
<stack name="udp">
  <transport type="UDP" socket-binding="jgroups-udp"/>
  <protocol type="PING"/>
  <protocol type="MERGE3"/>
  <protocol type="FD_SOCK" socket-binding="jgroups-udp-fd"/>
  <protocol type="FD_ALL"/>
  <protocol type="VERIFY_SUSPECT"/>
  <protocol type="pbcast.NAKACK2"/>
  <protocol type="UNICAST3"/>
  <protocol type="pbcast.STABLE"/>
  <protocol type="pbcast.GMS"/>
  <protocol type="UFC_NB"/>
  <protocol type="MFC_NB"/>
  <protocol type="FRAG3"/>
</stack>
<stack name="tcp">
  <transport type="TCP" socket-binding="jgroups-tcp"/>
  <protocol type="MPING" socket-binding="jgroups-mping"/>
  <protocol type="MERGE3"/>
  <protocol type="FD_SOCK" socket-binding="jgroups-tcp-fd"/>
  <protocol type="FD_ALL"/>
  <protocol type="VERIFY_SUSPECT"/>
  <protocol type="pbcast.NAKACK2">
    <property name="use_mcast_xmit">false</property>
  </protocol>
  <protocol type="UNICAST3"/>
  <protocol type="pbcast.STABLE"/>
  <protocol type="pbcast.GMS"/>
  <protocol type="MFC_NB"/>
  <protocol type="FRAG3"/>
</stack>
```

To improve performance, Infinispan uses some values for properties other than the JGroups default values. You should examine the following files to review the JGroups configuration for Infinispan:

- Infinispan servers
  - `jgroups-defaults.xml`
  - `infinispan-jgroups.xml`
- Embedded Infinispan
  - `default-jgroups-tcp.xml`
  - `default-jgroups-udp.xml`

The default `TCP` stack uses the `MPING` protocol for discovery, which uses `UDP` multicast.

Reference

- JGroups Protocol

* JGroups Discovery Protocols

6.2. Using Inline JGroups Stacks

Use inline JGroups stack definitions to customize cluster transport for optimal network performance.

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

Procedure

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

```
<infinispan>
  <!-- jgroups is the parent for stack declarations. -->
  <jgroups>
    <!-- Add JGroups stacks for Infinispan clustering. -->
    <stack name="prod">
      <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_NB max_credits="3m" min_threshold="0.40" />
      <MFC_NB max_credits="3m" min_threshold="0.40" />
      <FRAG3 />
    </stack>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <!-- Add JGroups stacks to clustered caches. -->
    <transport stack="prod" />
    ...
  </cache-container>
</infinispan>
```

Reference

Infinispan Configuration Schema

6.3. 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.
3. Modify transport properties with the `stack.combine` attribute.

For example, you want to evaluate using a Gossip router for cluster discovery using a `TCP` stack configuration named `prod`.

You can create a new stack named `gossip-prod` that inherits from `prod` and use `stack.combine` to change properties for the Gossip router configuration, as in the following example:

```
<jgroups>
  ...
  <!-- "gossip-prod" inherits properties from "prod" -->
  <stack name="gossip-prod" extends="prod">
    <!-- Use TCPGOSSIP discovery instead of MPING. -->
    <TCPGOSSIP initial_hosts="${jgroups.tunnel.gossip_router_hosts:localhost[12001]}"
             stack.combine="REPLACE" stack.position="MPING" />
    <!-- Remove FD_SOCK. -->
    <FD_SOCK stack.combine="REMOVE"/>
    <!-- Increase VERIFY_SUSPECT. -->
    <VERIFY_SUSPECT timeout="2000"/>
    <!-- Add SYM_ENCRYPT. -->
    <SYM_ENCRYPT sym_algorithm="AES"
                 key_store_name="defaultStore.keystore"
                 store_password="changeit"
                 alias="myKey" stack.combine="INSERT_AFTER" stack.position=
"pbcast.NAKACK2" />
  </stack>
  ...
</jgroups>
```

6.3.1. Stack Combine Attribute

`stack.combine` lets you override and modify inherited JGroups properties.

Value	Description
`COMBINE`	Overrides existing protocol attributes.
`REPLACE`	Replaces existing protocols that you identify with the `stack.position` attribute. 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.
`INSERT_AFTER`	Inserts protocols after any protocols that you identify with the `stack.position` attribute.
`REMOVE`	Removes protocols from the inherited configuration.

6.4. Using JGroups Stacks in External Files

Use JGroups transport configuration from external files.

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

Programmatic procedure

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

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
        .defaultTransport()
        .clusterName("prod-cluster")
         // Add custom JGroups stacks with the addProperty() method.
        .addProperty("configurationFile", "prod-jgroups-tcp.xml")
        .machineId("prod-machine").rackId("prod-rack")
        .build();
```

