Configuring Infinispan Caches

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 API

`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

The `CacheManager` API is the starting point for interacting with Infinispan caches. Cache managers control cache lifecycle; creating, modifying, and deleting cache instances.

Infinispan provides two `CacheManager` implementations:

`EmbeddedCacheManager`
: Entry point for caches when running Infinispan inside the same Java Virtual Machine (JVM) as the client application.

`RemoteCacheManager`
: Entry point for caches when running Infinispan Server in its own JVM. When you instantiate a `RemoteCacheManager` it establishes a persistent TCP connection to Infinispan Server through the Hot Rod endpoint.

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

1.3. 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
: Infinispan runs as a single node and never replicates read or write operations on cache entries.

Replicated

Infinispan replicates all cache entries on all nodes in a cluster and performs local read operations only.

Distributed

Infinispan replicates cache entries on a subset of nodes in a cluster and assigns entries to fixed owner nodes.
Infinispan requests read operations from owner nodes to ensure it returns the correct value.

Invalidation

Infinispan evicts stale data from all nodes whenever operations modify entries in the cache. Infinispan performs local read operations only.

Scattered

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.

1.3.1. Comparison of cache modes

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)
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**	**No indexing**	**All**	**All**	**No TX**

1.4. Local caches

Infinispan offers a local cache mode that is similar to a `ConcurrentHashMap`.

Caches offer more capabilities than simple maps, including write-through and write-behind to persistent storage as well as management capabilities such as eviction and expiration.

The Infinispan `Cache` API extends the `ConcurrentMap` API in Java, making it easy to migrate from a map to a Infinispan cache.

Local cache configuration

XML

```
<local-cache name="mycache"
             statistics="true">
  <encoding media-type="application/x-protostream"/>
</distributed-cache>
```

JSON

```
{
  "local-cache": {
    "name": "mycache",
    "statistics": "true",
    "encoding": {
      "media-type": "application/x-protostream"
    }
  }
}
```

YAML

```
localCache:
  name: "mycache"
  statistics: "true"
  encoding:
    mediaType: "application/x-protostream"
```

1.4.1. Simple caches

A simple cache is a type of local cache that disables support for the following capabilities:

- Transactions and invocation batching
- Persistent storage
- Custom interceptors
- Indexing

- Transcoding

However, you can use other Infinispan capabilities with simple caches such as expiration, eviction, statistics, and security features. If you configure a capability that is not compatible with a simple cache, Infinispan throws an exception.

Simple cache configuration

XML

```
<local-cache simple-cache="true" />
```

JSON

```
{
  "local-cache" : {
    "simple-cache" : "true"
  }
}
```

YAML

```
localCache:
  simpleCache: "true"
```

Chapter 2. Clustered caches

You can create embedded and remote caches on Infinispan clusters that replicate data across nodes.

2.1. Replicated caches

Infinispan replicates all entries in the cache to all nodes in the cluster. Each node can perform read operations locally.

Replicated caches provide a quick and easy way to share state across a cluster, but is suitable for clusters of less than ten nodes. Because the number of replication requests scales linearly with the number of nodes in the cluster, using replicated caches with larger clusters reduces performance. However you can use UDP multicasting for replication requests to improve performance.

Each key has a primary owner, which serializes data container updates in order to provide consistency.

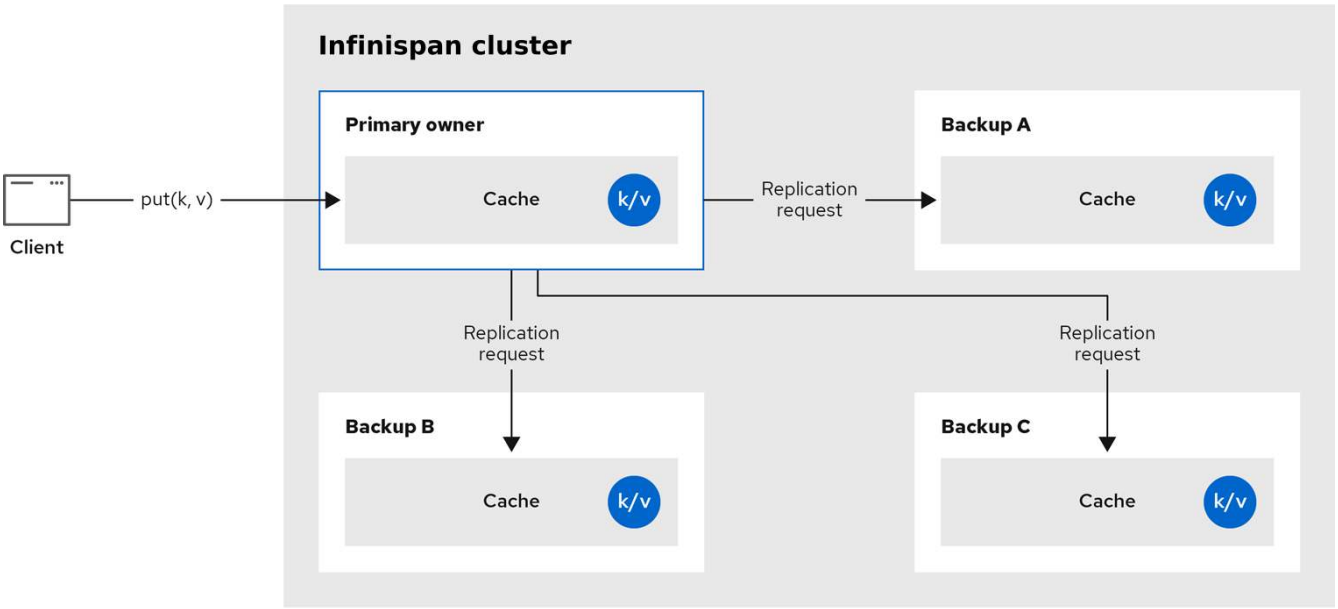

Figure 1. Replicated cache

Synchronous or asynchronous replication

- 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.

Transactions

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.

2.2. Distributed caches

Infinispan attempts to keep a fixed number of copies of any entry in the cache, configured as `numOwners`. This allows distributed caches 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.

Infinispan splits the owners of a key into one **primary owner**, which coordinates writes to the key, and zero or more **backup owners**.

The following diagram shows a write operation that a client sends to a backup owner. In this case the backup node forwards the write to the primary owner, which then replicates the write to the backup.

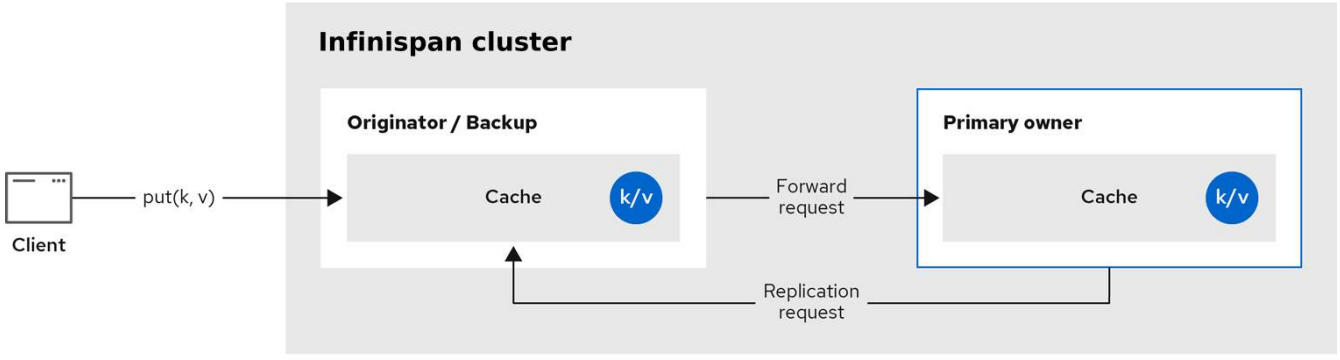

Figure 2. Cluster replication

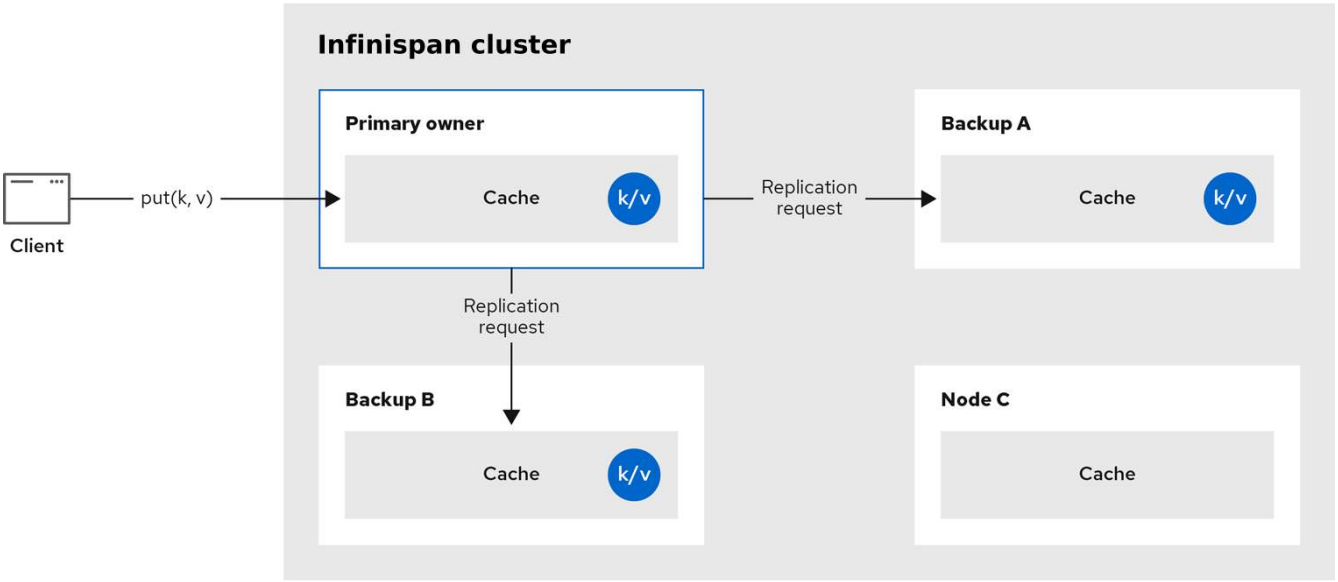

Figure 3. Distributed cache

Read operations

Read operations request the value from the primary owner. If the primary owner does not respond in a reasonable amount of time, Infinispan requests the value from the backup owners as well.

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.

Write operations

Write operations result in at most `2 * numOwners` messages. One message from the originator to the primary owner and `numOwners - 1` messages from the primary to the backup nodes along with the corresponding acknowledgment messages.

Cache topology changes may cause retries and additional messages for both read and write operations.

Synchronous or asynchronous replication

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.

Transactions

Transactional distributed caches send lock/prepare/commit/unlock messages to the affected nodes only, meaning all nodes that own at least one key affected by the transaction. 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.

2.2.1. Read consistency

Even with synchronous replication, distributed caches are not linearizable. For transactional

caches, they do not support serialization/snapshot isolation.

For example, a thread is carrying out a single put request:

```
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.

2.2.2. 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.

Consistent hash factory implementation	Description
`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.

Consistent hash factory implementation	Description
`TopologyAwareSyncConsistentHashFactory`	Equivalent to `SyncConsistentHashFactory` but used with server hinting to distribute data across the topology so that backed up copies of data are stored on different nodes in the topology than the primary owners. This is the default consistent hashing implementation with server hinting.
`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.
`TopologyAwareConsistentHashFactory`	Equivalent to `DefaultConsistentHashFactory` but used with server hinting to distribute data across the topology so that backed up copies of data are stored on different nodes in the topology than the primary owners.
`ReplicatedConsistentHashFactory`	Used internally to implement replicated caches. You should never explicitly select this algorithm in a distributed cache.

Hashing configuration

You can configure `ConsistentHashFactory` implementations, including custom ones, with embedded caches only.

XML

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

ConfigurationBuilder

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

Additional resources

- KeyPartitioner

2.2.3. Capacity factors

Capacity factors allocate the number of segments based on resources available to each node in the cluster.

The capacity factor for a node applies to segments for which that node is both the primary owner and backup owner. In other words, the capacity factor specifies is the total capacity that a node has in comparison to other nodes in the cluster.

The default value is `1` which means that all nodes in the cluster have an equal capacity and Infinispan allocates the same number of segments to all nodes in the cluster.

However, if nodes have different amounts of memory available to them, you can configure the capacity factor so that the Infinispan hashing algorithm assigns each node a number of segments weighted by its capacity.

The value for the capacity factor configuration must be a positive number and can be a fraction such as 1.5. You can also configure a capacity factor of `0` but is recommended only for nodes that join the cluster temporarily and should use the zero capacity configuration instead.

Zero capacity nodes

You can configure nodes where the capacity factor is `0` for every cache, user defined caches, and internal caches. When defining a zero capacity node, the node does not hold any data.

Zero capacity node configuration

XML

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

JSON

```json
{
  "infinispan" : {
    "cache-container" : {
      "zero-capacity-node" : "true"
    }
  }
}
```

YAML

```yaml
infinispan:
  cache-container:
    zero-capacity-node: "true"
```

ConfigurationBuilder

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

2.2.4. Level one (L1) caches

Infinispan nodes create local replicas when they retrieve entries from another node in the clsuter. L1 caches avoid repeatedly looking up entries on primary owner nodes and adds performance.

The following diagram illustrates how L1 caches work:

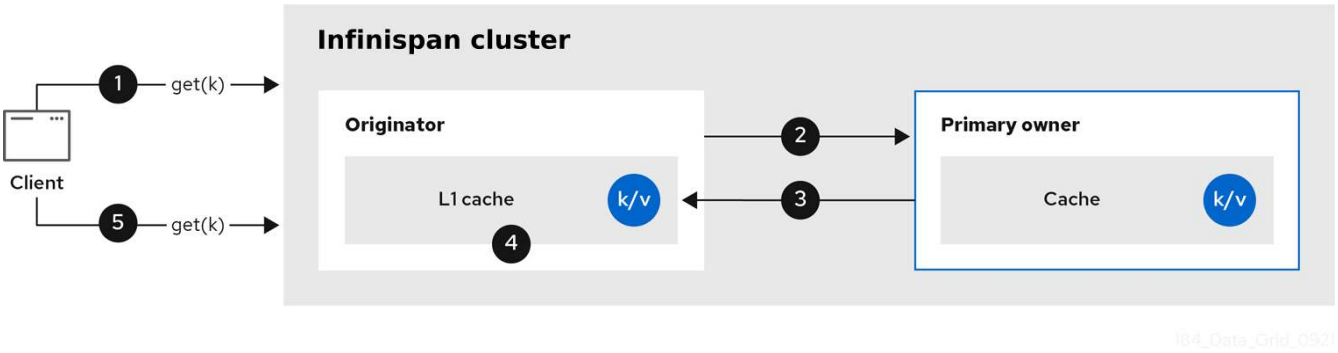

Figure 4. L1 cache

In the "L1 cache" diagram:

1. A client invokes `cache.get()` to read an entry for which another node in the cluster is the primary owner.
2. The originator node forwards the read operation to the primary owner.
3. The primary owner returns the key/value entry.
4. The originator node creates a local copy.
5. Subsequent `cache.get()` invocations return the local entry instead of forwarding to the primary owner.

L1 caching performance

Enabling L1 improves performance for read operations but requires primary owner nodes to broadcast invalidation messages when entries are modified. This ensures that Infinispan removes any out of date replicas across the cluster. However this also decreases performance of write operations and increases memory usage, reducing overall capacity of caches.

Infinispan evicts and expires local replicas, or L1 entries, like any other cache entry.

L1 cache configuration

XML

```
<distributed-cache l1-lifespan="5000"
                   l1-cleanup-interval="60000">
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "l1-lifespan": "5000",
    "l1-cleanup-interval": "60000"
  }
}
```

YAML

```
distributedCache:
  l1Lifespan: "5000"
  l1-cleanup-interval: "60000"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clustering().cacheMode(CacheMode.DIST_SYNC)
         .l1()
         .lifespan(5000, TimeUnit.MILLISECONDS)
         .cleanupTaskFrequency(60000, TimeUnit.MILLISECONDS);
```

2.2.5. 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.

Server hinting configuration

XML

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

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "transport" : {
        "cluster" : "MyCluster",
        "machine" : "LinuxServer01",
        "rack" : "Rack01",
        "site" : "US-WestCoast"
      }
    }
  }
}
```

YAML

```
cache-container:
  transport:
    cluster: "MyCluster"
    machine: "LinuxServer01"
    rack: "Rack01"
    site: "US-WestCoast"
```

GlobalConfigurationBuilder

```
GlobalConfigurationBuilder global = GlobalConfigurationBuilder.
defaultClusteredBuilder()
  .transport()
  .clusterName("MyCluster")
  .machineId("LinuxServer01")
  .rackId("Rack01")
  .siteId("US-WestCoast");
```

2.2.6. 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, Infinispan 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.

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

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

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

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

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

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

Lifecycle

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

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

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

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

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

Topology changes

When the cache topology changes, 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.

2.2.7. Grouping API

Complementary to the 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.

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.

When the `Grouping` 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).

To use the `Grouping` API, you must enable groups.

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

```
<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, for example:

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

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

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

The group method must return a `String`

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

```
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.

Example `Grouper` implementation

```
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:

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

Or, if you are using XML:

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

Advanced API

`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.

2.3. Invalidation caches

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.

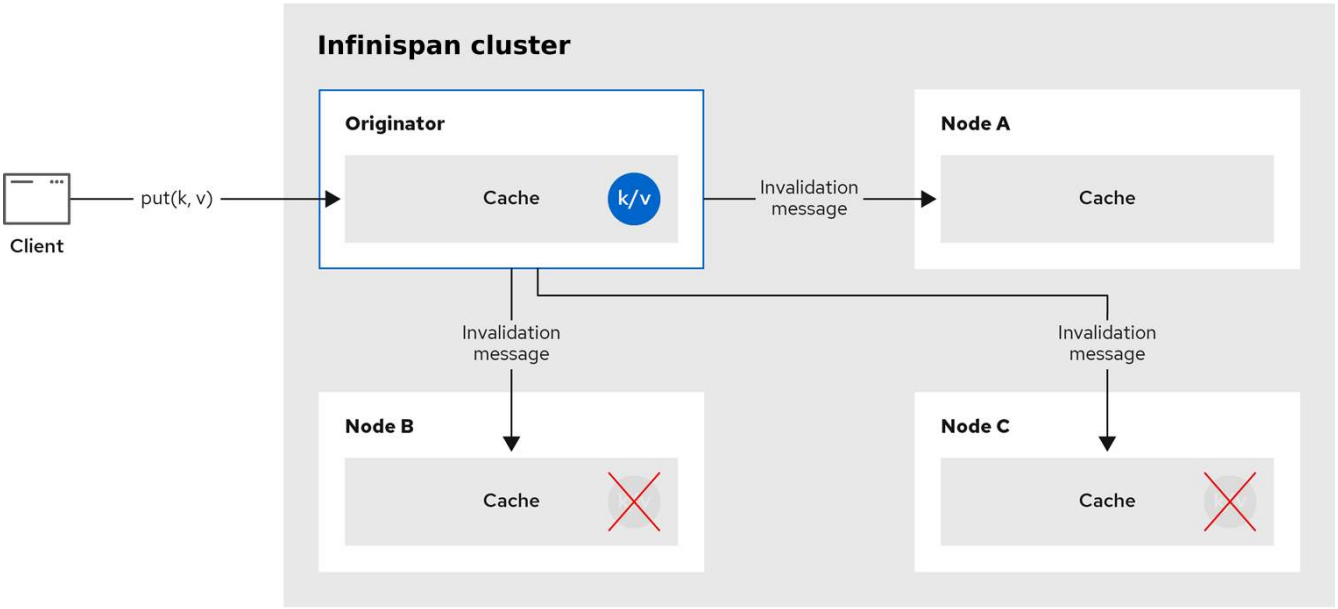

Figure 5. Invalidation cache

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, non-shared, cache 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.

Synchronous or asynchronous replication

When synchronous, a write blocks until all nodes in the cluster have evicted the stale value. When asynchronous, the originator broadcasts invalidation messages but does not wait for responses. That means other nodes still see the stale value for a while after the write completed on the originator.

Transactions

Transactions can be used to batch the invalidation messages. Transactions acquire the key lock on 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 (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.

2.4. Scattered caches

Scattered caches are very similar to distributed caches as they allow linear scaling of the cluster. Scattered caches allow single node failure by maintaining two copies of the data (`numOwners=2`). Unlike distributed caches, 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 (distributed caches require 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 lowers the performance of scattered caches in clusters with a large number of nodes.

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.

You cannot use scattered caches with transactions or asynchronous replication.

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

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

```
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.