Declarative procedure

- Add your JGroups stack file and then configure the Infinispan cluster to use it.

```
<infinispan>
  <jgroups>
     <!-- Add custom JGroups stacks in external files. -->
     <stack-file name="prod-tcp" path="prod-jgroups-tcp.xml"/>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <!-- Add custom JGroups stacks to clustered caches. -->
    <transport stack="prod-tcp" />
    <replicated-cache name="replicatedCache"/>
  </cache-container>
  ...
</infinispan>
```

Reference

- GlobalConfigurationBuilder.transport()
- TransportConfigurationBuilder

6.5. 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
```

6.5.1. System Properties for Default JGroups Stacks

`default-jgroups-udp.xml`

System Property	Description	Default Value	Required/Optional
`jgroups.udp.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.udp.mcast_port`	Port for the multicast socket.	`46655`	Optional
`jgroups.udp.ip_ttl`	Specifies the 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

`default-jgroups-tcp.xml`

System Property	Description	Default Value	Required/Optional
`jgroups.tcp.address`	IP address for TCP transport.	`127.0.0.1`	Optional
`jgroups.tcp.port`	Port for the TCP socket.	`7800`	Optional
`jgroups.udp.mcast_addr`	IP address for multicast discovery. The IP address must be a valid "class D" address that is suitable for IP multicast.	`228.6.7.8`	Optional
`jgroups.udp.mcast_port`	Port for the multicast socket.	`46655`	Optional
`jgroups.udp.ip_ttl`	Specifies the 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

`default-jgroups-ec2.xml`

System Property	Description	Default Value	Required/Optional
`jgroups.tcp.address`	IP address for TCP transport.	`127.0.0.1`	Optional
`jgroups.tcp.port`	Port for the TCP socket.	`7800`	Optional
`jgroups.s3.access_key`	Amazon S3 access key for an S3 bucket.	No default value.	Optional

System Property	Description	Default Value	Required/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 already exist and be unique.	No default value.	Optional

`default-jgroups-kubernetes.xml`

System Property	Description	Default Value	Required/Optional
`jgroups.tcp.address`	IP address for TCP transport.	`eth0`	Optional
`jgroups.tcp.port`	Port for the TCP socket.	`7800`	Optional

Reference

- JGroups System Properties
- JGroups Protocol List

6.6. 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

Chapter 7. 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.

7.1. 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. 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

7.3. 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)

7.4. 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

7.5. 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>
  <!-- Replace ${version.jgroups.native_s3_ping} with the
  version of the native-s3-ping module you want to use. -->
  <version>${version.jgroups.native_s3_ping}</version>
</dependency>
```

7.6. 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

7.7. 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>
  <!-- Replace ${version.jgroups.azure} with the
  version of the jgroups-azure module you want to use. -->
  <version>${version.jgroups.azure}</version>
</dependency>
```

7.8. 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>
  <!-- Replace ${version.jgroups.google} with the
  version of the jgroups-goole module you want to use. -->
  <version>${version.jgroups.google}</version>
</dependency>
```

Chapter 8. Configuring Eviction and Expiration

8.1. Eviction and Data Container

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

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

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

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

8.2. Enabling Eviction

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

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

8.2.1. Eviction strategy

Strategies control how the eviction is handled.

The possible choices are

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

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

`REMOVE`

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

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

`EXCEPTION`

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

8.2.2. Eviction types

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

`COUNT`

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

`MEMORY`

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

8.2.3. Storage type

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

`OBJECT`

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

`BINARY`

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

`OFF-HEAP`

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

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

8.2.4. More defaults

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

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

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

8.3. Expiration

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

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

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

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

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

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

8.3.1. Difference between Eviction and Expiration

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

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

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

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

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

8.3.2. Expiration details

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

Maximum Idle Expiration

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

Local Max Idle

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

- accessed directly (`Cache.get`).

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

Clustered Max Idle

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

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

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

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

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

Configuration

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

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

Programmatically, the same would be defined using:

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

Memory Based Eviction Configuration

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

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

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

Default values

Eviction is disabled by default. Default values are used:

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

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

Using expiration

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

For example, assume the following configuration:

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

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

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

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

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

8.3.3. Expiration designs

Central to expiration is an ExpirationManager.

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

The expiration manager processes expirations in the following manner:

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

Chapter 9. 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.

9.1. Infinispan Cache Stores

Cache stores connect Infinispan to persistent data sources and implement the following interfaces:

`org.infinispan.persistence.spi.CacheLoader`
: Allows Infinispan to load data from persistent storage.