2.5. Asynchronous replication

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.

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.

2.5.1. Return values with asynchronous replication

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.

2.6. Configuring initial cluster size

Infinispan handles cluster topology changes dynamically. This means that nodes do not need to wait for other nodes to join the cluster before Infinispan initializes the caches.

If your applications require a specific number of nodes in the cluster before caches start, you can configure the initial cluster size as part of the transport.

Procedure

1. Open your Infinispan configuration for editing.
2. Set the minimum number of nodes required before caches start with the `initial-cluster-size` attribute or `initialClusterSize()` method.
3. Set the timeout, in milliseconds, after which the cache manager does not start with the `initial-cluster-timeout` attribute or `initialClusterTimeout()` method.

4. Save and close your Infinispan configuration.

Initial cluster size configuration

XML

```xml
<infinispan>
  <cache-container>
    <transport initial-cluster-size="4"
               initial-cluster-timeout="30000" />
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "transport" : {
        "initial-cluster-size" : "4",
        "initial-cluster-timeout" : "30000"
      }
    }
  }
}
```

YAML

```
infinispan:
  cache-container:
    transport:
      initial-cluster-size: "4"
      initial-cluster-timeout: "30000"
```

ConfigurationBuilder

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

Chapter 3. Infinispan cache configuration

Cache configuration controls how Infinispan stores your data.

As part of your cache configuration, you declare the cache mode you want to use. For instance, you can configure Infinispan clusters to use replicated caches or distributed caches.

Your configuration also defines the characteristics of your caches and enables the Infinispan capabilities that you want to use when handling data. For instance, you can configure how Infinispan encodes entries in your caches, whether replication requests happen synchronously or asynchronously between nodes, if entries are mortal or immortal, and so on.

3.1. Declarative cache configuration

You can configure caches declaratively, in XML or JSON format, according to the Infinispan schema.

Declarative cache configuration has the following advantages over programmatic configuration:

Portability

Define each configuration in a standalone file that you can use to create embedded and remote caches.
You can also use declarative configuration to create caches with Infinispan Operator for clusters running on Kubernetes.

Simplicity

Keep markup languages separate to programming languages.
For example, to create remote caches it is generally better to not add complex XML directly to Java code.

Infinispan Server configuration extends `infinispan.xml` to include cluster transport mechanisms, security realms, and endpoint configuration. If you declare caches as part of your Infinispan Server configuration you should use management tooling, such as Ansible or Chef, to keep it synchronized across the cluster.

To dynamically synchronize remote caches across Infinispan clusters, create them at runtime.

3.1.1. Cache configuration

You can create declarative cache configuration in XML, JSON, and YAML format.

All declarative caches must conform to the Infinispan schema. Configuration in JSON format must follow the structure of an XML configuration, elements correspond to objects and attributes correspond to fields.

Distributed caches

XML

```xml
<distributed-cache owners="2"
                   segments="256"
                   capacity-factor="1.0"
                   l1-lifespan="5000"
                   mode="SYNC"
                   statistics="true">
  <encoding media-type="application/x-protostream"/>
  <locking isolation="REPEATABLE_READ"/>
  <transaction mode="FULL_XA"
               locking="OPTIMISTIC"/>
  <expiration lifespan="5000"
              max-idle="1000" />
  <memory max-count="1000000"
          when-full="REMOVE"/>
  <indexing enabled="true"
            storage="local-heap">
    <index-reader refresh-interval="1000"/>
  </indexing>
  <partition-handling when-split="ALLOW_READ_WRITES"
                      merge-policy="PREFERRED_NON_NULL"/>
  <persistence passivation="false">
    <!-- Persistent storage configuration. -->
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "mode": "SYNC",
    "owners": "2",
    "segments": "256",
    "capacity-factor": "1.0",
    "l1-lifespan": "5000",
    "statistics": "true",
    "encoding": {
      "media-type": "application/x-protostream"
    },
    "locking": {
      "isolation": "REPEATABLE_READ"
    },
    "transaction": {
      "mode": "FULL_XA",
      "locking": "OPTIMISTIC"
    },
    "expiration" : {
      "lifespan" : "5000",
      "max-idle" : "1000"
    },
    "memory": {
      "max-count": "1000000",
      "when-full": "REMOVE"
    },
    "indexing" : {
      "enabled" : true,
      "storage" : "local-heap",
      "index-reader" : {
        "refresh-interval" : "1000"
      }
    },
    "partition-handling" : {
      "when-split" : "ALLOW_READ_WRITES",
      "merge-policy" : "PREFERRED_NON_NULL"
    },
    "persistence" : {
      "passivation" : false
    }
  }
}
```

YAML

```yaml
distributedCache:
  mode: "SYNC"
  owners: "2"
  segments: "256"
  capacityFactor: "1.0"
  l1Lifespan: "5000"
  statistics: "true"
  encoding:
    mediaType: "application/x-protostream"
  locking:
    isolation: "REPEATABLE_READ"
  transaction:
    mode: "FULL_XA"
    locking: "OPTIMISTIC"
  expiration:
    lifespan: "5000"
    maxIdle: "1000"
  memory:
    maxCount: "1000000"
    whenFull: "REMOVE"
  indexing:
    enabled: "true"
    storage: "local-heap"
    indexReader:
      refreshInterval: "1000"
  partitionHandling:
    whenSplit: "ALLOW_READ_WRITES"
    mergePolicy: "PREFERRED_NON_NULL"
  persistence:
    passivation: "false"
    # Persistent storage configuration.
```

Replicated caches

XML

```xml
<replicated-cache segments="256"
                  mode="SYNC"
                  statistics="true">
  <encoding media-type="application/x-protostream"/>
  <locking isolation="REPEATABLE_READ"/>
  <transaction mode="FULL_XA"
               locking="OPTIMISTIC"/>
  <expiration lifespan="5000"
              max-idle="1000" />
  <memory max-count="1000000"
          when-full="REMOVE"/>
  <indexing enabled="true"
            storage="local-heap">
    <index-reader refresh-interval="1000"/>
  </indexing>
  <partition-handling when-split="ALLOW_READ_WRITES"
                      merge-policy="PREFERRED_NON_NULL"/>
  <persistence passivation="false">
    <!-- Persistent storage configuration. -->
  </persistence>
</replicated-cache>
```

JSON

```
{
  "replicated-cache": {
    "mode": "SYNC",
    "segments": "256",
    "statistics": "true",
    "encoding": {
      "media-type": "application/x-protostream"
    },
    "locking": {
      "isolation": "REPEATABLE_READ"
    },
    "transaction": {
      "mode": "FULL_XA",
      "locking": "OPTIMISTIC"
    },
    "expiration" : {
      "lifespan" : "5000",
      "max-idle" : "1000"
    },
    "memory": {
      "max-count": "1000000",
      "when-full": "REMOVE"
    },
    "indexing" : {
      "enabled" : true,
      "storage" : "local-heap",
      "index-reader" : {
        "refresh-interval" : "1000"
      }
    },
    "partition-handling" : {
      "when-split" : "ALLOW_READ_WRITES",
      "merge-policy" : "PREFERRED_NON_NULL"
    },
    "persistence" : {
      "passivation" : false
    }
  }
}
```

YAML

```yaml
replicatedCache:
  mode: "SYNC"
  segments: "256"
  statistics: "true"
  encoding:
    mediaType: "application/x-protostream"
  locking:
    isolation: "REPEATABLE_READ"
  transaction:
    mode: "FULL_XA"
    locking: "OPTIMISTIC"
  expiration:
    lifespan: "5000"
    maxIdle: "1000"
  memory:
    maxCount: "1000000"
    whenFull: "REMOVE"
  indexing:
    enabled: "true"
    storage: "local-heap"
    indexReader:
      refreshInterval: "1000"
  partitionHandling:
    whenSplit: "ALLOW_READ_WRITES"
    mergePolicy: "PREFERRED_NON_NULL"
  persistence:
    passivation: "false"
    # Persistent storage configuration.
```

Additional resources

- Infinispan configuration schema reference
- infinispan-config-13.0.xsd

3.2. Adding cache templates

The Infinispan schema includes `*-cache-configuration` elements that you can use to create templates. You can then create caches on demand, using the same configuration multiple times.

Procedure

1. Open your Infinispan configuration for editing.
2. Add the cache configuration with the appropriate `*-cache-configuration` element or object to the cache manager.
3. Save and close your Infinispan configuration.

Cache template example

XML

```xml
<infinispan>
   <cache-container>
      <distributed-cache-configuration name="my-dist-template"
                                       mode="SYNC"
                                       statistics="true">
        <encoding media-type="application/x-protostream"/>
        <memory max-count="1000000"
                when-full="REMOVE"/>
        <expiration lifespan="5000"
                    max-idle="1000"/>
      </distributed-cache-configuration>
   </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "distributed-cache-configuration" : {
        "name" : "my-dist-template",
        "mode": "SYNC",
        "statistics": "true",
        "encoding": {
          "media-type": "application/x-protostream"
        },
        "expiration" : {
          "lifespan" : "5000",
          "max-idle" : "1000"
        },
        "memory": {
          "max-count": "1000000",
          "when-full": "REMOVE"
        }
      }
    }
  }
}
```

YAML

```yaml
infinispan:
  cacheContainer:
    distributedCacheConfiguration:
      name: "my-dist-template"
      mode: "SYNC"
      statistics: "true"
      encoding:
        mediaType: "application/x-protostream"
      expiration:
        lifespan: "5000"
        maxIdle: "1000"
      memory:
        maxCount: "1000000"
        whenFull: "REMOVE"
```

3.2.1. Creating caches from templates

Create caches from configuration templates.

Templates for remote caches are available from the **Cache templates** menu in Infinispan Console.

Prerequisites

- Add at least one cache template to the cache manager.

Procedure

1. Open your Infinispan configuration for editing.
2. Specify the template from which the cache inherits with the `configuration` attribute or field.
3. Save and close your Infinispan configuration.

Cache configuration inherited from a template

XML

```xml
<distributed-cache configuration="my-dist-template" />
```

JSON

```json
{
  "distributed-cache": {
    "configuration": "my-dist-template"
  }
}
```

YAML

```yaml
distributedCache:
  configuration: "my-dist-template"
```

3.2.2. Cache template inheritance

Cache configuration templates can inherit from other templates to extend and override settings.

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

Additionally, template inheritance is additive for elements that have multiple values. A cache that inherits from another template merges the values from that template, which can override properties.

Template inheritance example

XML

```xml
<infinispan>
  <cache-container>
    <distributed-cache-configuration name="base-template">
      <encoding media-type="application/x-protostream"/>
      <expiration lifespan="5000"
                  max-idle="1000"/>
    </distributed-cache-configuration>
    <distributed-cache-configuration name="extended-template"
                                     configuration="base-template">
      <expiration lifespan="10000"/>
    </distributed-cache-configuration>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "distributed-cache-configuration" : {
        "name" : "base-template",
        "encoding": {
          "media-type": "application/x-protostream"
        },
        "expiration" : {
          "lifespan" : "5000",
          "max-idle" : "1000"
        }
      },
      "distributed-cache-configuration" : {
        "name" : "extended-template",
        "expiration" : {
          "lifespan" : "10000"
        }
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    - distributedCacheConfiguration:
      name: "base-template"
      encoding:
        mediaType: "application/x-protostream"
      expiration:
        lifespan: "5000"
        maxIdle: "1000"
    - distributedCacheConfiguration:
      name: "extended-template"
      expiration:
        lifespan: "10000"
```

3.2.3. Cache template wildcards

You can add wildcards to cache configuration template names. If you then create caches where the name matches the wildcard, Infinispan applies the configuration template.

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

Template wildcard example

XML

```xml
<infinispan>
   <cache-container>
      <distributed-cache-configuration name="async-dist-cache-*"
                                       mode="ASYNC"
                                       statistics="true">
        <encoding media-type="application/x-protostream"/>
      </distributed-cache-configuration>
   </cache-container>
</infinispan>
```

JSON

```json
{
  "infinispan" : {
    "cache-container" : {
      "distributed-cache-configuration" : {
        "name" : "async-dist-cache-*",
        "mode": "ASYNC",
        "statistics": "true",
        "encoding": {
          "media-type": "application/x-protostream"
        }
      }
    }
  }
}
```

YAML

```yaml
infinispan:
  cacheContainer:
    distributedCacheConfiguration:
      name: "async-dist-cache-*"
      mode: "ASYNC"
      statistics: "true"
      encoding:
        mediaType: "application/x-protostream"
```

Using the preceding example, if you create a cache named "async-dist-cache-prod" then Infinispan uses the configuration from the `async-dist-cache-*` template.

3.2.4. Cache templates from multiple XML files

Split cache configuration templates into multiple XML files for granular flexibility and reference them with XML inclusions (XInclude).

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

You must also add the `xmlns:xi="http://www.w3.org/2001/XInclude"` namespace to `infinispan.xml` to use XInclude.

Xinclude cache template

```
<infinispan xmlns:xi="http://www.w3.org/2001/XInclude">
  <cache-container default-cache="cache-1">
    <!-- References files that contain cache configuration templates. -->
    <xi:include href="distributed-cache-template.xml" />
    <xi:include href="replicated-cache-template.xml" />
  </cache-container>
</infinispan>
```

Infinispan also provides an `infinispan-config-fragment-13.0.xsd` schema that you can use with configuration fragments.

Configuration fragment schema

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

Additional resources

- XInclude specification

3.3. Creating remote caches

When you create remote caches at runtime, Infinispan Server synchronizes your configuration across the cluster so that all nodes have a copy. For this reason you should always create remote caches dynamically with the following mechanisms:

- Infinispan Console
- Infinispan Command Line Interface (CLI)
- Hot Rod or HTTP clients

3.3.1. Default cache manager

Infinispan Server provides a default cache manager to control the lifecycle of remote caches. Starting Infinispan Server automatically instantiates the cache manager so you can create and delete remote caches and other resources like Protobuf schema.

Default cache manager

```
<infinispan>
  <!-- Creates a cache manager named "default" that exports metrics. -->
  <cache-container name="default"
                   statistics="true">
    <!-- Adds cluster transport that uses the default JGroups TCP stack. -->
    <transport cluster="${infinispan.cluster.name:cluster}"
               stack="${infinispan.cluster.stack:tcp}"
                 node-name="${infinispan.node.name:}"/>
  </cache-container>
</infinispan>
```

After you start Infinispan Server and add user credentials, you can view details about the cache manager and get cluster information from Infinispan Console.

- Open `127.0.0.1:11222` in any browser.

You can also get information about the cache manager through the Command Line Interface (CLI) or REST API:

CLI

Run the `describe` command in the default container.

```
[//containers/default]> describe
```

REST

Open `127.0.0.1:11222/rest/v2/cache-managers/default/` in any browser.

3.3.2. Creating caches with Infinispan Console

Use Infinispan Console to create remote caches in an intuitive visual interface from any web browser.

Prerequisites

- Create a Infinispan user with `admin` permissions.
- Start at least one Infinispan Server instance.
- Have a Infinispan cache configuration.

Procedure

1. Open `127.0.0.1:11222/console/` in any browser.
2. Select **Create Cache** and follow the steps as Infinispan Console guides you through the process.

3.3.3. Creating remote caches with the Infinispan CLI

Use the Infinispan Command Line Interface (CLI) to add remote caches on Infinispan Server.

Prerequisites

- Create a Infinispan user with `admin` permissions.
- Start at least one Infinispan Server instance.
- Have a Infinispan cache configuration.

Procedure

1. Start the CLI and enter your credentials when prompted.

```
$ bin/cli.sh
```

2. Use the `create cache` command to create remote caches.

   For example, create a cache named "mycache" from a file named `mycache.xml` as follows:

```
[//containers/default]> create cache --file=mycache.xml mycache
```

Verification

1. List all remote caches with the `ls` command.

```
[//containers/default]> ls caches
mycache
```

2. View cache configuration with the `describe` command.

```
[//containers/default]> describe caches/mycache
```

3.3.4. Creating remote caches from Hot Rod clients

Use the Infinispan Hot Rod API to create remote caches on Infinispan Server from Java, C++, .NET/C#, JS clients and more.

This procedure shows you how to use Hot Rod Java clients that create remote caches on first access. You can find code examples for other Hot Rod clients in the Infinispan Tutorials.

Prerequisites

- Create a Infinispan user with `admin` permissions.
- Start at least one Infinispan Server instance.
- Have a Infinispan cache configuration.

Procedure

- Invoke the `remoteCache()` method as part of your the `ConfigurationBuilder`.
- Set the `configuration` or `configuration_uri` properties in the `hotrod-client.properties` file on

your classpath.

ConfigurationBuilder

```
File file = new File("path/to/infinispan.xml")
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.remoteCache("another-cache")
       .configuration("<distributed-cache name=\"another-cache\"/>");
builder.remoteCache("my.other.cache")
       .configurationURI(file.toURI());
```

hotrod-client.properties

```
infinispan.client.hotrod.cache.another-cache.configuration=<distributed-cache
name=\"another-cache\"/>
infinispan.client.hotrod.cache.[my.other.cache].configuration_uri=file:///path/to/infi
nispan.xml
```

If the name of your remote cache contains the `.` character, you must enclose it in square brackets when using `hotrod-client.properties` files.

Additional resources

- Hot Rod Client Configuration
- org.infinispan.client.hotrod.configuration.RemoteCacheConfigurationBuilder

3.3.5. Creating remote caches with the REST API

Use the Infinispan REST API to create remote caches on Infinispan Server from any suitable HTTP client.

Prerequisites

- Create a Infinispan user with `admin` permissions.
- Start at least one Infinispan Server instance.
- Have a Infinispan cache configuration.

Procedure

- Invoke `POST` requests to `/rest/v2/caches/<cache_name>` with cache configuration in the payload.

Additional resources

- Creating and Managing Caches with the REST API

3.4. Creating embedded caches

Infinispan provides an `EmbeddedCacheManager` API that lets you control both the Cache Manager and embedded cache lifecycles programmatically.

3.4.1. Adding Infinispan to your project

Add Infinispan to your project to create embedded caches in your applications.

Prerequisites

- Configure your project to get Infinispan artifacts from the Maven repository.

Procedure

- Add the `infinispan-core` artifact as a dependency in your `pom.xml` as follows:

```
<dependencies>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-core</artifactId>
  </dependency>
</dependencies>
```

3.4.2. Configuring embedded caches

Infinispan provides a `GlobalConfigurationBuilder` API that controls the cache manager and a `ConfigurationBuilder` API that configures embedded caches.

Prerequisites

- Add the `infinispan-core` artifact as a dependency in your `pom.xml`.

Procedure

1. Initialize the default cache manager so you can add embedded caches.
2. Add at least one embedded cache with the `ConfigurationBuilder` API.
3. Invoke the `getOrCreateCache()` method that either creates embedded caches on all nodes in the cluster or returns caches that already exist.

```
// Set up a clustered cache manager.
GlobalConfigurationBuilder global = GlobalConfigurationBuilder.
defaultClusteredBuilder();
// Initialize the default cache manager.
DefaultCacheManager cacheManager = new DefaultCacheManager(global.build());
// Create a distributed cache with synchronous replication.
ConfigurationBuilder builder = new ConfigurationBuilder();
                    builder.clustering().cacheMode(CacheMode.DIST_SYNC);
// Obtain a volatile cache.
Cache<String, String> cache = cacheManager.administration().withFlags
(CacheContainerAdmin.AdminFlag.VOLATILE).getOrCreateCache("myCache", builder.build());
```

Additional resources

- EmbeddedCacheManager
- EmbeddedCacheManager Configuration

- org.infinispan.configuration.global.GlobalConfiguration
- org.infinispan.configuration.cache.ConfigurationBuilder

Chapter 4. Enabling and configuring Infinispan statistics and JMX monitoring

Infinispan can provide Cache Manager and cache statistics as well as export JMX MBeans.