`org.infinispan.persistence.spi.CacheWriter`
: Allows Infinispan to persist data to persistent storage.

9.1.1. 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. Add Infinispan cache stores with the appropriate configuration, as in the following examples:

Declarative configuration for a file-based cache 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="false"
           path="${java.io.tmpdir}">
      <write-behind thread-pool-size="5" />
   </file-store>
</persistence>
```

Declarative configuration for a custom cache store configuration

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

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

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

Custom cache stores include `property` parameters that let you configure specific attributes for your cache store.

Programmatic configuration for a single file cache store

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

Reference

- Infinispan Configuration Schema
- Infinispan Cache Store Implementations
- Creating Custom Cache Stores

9.1.2. 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 threads 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.

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

9.1.3. 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

9.1.4. 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.

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

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

Reference

Key Ownership

9.1.5. 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.

9.1.6. Write-Through

Write-Through is an 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

9.1.7. Write-Behind

Write-Behind is an cache writing mode where writes to memory are synchronous and writes to cache stores are asynchronous. When a client application updates a cache entry, Infinispan adds the update to a modification queue and then modifies the cache store in a different thread than the client thread.

You can configure the number of threads that consume the modification queue and apply updates to the underlying cache store. The modification queue fills up if there are not enough threads to handle the updates or if the underlying cache store becomes unavailable. When this occurs, Infinispan uses Write-Through mode until the modification queue can accept new entries.

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"
                 thread-pool-size="23"
                 fail-silently="true"/>
   </file-store>
</persistence>
```

Reference

Write-Through

9.2. Cache Store Implementations

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

9.2.1. 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.

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

9.2.2. Single File Cache Stores

Single File cache stores, `SingleFileStore`, persist data to a single file and maintains an in-memory index of keys and their values in that file.

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

Programmatic configuration

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

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

- Infinispan configuration schema SingleFileStore

9.2.3. 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` supports segmentation and uses an additional column in the database table to represent the segment to which an entry belongs.

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.

Specify a connection factory declaratively with the `<connectionPool />`, `<dataSource />`, or `<simpleConnection />` elements.

Specify a connection factory programmatically with the `connectionPool()`, `dataSource()`, or `simpleConnection()` methods in the `JdbcStringBasedStoreConfigurationBuilder` class.

Declarative configuration with `PooledConnectionFactory`

```
<persistence>
   <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:10.1"
     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" />
      </string-keyed-table>
   </string-keyed-jdbc-store>
</persistence>
```

Declarative configuration with `ManagedConnectionFactory`

```
<string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:10.1"
  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" />
   </string-keyed-table>
</string-keyed-jdbc-store>
```

Programmatic configuration with `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")
      .connectionPool()
         .connectionUrl("jdbc:h2:mem:infinispan_string_based;DB_CLOSE_DELAY=-1")
         .username("sa")
         .driverClass("org.h2.Driver");
```

Programmatic configuration with `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")
      .dataSource()
         .jndiUrl("java:/StringStoreWithManagedConnectionTest/DS");
```

Reference

- JDBC cache store configuration schema
- JdbcStringBasedStore
- JdbcStringBasedStoreConfiguration

9.2.4. 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:10.1"
         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.
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

The following is an example for using a JPA cache store:

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

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

2. Create a user entity class.

```
@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.

```
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:

```
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:

```
@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;
    ...
}
```

9.2.5. Remote Cache Stores

Remote cache stores, `RemoteStore`, use a remote Infinispan cluster as storage.

`RemoteStore` uses the Hot Rod protocol to communicate with remote Infinispan clusters.

In the following configuration examples, `RemoteStore` uses the remote cache named "mycache" on Infinispan servers "one" and "two":

Declarative configuration

```
<persistence>
   <remote-store xmlns="urn:infinispan:config:store:remote:10.1" 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 with Infinispan Hot Rod protocol version 2.3 or later.

> Ensure the number of segments and hash are the same between the Remote cache store and Infinispan servers, otherwise bulk operations do not return correct results.

Reference

- Remote cache store configuration schema
- RemoteStore
- RemoteStoreConfigurationBuilder

9.2.6. 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:10.1" path=
"/tmp/rocksdb/data">
         <expiration path="/tmp/rocksdb/expired"/>
         <property name="database.max_background_compactions">2</property>
         <property name="data.write_buffer_size">512MB</property>
      </rocksdb-store>
   </persistence>
</local-cache>
```

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(...));
```

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

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

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 gets automatically created.
`expiredLocation`	Specifies the path to the RocksDB database that provides the cache store for expired data. If you do not set the location, it gets automatically created.

Parameter	Description
`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.