4.1. Enabling statistics in embedded caches

Configure Infinispan to export statistics for the cache manager and embedded caches.

Procedure

1. Open your Infinispan configuration for editing.
2. Add the `statistics="true"` attribute or the `.statistics(true)` method.
3. Save and close your Infinispan configuration.

Embedded cache statistics

XML

```
<infinispan>
  <cache-container statistics="true">
    <distributed-cache statistics="true"/>
    <replicated-cache statistics="true"/>
  </cache-container>
</infinispan>
```

GlobalConfigurationBuilder

```
GlobalConfigurationBuilder global = GlobalConfigurationBuilder.
defaultClusteredBuilder().cacheContainer().statistics(true);
DefaultCacheManager cacheManager = new DefaultCacheManager(global.build());

Configuration builder = new ConfigurationBuilder();
builder.statistics().enable();
```

4.2. Enabling statistics in remote caches

Infinispan Server automatically enables statistics for the default cache manager. However, you must explicitly enable statistics for your caches.

Procedure

1. Open your Infinispan configuration for editing.
2. Add the `statistics` attribute or field and specify `true` as the value.
3. Save and close your Infinispan configuration.

Remote cache statistics

XML

```
<distributed-cache statistics="true" />
```

JSON

```
{
  "distributed-cache": {
    "statistics": "true"
  }
}
```

YAML

```
distributed-cache:
  statistics: true
```

4.3. Enabling Hot Rod client statistics

Hot Rod Java clients can provide statistics that include remote cache and near-cache hits and misses as well as connection pool usage.

Procedure

1. Open your Hot Rod Java client configuration for editing.
2. Set `true` as the value for the `statistics` property or invoke the `statistics().enable()` methods.
3. Export JMX MBeans for your Hot Rod client with the `jmx` and `jmx_domain` properties or invoke the `jmxEnable()` and `jmxDomain()` methods.
4. Save and close your client configuration.

Hot Rod Java client statistics

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.statistics().enable()
         .jmxEnable()
         .jmxDomain("my.domain.org")
       .addServer()
         .host("127.0.0.1")
         .port(11222);
RemoteCacheManager remoteCacheManager = new RemoteCacheManager(builder.build());
```

hotrod-client.properties

```
infinispan.client.hotrod.statistics = true
infinispan.client.hotrod.jmx = true
infinispan.client.hotrod.jmx_domain = my.domain.org
```

4.4. Configuring Infinispan metrics

Infinispan generates metrics that are compatible with the MicroProfile Metrics API.

- Gauges provide values such as the average number of nanoseconds for write operations or JVM uptime.
- Histograms provide details about operation execution times such as read, write, and remove times.

By default, Infinispan generates gauges when you enable statistics but you can also configure it to generate histograms.

Procedure

1. Open your Infinispan configuration for editing.
2. Add the `metrics` element or object to the cache container.
3. Enable or disable gauges with the `gauges` attribute or field.
4. Enable or disable histograms with the `histograms` attribute or field.
5. Save and close your client configuration.

Metrics configuration

XML

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

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "statistics" : "true",
      "metrics" : {
        "gauges" : "true",
        "histograms" : "true"
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    statistics: "true"
    metrics:
      gauges: "true"
      histograms: "true"
```

GlobalConfigurationBuilder

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder()
  //Computes and collects statistics for the Cache Manager.
  .statistics().enable()
  //Exports collected statistics as gauge and histogram metrics.
  .metrics().gauges(true).histograms(true)
  .build();
```

Verification

Infinispan Server exposes statistics through the `metrics` endpoint. You can collect metrics with any monitoring tool that supports the OpenMetrics format, such as Prometheus.

Infinispan metrics are provided at the `vendor` scope. Metrics related to the JVM are provided in the `base` scope.

You can retrieve metrics from Infinispan Server as follows:

```
$ curl -v http://localhost:11222/metrics
```

To retrieve metrics in MicroProfile JSON format, do the following:

```
$ curl --header "Accept: application/json" http://localhost:11222/metrics
```

For embedded caches, you must add the necessary MicroProfile API and provider JARs to your classpath to export Infinispan metrics.

Additional resources

- Eclipse MicroProfile Metrics

4.5. Registering JMX MBeans

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

Procedure

1. Open your Infinispan configuration for editing.
2. Add the `jmx` element or object to the cache container and specify `true` as the value for the `enabled` attribute or field.
3. Add the `domain` attribute or field and specify the domain where JMX MBeans are exposed, if required.
4. Save and close your client configuration.

JMX configuration

XML

```
<infinispan>
  <cache-container statistics="true">
    <jmx enabled="true"
         domain="example.com"/>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "statistics" : "true",
      "jmx" : {
        "enabled" : "true",
        "domain" : "example.com"
      }
    }
  }
}
```

YAML

```yaml
infinispan:
  cacheContainer:
    statistics: "true"
    jmx:
      enabled: "true"
      domain: "example.com"
```

GlobalConfigurationBuilder

```
GlobalConfiguration global = GlobalConfigurationBuilder.defaultClusteredBuilder()
   .jmx().enable()
   .domain("org.mydomain");
```

4.5.1. Enabling JMX remote ports

Provide unique remote JMX ports to expose Infinispan MBeans through connections in JMXServiceURL format.

Infinispan Server does not expose JMX remotely via the single port endpoint. If you want to remotely access Infinispan Server via JMX you must enable a remote port.

Procedure

- Pass the following system properties to Infinispan at startup:

```
-Dcom.sun.management.jmxremote
-Dcom.sun.management.jmxremote.port=9999
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false
```

4.5.2. Infinispan MBeans

Infinispan exposes JMX MBeans that represent manageable resources.

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

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

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

Additional resources

- Infinispan JMX Components

4.5.3. Registering MBeans in custom MBean servers

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

Prerequisites

- Create an implementation of `MBeanServerLookup` so that the `getMBeanServer()` method returns the custom MBeanServer instance.
- Configure Infinispan to register JMX MBeans.

Procedure

1. Open your Infinispan configuration for editing.
2. Add the `mbean-server-lookup` attribute or field to the JMX configuration for the cache manager.
3. Specify fully qualified name (FQN) of your `MBeanServerLookup` implementation.
4. Save and close your client configuration.

JMX MBean server lookup configuration

XML

```
<cache-container statistics="true">
  <jmx enabled="true"
       domain="example.com"
       mbean-server-lookup="com.example.MyMBeanServerLookup" />
</cache-container>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "statistics" : "true",
      "jmx" : {
        "enabled" : "true",
        "domain" : "example.com",
        "mbean-server-lookup" : "com.example.MyMBeanServerLookup"
      }
    }
  }
}
```

YAML

```yaml
infinispan:
  cacheContainer:
    statistics: "true"
    jmx:
      enabled: "true"
      domain: "example.com"
      mbeanServerLookup: "com.example.MyMBeanServerLookup"
```

GlobalConfigurationBuilder

```
GlobalConfiguration global = GlobalConfigurationBuilder.defaultClusteredBuilder()
   .jmx().enable()
   .domain("org.mydomain")
   .mBeanServerLookup(new com.acme.MyMBeanServerLookup());
```

Chapter 5. Configuring JVM memory usage

Control how Infinispan stores data in JVM memory by:

- Managing JVM memory usage with eviction that automatically removes data from caches.
- Adding lifespan and maximum idle times to expire entries and prevent stale data.
- Configuring Infinispan to store data in off-heap, native memory.

5.1. Default memory configuration

By default Infinispan stores cache entries as objects in the JVM heap. Over time, as applications add entries, the size of caches can exceed the amount of memory that is available to the JVM. Likewise, if Infinispan is not the primary data store, then entries become out of date which means your caches contain stale data.

XML

```xml
<distributed-cache>
  <memory storage="HEAP"/>
</distributed-cache>
```

JSON

```json
{
  "distributed-cache": {
    "memory" : {
      "storage": "HEAP"
    }
  }
}
```

YAML

```yaml
distributedCache:
  memory:
    storage: "HEAP"
```

5.2. Eviction and expiration

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

- ☑ Eviction lets Infinispan control the size of the data container by removing entries when the container becomes larger than a configured threshold.
- ☑ Expiration limits the amount of time entries can exist. Infinispan uses a scheduler to periodically remove expired entries. Entries that are expired but not yet removed are

immediately removed on access; in this case `get()` calls for expired entries return "null" values.

- ☑ Eviction is local to Infinispan nodes.
- ☑ Expiration takes place across Infinispan clusters.
- ☑ You can use eviction and expiration together or independently of each other.
- ☑ You can configure eviction and expiration declaratively in `infinispan.xml` to apply cache-wide defaults for entries.
- ☑ You can explicitly define expiration settings for specific entries but you cannot define eviction on a per-entry basis.
- ☑ You can manually evict entries and manually trigger expiration.

5.3. Eviction with Infinispan caches

Eviction lets you control the size of the data container by removing entries from memory in one of two ways:

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

Eviction drops one entry from the data container at a time and is local to the node on which it occurs.

Eviction removes entries from memory but not from persistent cache stores. To ensure that entries remain available after Infinispan evicts them, and to prevent inconsistencies with your data, you should configure persistent storage.

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

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

5.3.1. Eviction strategies

When you configure Infinispan eviction you specify:

- The maximum size of the data container.
- A strategy for removing entries when the cache reaches the threshold.

You can either perform eviction manually or configure Infinispan to do one of the following:

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

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

Refer to the schema reference for more details about the eviction strategies.

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

Additional resources

- Caffeine
- Infinispan configuration schema reference

5.3.2. Configuring maximum count eviction

Limit the size of Infinispan caches to a total number of entries.

Procedure

1. Open your Infinispan configuration for editing.
2. Specify the total number of entries that caches can contain before Infinispan performs eviction with either the `max-count` attribute or `maxCount()` method.
3. Set one of the following as the eviction strategy to control how Infinispan removes entries with the `when-full` attribute or `whenFull()` method.
   - `REMOVE` Infinispan performs eviction. This is the default strategy.
   - `MANUAL` You perform eviction manually for embedded caches.
   - `EXCEPTION` Infinispan throws an exception instead of evicting entries.
4. Save and close your Infinispan configuration.

Maximum count eviction

In the following example, Infinispan removes an entry when the cache contains a total of 500 entries and a new entry is created:

XML

```
<distributed-cache>
  <memory max-count="500" when-full="REMOVE"/>
</distributed-cache>
```

JSON

```
{
  "distributed-cache" : {
    "memory" : {
      "max-count" : "500",
      "when-full" : "REMOVE"
    }
  }
}
```

YAML

```
distributedCache:
  memory:
    maxCount: "500"
    whenFull: "REMOVE"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.memory().maxCount(500).whenFull(EvictionStrategy.REMOVE);
```

Additional resources

- Infinispan configuration schema reference
- org.infinispan.configuration.cache.MemoryConfigurationBuilder

5.3.3. Configuring maximum size eviction

Limit the size of Infinispan caches to a maximum amount of memory.

Procedure

1. Open your Infinispan configuration for editing.
2. Specify `application/x-protostream` as the media type for cache encoding.

   You must specify a binary media type to use maximum size eviction.

3. Configure the maximum amount of memory, in bytes, that caches can use before Infinispan performs eviction with the `max-size` attribute or `maxSize()` method.
4. Optionally specify a byte unit of measurement.

   The default is B (bytes). Refer to the configuration schema for supported units.

5. Set one of the following as the eviction strategy to control how Infinispan removes entries with either the `when-full` attribute or `whenFull()` method.
   - `REMOVE` Infinispan performs eviction. This is the default strategy.

- `MANUAL` You perform eviction manually for embedded caches.
- `EXCEPTION` Infinispan throws an exception instead of evicting entries.

6. Save and close your Infinispan configuration.

Maximum size eviction

In the following example, Infinispan removes an entry when the size of the cache reaches 1.5 GB (gigabytes) and a new entry is created:

XML

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

JSON

```
{
  "distributed-cache" : {
    "encoding" : {
      "media-type" : "application/x-protostream"
    },
    "memory" : {
      "max-size" : "1.5GB",
      "when-full" : "REMOVE"
    }
  }
}
```

YAML

```
distributedCache:
  encoding:
    mediaType: "application/x-protostream"
  memory:
    maxSize: "1.5GB"
    whenFull: "REMOVE"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.encoding().mediaType("application/x-protostream")
       .memory()
         .maxSize("1.5GB")
         .whenFull(EvictionStrategy.REMOVE);
```

Additional resources

- Infinispan configuration schema reference
- org.infinispan.configuration.cache.EncodingConfiguration
- org.infinispan.configuration.cache.MemoryConfigurationBuilder
- Cache Encoding and Marshalling

5.3.4. Manual eviction

If you choose the manual eviction strategy, Infinispan does not perform eviction. You must do so manually with the `evict()` method.

You should use manual eviction with embedded caches only. For remote caches, you should always configure Infinispan with the `REMOVE` or `EXCEPTION` eviction strategy.

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

XML

```xml
<distributed-cache>
  <memory max-count="500" when-full="MANUAL"/>
</distributed-cache>
```

JSON

```json
{
  "distributed-cache" : {
    "memory" : {
      "max-count" : "500",
      "when-full" : "MANUAL"
    }
  }
 }
```

YAML

```yaml
distributedCache:
  memory:
    maxCount: "500"
    whenFull: "MANUAL"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.encoding().mediaType("application/x-protostream")
       .memory()
         .maxSize("1.5GB")
         .whenFull(EvictionStrategy.REMOVE);
```

5.3.5. Passivation with eviction

Passivation persists data to cache stores when Infinispan evicts entries. You should always enable eviction if you enable passivation, as in the following examples:

XML

```
<distributed-cache>
  <persistence passivation="true">
    <!-- Persistent storage configuration. -->
  </persistence>
  <memory max-count="100"/>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "memory" : {
      "max-count" : "100"
    },
    "persistence" : {
      "passivation" : true
    }
  }
}
```

YAML

```
distributedCache:
  memory:
    maxCount: "100"
  persistence:
    passivation: "true"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.memory().maxCount(100);
builder.persistence().passivation(true); //Persistent storage configuration
```

5.4. Expiration with lifespan and maximum idle

Expiration configures Infinispan to remove entries from caches when they reach one of the following time limits:

Lifespan

Sets the maximum amount of time that entries can exist.

Maximum idle

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

Maximum idle expiration does not currently support caches with persistent storage.

If you use expiration and eviction with the `EXCEPTION` eviction strategy, entries that are expired, but not yet removed from the cache, count towards the size of the data container.

5.4.1. How expiration works

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

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

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

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

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

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

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

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

5.4.2. Expiration reaper

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

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

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

If you use clustered cache modes, you should never disable the expiration reaper.

Infinispan always uses the expiration reaper when using cache stores. In this case you cannot disable it.

Additional resources

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

5.4.3. Maximum idle and clustered caches

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

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

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

The touch commands that Infinispan send have the following considerations:

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

Additional information

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

5.4.4. Configuring lifespan and maximum idle times for caches

Set lifespan and maximum idle times for all entries in a cache.

Procedure

1. Open your Infinispan configuration for editing.
2. Specify the amount of time, in milliseconds, that entries can stay in the cache with the `lifespan` attribute or `lifespan()` method.
3. Specify the amount of time, in milliseconds, that entries can remain idle after last access with the `max-idle` attribute or `maxIdle()` method.
4. Save and close your Infinispan configuration.

Expiration for Infinispan caches

In the following example, Infinispan expires all cache entries after 5 seconds or 1 second after the last access time, whichever happens first:

XML

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

JSON

```
{
  "replicated-cache" : {
    "expiration" : {
      "lifespan" : "5000",
      "max-idle" : "1000"
    }
  }
}
```

YAML

```
replicatedCache:
  expiration:
    lifespan: "5000"
    maxIdle: "1000"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.expiration().lifespan(5000, TimeUnit.MILLISECONDS)
                    .maxIdle(1000, TimeUnit.MILLISECONDS);
```

5.4.5. Configuring lifespan and maximum idle times per entry

Specify lifespan and maximum idle times for individual entries. When you add lifespan and maximum idle times to entries, those values take priority over expiration configuration for caches.

When you explicitly define lifespan and maximum idle time values for cache entries, Infinispan replicates those values across the cluster along with the cache entries. Likewise, Infinispan writes expiration values along with the entries to persistent storage.

Procedure

- For remote caches, you can add lifespan and maximum idle times to entries interactively with the Infinispan Console.

  With the Infinispan Command Line Interface (CLI), use the `--max-idle=` and `--ttl=` arguments with the `put` command.

- For both remote and embedded caches, you can add lifespan and maximum idle times with `cache.put()` invocations.

```
//Lifespan of 5 seconds.
//Maximum idle time of 1 second.
cache.put("hello", "world", 5, TimeUnit.SECONDS, 1, TimeUnit.SECONDS);

//Lifespan is disabled with a value of -1.
//Maximum idle time of 1 second.
cache.put("hello", "world", -1, TimeUnit.SECONDS, 1, TimeUnit.SECONDS);
```

Additional resources

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

5.5. JVM heap and off-heap memory

Infinispan stores cache entries in JVM heap memory by default. You can configure Infinispan to use off-heap storage, which means that your data occupies native memory outside the managed JVM memory space.

The following diagram is a simplified illustration of the memory space for a JVM process where Infinispan is running:

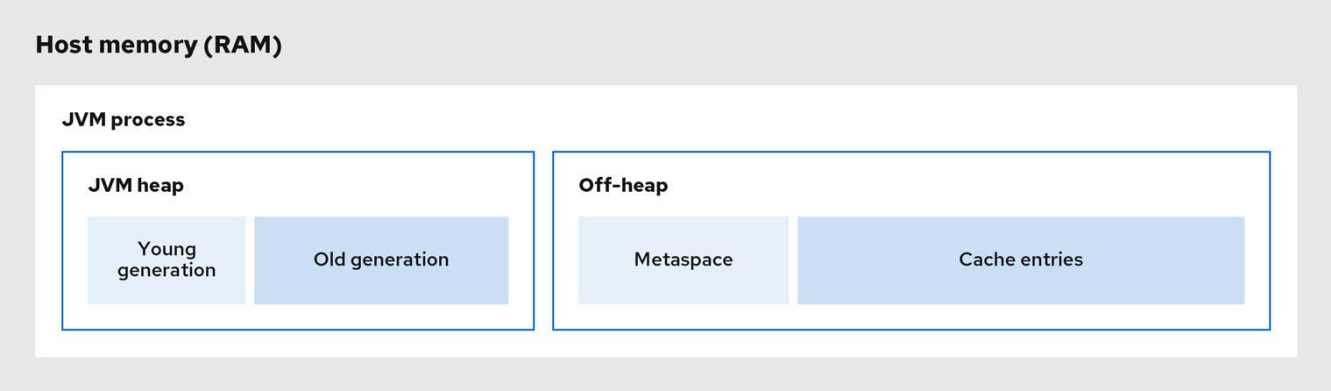

Figure 6. JVM memory space

JVM heap memory

The heap is divided into young and old generations that help keep referenced Java objects and other application data in memory. The GC process reclaims space from unreachable objects, running more frequently on the young generation memory pool.

When Infinispan stores cache entries in JVM heap memory, GC runs can take longer to complete as you start adding data to your caches. Because GC is an intensive process, longer and more frequent runs can degrade application performance.

Off-heap memory

Off-heap memory is native available system memory outside JVM memory management. The *JVM memory space* diagram shows the `Metaspace` memory pool that holds class metadata and is allocated from native memory. The diagram also represents a section of native memory that holds Infinispan cache entries.

Off-heap memory:

- Uses less memory per entry.
- Improves overall JVM performance by avoiding Garbage Collector (GC) runs.

One disadvantage, however, is that JVM heap dumps do not show entries stored in off-heap memory.

5.5.1. Off-heap data storage

When you add entries to off-heap caches, Infinispan dynamically allocates native memory to your data.

Infinispan hashes the serialized `byte[]` for each key into buckets that are similar to a standard Java `HashMap`. Buckets include address pointers that Infinispan uses to locate entries that you store in off-heap memory.

Even though Infinispan stores cache entries in native memory, run-time operations require JVM heap representations of those objects. For instance, `cache.get()` operations read objects into heap memory before returning. Likewise, state transfer operations hold subsets of objects in heap memory while they take place.

Object equality

Infinispan determines equality of Java objects in off-heap storage using the serialized byte[] representation of each object instead of the object instance.

Data consistency

Infinispan uses an array of locks to protect off-heap address spaces. The number of locks is twice the number of cores and then rounded to the nearest power of two. This ensures that there is an even distribution of `ReadWriteLock` instances to prevent write operations from blocking read operations.

5.5.2. Configuring off-heap memory

Configure Infinispan to store cache entries in native memory outside the JVM heap space.

Procedure

1. Open your Infinispan configuration for editing.
2. Set `OFF_HEAP` as the value for the `storage` attribute or `storage()` method.
3. Set a boundary for the size of the cache by configuring eviction.
4. Save and close your Infinispan configuration.

Off-heap storage

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

XML