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

9.2.7. 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:10.1">
        <index path="/tmp/sifs/testCache/index" />
        <data path="/tmp/sifs/testCache/data" />
    </soft-index-file-store>
</persistence>
```

Programmatic configuration

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

9.2.8. Custom Cache Stores

You can create custom cache stores that implement one or more of the Infinispan persistent SPIs.

Reference

- Persistence SPI
- Persistence SPI package summary

Implementing Custom Cache Stores

Create custom cache stores that implement both `CacheWriter` and `CacheLoader` interfaces to fetch and persist data to external storage.

1. Implement the appropriate Infinispan persistent SPIs.
2. Annotate your store class with the `@Store` annotation and specify the appropriate properties.

   For example, if your cache store is shared, use the `@Store(shared = true)` annotation.

3. Create a custom cache store configuration and builder.
   a. Extend `AbstractStoreConfiguration` and `AbstractStoreConfigurationBuilder`.

      Extend `AbstractSegmentedStoreConfiguration` instead of `AbstractStoreConfiguration` for a segmented cache store that creates a different store instance per segment.

   b. Optionally add the following annotations to ensure that your custom configuration builder parses your cache store configuration from XML:
      - `@ConfigurationFor`
      - `@BuiltBy`
      - `@ConfiguredBy`

      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 store and its configuration do not declare the `@Store` and `@ConfigurationFor` annotations, a warning message is logged when Infinispan initializes the cache.

Custom Cache Store Configuration

After you implement your custom cache store, 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 a file under `META-INF/services/` that contains the fully qualified class name of your store implementation.

   The name of the service file should reflect the interface that your store implements. For example, if your store implements the `AdvancedCacheWriter` interface then you should create the following file:

   `/META-INF/services/org.infinispan.persistence.spi.AdvancedCacheWriter`
3. Add your JAR file to the `server/lib` directory of your Infinispan server.

9.3. Infinispan Persistence SPIs

Infinispan Service Provider Interfaces (SPI) enable read and write operations to external storage and provide the following features:

Portability across JCache-compliant vendors

The Infinispan `CacheWriter` and `CacheLoader` interfaces align with the `JSR-107` JCache specification.

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

- JSR-107
- CacheWriter
- CacheLoader

9.3.1. Persistence SPI Classes

The following notes apply to Infinispan persistence SPI classes:

ByteBuffer

Abstracts the serialized form of an object.

MarshallableEntry

Abstracts the information held within a persistent store corresponding to a key/value added to the cache. Provides a method for reading this information both in serialized (ByteBuffer) and deserialized (Object) format. Normally data read from the store is kept in serialized format and lazily deserialized on demand, within the MarshallableEntry implementation.

CacheWriter and CacheLoader

Provide basic methods for writing to and reading from cache stores.

AdvancedCacheLoader and AdvancedCacheWriter

Provide bulk operations to manipulate the underlaying storage, such as parallel iteration and purging of expired entries, clear and size.

SegmentedAdvancedLoadWriteStore

Provides all the operations that deal with segments.

Cache stores can be segmented if they do one of the following:

- Implement the SegmentedAdvancedLoadWriteStore interface. In this case only a single store instance is used per cache.
- Has a configuration that extends the AbstractSegmentedConfiguration abstract class.

  This requires you to implement the `newConfigurationFrom()` method where it is expected that a new `StoreConfiguration` instance is created per invocation. This creates a store instance per segment to which a node can write. Stores might start and stop as data is moved between nodes.

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

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

If you want to migrate your existing store to the new API or to write a new cache store implementation, the SingleFileStore provides a good starting point.

9.4. Migrating Cache Stores Between Infinispan Versions

9.4.1. Store Migrator

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

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

Migrating Cache Stores

To perform a migration with `StoreMigrator`,

1. Put `infinispan-tools-10.1.jar` and dependencies for your source and target databases, such as JDBC drivers, on your classpath.
2. Create a `.properties` file that contains configuration properties for the source and target cache stores.

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

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

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

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

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

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

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

Store Migrator Properties

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

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

Common Properties

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

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

JDBC Properties

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

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

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

- `CURRENT` Infinispan 9.x marshaller.

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

<table>
<tr><th>Property</th><th>Description</th><th>Example value</th><th>Required</th></tr>
<tr><td>table.<binary|string>.<id|data|timestamp>.type</td><td>Type of the column</td><td>VARCHAR</td><td>TRUE</td></tr>
<tr><td>key_to_string_mapper</td><td>TwoWayKey2StringMapper Class</td><td>org.infinispan.persistence.keymappers.DefaultTwoWayKey2StringMapper</td><td></td></tr>
</table>

LevelDB/RocksDB Properties

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

SingleFileStore Properties

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

SoftIndexFileStore Properties

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