```
<replicated-cache>
  <memory storage="OFF_HEAP" max-count="500"/>
</replicated-cache>
```

JSON

```
{
  "replicated-cache" : {
    "memory" : {
      "storage" : "OBJECT",
      "max-count" : "500"
    }
  }
}
```

YAML

```
replicatedCache:
  memory:
    storage: "OFF_HEAP"
    maxCount: "500"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.memory().storage(StorageType.OFF_HEAP).maxCount(500);
```

Additional resources

- Infinispan configuration schema reference
- org.infinispan.configuration.cache.MemoryConfigurationBuilder

Chapter 6. Configuring persistent storage

Infinispan uses cache stores and loaders to interact with persistent storage.

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.

6.1. Passivation

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

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

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

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

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

6.1.1. How passivation works

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

6.2. Write-through cache stores

Write-through is a cache writing mode where writes to memory and writes to cache stores are synchronous. When a client application updates a cache entry, in most cases by invoking

`Cache.put()`, Infinispan does not return the call until it updates the cache store. This cache writing mode results in updates to the cache store concluding within the boundaries of the client thread.

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

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

Write-through configuration

Infinispan uses write-through mode unless you explicitly add write-behind configuration to your caches. There is no separate element or method for configuring write-through mode.

For example, the following configuration adds a file-based store to the cache that implicitly uses write-through mode:

```
<distributed-cache>
  <persistence passivation="false">
    <file-store fetch-state="true">
      <index path="path/to/index" />
      <data path="path/to/data" />
    </file-store>
  </persistence>
</distributed-cache>
```

6.3. Write-behind cache stores

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

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

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

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

Write-behind mode provides a performance advantage over write-through mode because cache operations do not need to wait for updates to the underlying cache store to complete. However, data in the cache store remains inconsistent with data in the cache until the modification queue is processed. For this reason, write-behind mode is suitable for cache stores with low latency, such as

unshared and local file-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

XML

```xml
<distributed-cache>
  <persistence>
    <table-jdbc-store xmlns="urn:infinispan:config:store:sql:13.0"
                      dialect="H2"
                      shared="true"
                      table-name="books">
      <connection-pool connection-url="jdbc:h2:mem:infinispan"
                       username="sa"
                       password="changeme"
                       driver="org.h2.Driver"/>
      <write-behind modification-queue-size="2048"
                    fail-silently="true"/>
    </table-jdbc-store>
  </persistence>
</distributed-cache>
```

JSON

```json
{
  "distributed-cache": {
    "persistence" : {
      "table-jdbc-store": {
        "dialect": "H2",
        "shared": "true",
        "table-name": "books",
        "connection-pool": {
          "connection-url": "jdbc:h2:mem:infinispan",
          "driver": "org.h2.Driver",
          "username": "sa",
          "password": "changeme"
        },
        "write-behind" : {
          "modification-queue-size" : "2048",
          "fail-silently" : true
        }
      }
    }
  }
}
```

YAML

```yaml
distributed-cache:
  persistence:
    table-jdbc-store:
      dialect: "H2"
      shared: "true"
      table-name: "books"
      connection-pool:
        connection-url: "jdbc:h2:mem:infinispan"
        driver: "org.h2.Driver"
        username: "sa"
        password: "changeme"
      writeBehind:
        modificationQueueSize: "2048"
        failSilently: "true"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
       .async()
       .modificationQueueSize(2048)
       .failSilently(true);
```

Failing silently

Write-behind configuration includes a `fail-silently` parameter that controls what happens when either the cache store is unavailable or the modification queue is full.

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

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

6.4. 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.

The number of segments you configure for cache stores must match the number of segments you define in the Infinispan configuration with the `clustering.hash.numSegments` parameter.

If you change the `numSegments` parameter in the configuration after you add a segmented cache store, Infinispan cannot read data from that cache store.

6.5. Shared cache stores

Infinispan cache stores can be local to a given node or shared across all nodes in the cluster. By default, cache stores are local (`shared="false"`).

- Local cache stores are unique to each node; for example, a file-based cache store that persists data to the host filesystem.

  Local cache stores can fetch state and purge on startup to avoid loading stale entries from persistent storage.

- Shared cache stores allow multiple nodes to use the same persistent storage; for example, a JDBC cache store that allows multiple nodes to access the same database.

  Shared cache stores ensure that only the primary owner write to persistent storage, instead of backup nodes performing write operations for every modification.

Never configure shared cache stores to fetch state and purge on startup. Fetching state with shared cache stores results in performance issues and longer cluster start times. Purging deletes data, which is not typically the desired behavior for persistent storage.

Local cache store

```
<persistence>
  <store shared="false"
         fetch-state="true"
         purge="true"/>
</persistence>
```

Shared cache store

```
<persistence>
  <store shared="true"
         fetch-state="false"
         purge="false"/>
</persistence>
```

Additional resources

- Infinispan Configuration Schema

6.6. Transactions with persistent cache stores

Infinispan supports transactional operations with JDBC-based cache stores only. To configure caches as transactional, you 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 these cases manual recovery is not possible with cache stores.

6.7. Global persistent location

Infinispan preserves global state so that it can restore cluster topology and cached data after restart.

Remote caches

Infinispan Server saves cluster state to the `$ISPN_HOME/server/data` directory.

You should never delete or modify the `server/data` directory or its content. Infinispan restores cluster state from this directory when you restart your server instances.

Changing the default configuration or directly modifying the `server/data` directory can cause unexpected behavior and lead to data loss.

Embedded caches

Infinispan defaults to the `user.dir` system property as the global persistent location. In most cases this is the directory where your application starts.

For clustered embedded caches, such as replicated or distributed, you should always enable and configure a global persistent location to restore cluster topology.

You should never configure an absolute path for a file-based cache store that is outside the global persistent location. If you do, Infinispan writes the following exception to logs:

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

6.7.1. Configuring the global persistent location

Enable and configure the location where Infinispan stores global state for clustered embedded caches.

Infinispan Server enables global persistence and configures a default location. You should not disable global persistence or change the default configuration for remote caches.

Prerequisites

- Add Infinispan to your project.

Procedure

1. Enable global state in one of the following ways:
   - Add the `global-state` element to your Infinispan configuration.
   - Call the `globalState().enable()` methods in the `GlobalConfigurationBuilder` API.
2. Define whether the global persistent location is unique to each node or shared between the cluster.

Location type	Configuration
Unique to each node	`persistent-location` element or `persistentLocation()` method
Shared between the cluster	`shared-persistent-location` element or `sharedPersistentLocation(String)` method

3. Set the path where Infinispan stores cluster state.

   For example, file-based cache stores the path is a directory on the host filesystem.

   Values can be:

   - Absolute and contain the full location including the root.
   - Relative to a root location.
4. If you specify a relative value for the path, you must also specify a system property that resolves to a root location.

   For example, on a Linux host system you set `global/state` as the path. You also set the `my.data` property that resolves to the `/opt/data` root location. In this case Infinispan uses `/opt/data/global/state` as the global persistent location.

Global persistent location configuration

XML

```xml
<infinispan>
  <cache-container>
    <global-state>
      <persistent-location path="global/state" relative-to="my.data"/>
    </global-state>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "global-state": {
        "persistent-location" : {
          "path" : "global/state",
          "relative-to" : "my.data"
        }
      }
    }
  }
}
```

YAML

```
cacheContainer:
  globalState:
      persistentLocation:
        path: "global/state"
        relativeTo : "my.data"
```

GlobalConfigurationBuilder

```
new GlobalConfigurationBuilder().globalState()
                                .enable()
                                .persistentLocation("global/state", "my.data");
```

Additional resources

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

6.8. File-based cache stores

File-based cache stores provide persistent storage on the local host filesystem where Infinispan is running. For clustered caches, file-based cache stores are unique to each Infinispan node.

Never use filesystem-based cache stores on shared file systems, such as an NFS or Samba share, because they do not provide file locking capabilities and data corruption can occur.

Additionally if you attempt to use transactional caches with shared file systems, unrecoverable failures can happen when writing to files during the commit phase.

Soft-Index File Stores

`SoftIndexFileStore` is the default implementation for file-based cache stores and stores data in a set of append-only files.

When append-only files:

- Reach their maximum size, Infinispan creates a new file and starts writing to it.
- Reach the compaction threshold of less than 50% usage, Infinispan overwrites the entries to a new file and then deletes the old file.

B+ trees

To improve performance, append-only files in a `SoftIndexFileStore` are indexed using a **B+ Tree** that can be stored both on disk and in memory. The in-memory index uses Java soft references to ensure it can be rebuilt if removed by Garbage Collection (GC) then requested again.

Because `SoftIndexFileStore` uses Java soft references to keep indexes in memory, it helps prevent out-of-memory exceptions. GC removes indexes before they consume too much memory while still falling back to disk.

You can configure any number of B+ trees with the `segments` attribute on the `index` element declaratively or with the `indexSegments()` method programmatically. By default Infinispan creates up to 16 B+ trees, which means there can be up to 16 indexes. Having multiple indexes prevents bottlenecks from concurrent writes to an index and reduces the number of entries that Infinispan needs to keep in memory. As it iterates over a soft-index file store, Infinispan reads all entries in an index at the same time.

Each entry in the B+ tree is a node. By default, the size of each node is limited to 4096 bytes. `SoftIndexFileStore` throws an exception if keys are longer after serialization occurs.

Segmentation

Soft-index file stores are always segmented.

The `AdvancedStore.purgeExpired()` method is not implemented in `SoftIndexFileStore`.

Single File Cache Stores

Single file cache stores are now deprecated and planned for removal.

Single File cache stores, `SingleFileStore`, persist data to file. Infinispan also maintains an in-

memory index of keys while keys and values are stored in the file.

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 are larger or there can be a larger number of them.

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.

Segmentation

Single file cache stores are segmented by default with a separate instance per segment, which results in multiple directories. Each directory is a number that represents the segment to which the data maps.

6.8.1. Configuring file-based cache stores

Add file-based cache stores to Infinispan to persist data on the host filesystem.

Prerequisites

- Enable global state and configure a global persistent location if you are configuring embedded caches.

Procedure

1. Add the `persistence` element to your cache configuration.
2. Optionally specify `true` as the value for the `passivation` attribute to write to the file-based cache store only when data is evicted from memory.
3. Include the `file-store` element and configure attributes as appropriate.
4. Specify `false` as the value for the `shared` attribute.

   File-based cache stores should always be unique to each Infinispan instance. If you want to use the same persistent across a cluster, configure shared storage such as a JDBC string-based cache store .

5. Configure the `index` and `data` elements to specify the location where Infinispan creates indexes and stores data.
6. Include the `write-behind` element if you want to configure the cache store with write-behind mode.

File-based cache store configuration

XML

```xml
<distributed-cache>
  <persistence passivation="true">
     <file-store shared="false">
        <data path="data"/>
        <index path="index"/>
        <write-behind modification-queue-size="2048" />
     </file-store>
  </persistence>
</distributed-cache>
```

JSON

```json
{
  "distributed-cache": {
    "persistence": {
      "passivation": true,
      "file-store" : {
        "shared": false,
        "data": {
          "path": "data"
        },
        "index": {
          "path": "index"
        },
        "write-behind": {
          "modification-queue-size": "2048"
        }
      }
    }
  }
}
```

YAML

```yaml
distributed-cache:
  persistence:
    passivation: "true"
    fileStore:
      shared: "false"
      data: "data"
      index: "index"
      writeBehind:
        modificationQueueSize: "2048"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().passivation(true)
       .addSoftIndexFileStore()
          .shared(false)
          .dataLocation("data")
          .indexLocation("index")
          .modificationQueueSize(2048);
```

6.8.2. Configuring single file cache stores

If required, you can configure Infinispan to create single file stores.

Single file stores are deprecated. You should use soft-index file stores for better performance and data consistency in comparison with single file stores.

Prerequisites

- Enable global state and configure a global persistent location if you are configuring embedded caches.

Procedure

1. Add the `persistence` element to your cache configuration.
2. Optionally specify `true` as the value for the `passivation` attribute to write to the file-based cache store only when data is evicted from memory.
3. Include the `single-file-store` element.
4. Specify `false` as the value for the `shared` attribute.
5. Configure any other attributes as appropriate.
6. Include the `write-behind` element to configure the cache store as write behind instead of as write through.

Single file cache store configuration

XML

```
<distributed-cache>
  <persistence passivation="true">
    <single-file-store shared="false"
                       preload="true"
                       fetch-state="true"/>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence" : {
      "passivation" : true,
      "single-file-store" : {
        "shared" : false,
        "preload" : true,
        "fetch-state" : true
      }
    }
  }
}
```

YAML

```
distributed-cache:
  persistence:
    passivation: "true"
    singleFileStore:
      shared: "false"
      preload: "true"
      fetchState: "true"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().passivation(true)
       .addStore(SingleFileStoreConfigurationBuilder.class)
          .shared(false)
          .preload(true)
          .fetchPersistentState(true);
```

6.9. JDBC connection factories

Infinispan provides different `ConnectionFactory` implementations that allow you to connect to databases. You use JDBC connections with SQL cache stores and JDBC string-based caches stores.

Connection pools

Connection pools are suitable for standalone Infinispan deployments and are based on Agroal.

XML

```
<distributed-cache>
  <persistence>
     <connection-pool connection-url="jdbc:h2:mem:infinispan;DB_CLOSE_DELAY=-1"
                      username="sa"
                      password="changeme"
                      driver="org.h2.Driver"/>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "connection-pool": {
        "connection-url": "jdbc:h2:mem:infinispan_string_based",
        "driver": "org.h2.Driver",
        "username": "sa",
        "password": "changeme"
      }
    }
  }
}
```

YAML

```
distributed-cache:
  persistence:
    connectionPool:
      connectionUrl: "jdbc:h2:mem:infinispan_string_based;DB_CLOSE_DELAY=-1"
      driver: org.h2.Driver
      username: sa
      password: changeme
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
       .connectionPool()
         .connectionUrl("jdbc:h2:mem:infinispan_string_based;DB_CLOSE_DELAY=-1")
         .username("sa")
         .driverClass("org.h2.Driver");
```

Managed datasources

Datasource connections are suitable for managed environments such as application servers.

XML

```
<distributed-cache>
  <persistence>
    <data-source jndi-url="java:/StringStoreWithManagedConnectionTest/DS" />
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "data-source": {
        "jndi-url": "java:/StringStoreWithManagedConnectionTest/DS"
      }
    }
  }
}
```

YAML

```
distributed-cache:
  persistence:
    dataSource:
      jndiUrl: "java:/StringStoreWithManagedConnectionTest/DS"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
       .dataSource()
         .jndiUrl("java:/StringStoreWithManagedConnectionTest/DS");
```

Simple connections

Simple connection factories create database connections on a per invocation basis and are intended for use with test or development environments only.

XML

```
<distributed-cache>
  <persistence>
    <simple-connection connection-url="jdbc:h2://localhost"
                       username="sa"
                       password="changeme"
                       driver="org.h2.Driver"/>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "simple-connection": {
        "connection-url": "jdbc:h2://localhost",
        "driver": "org.h2.Driver",
        "username": "sa",
        "password": "changeme"
      }
    }
  }
}
```

YAML

```
distributed-cache:
  persistence:
    simpleConnection:
      connectionUrl: "jdbc:h2://localhost"
      driver: org.h2.Driver
      username: sa
      password: changeme
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
       .simpleConnection()
         .connectionUrl("jdbc:h2://localhost")
         .driverClass("org.h2.Driver")
         .username("admin")
         .password("changeme");
```

Additional resources

- PooledConnectionFactoryConfigurationBuilder

- ManagedConnectionFactoryConfigurationBuilder
- SimpleConnectionFactoryConfigurationBuilder

6.9.1. Configuring Infinispan Server with managed datasources

Create managed datasources as part of your Infinispan Server configuration to optimize connection pooling and performance for JDBC database connections. You can then use these datasources as part of your JDBC cache store configuration.

Prerequisites

- Copy database drivers to the `server/lib` directory in your Infinispan Server installation.

Use the `install` command with the Infinispan Command Line Interface (CLI) to download the required drivers to the `server/lib` directory, for example:

```
install org.postgresql:postgresql:42.1.3
```

Procedure

1. Open your Infinispan Server configuration for editing.
2. Add a new `data-source` to the `data-sources` section.
3. Uniquely identify the datasource with the `name` attribute or field.
4. Specify a JNDI name for the datasource with the `jndi-name` attribute or field.

You use the JNDI name to specify the datasource in your JDBC cache store configuration.

5. Set `true` as the value of the `statistics` attribute or field to enable statistics for the datasource through the `/metrics` endpoint.
6. Provide JDBC driver details that define how to connect to the datasource in the `connection-factory` section.
   a. Specify the name of the database driver with the `driver` attribute or field.
   b. Specify the JDBC connection url with the `url` attribute or field.
   c. Specify credentials with the `username` and `password` attributes or fields.
   d. Provide any other configuration as appropriate.
7. Define how Infinispan Server nodes pool and reuse connections with connection pool tuning properties in the `connection-pool` section.

Verification

Use the Infinispan Command Line Interface (CLI) to test the datasource connection, as follows:

1. Start a CLI session.

```
$ bin/cli.sh
```

2. List all datasources and confirm the one you created is available.

```
[//containers/default]> server datasource ls
```

3. Test a datasource connection.

```
[//containers/default]> server datasource test my-datasource
```

Next steps

Now that you have created a managed datasource in your Infinispan Server configuration, you can reference the JNDI name in cache store configuration as in the following example:

```
<distributed-cache-configuration name="persistent-cache" xmlns:jdbc=
"urn:infinispan:config:store:jdbc:13.0">
    <persistence>
        <jdbc:string-keyed-jdbc-store>
            <!-- Specifies the JNDI name for the datasource connection
                 in Infinispan Server configuration. -->
            <jdbc:data-source jndi-url="jdbc/postgres"/>
            <jdbc:string-keyed-table drop-on-exit="true"
                         create-on-start="true"
                         prefix="TBL">
                <jdbc:id-column name="ID" type="VARCHAR(255)"/>
                <jdbc:data-column name="DATA" type="BYTEA"/>
                <jdbc:timestamp-column name="TS" type="BIGINT"/>
                <jdbc:segment-column name="S" type="INT"/>
            </jdbc:string-keyed-table>
        </jdbc:string-keyed-jdbc-store>
    </persistence>
</distributed-cache-configuration>
```

Managed datasources for JDBC connections

Managed datasources centralize JDBC connection configuration for your Infinispan cluster and share connection pools for Infinispan Server nodes.

XML

```xml
<data-sources>
   <!-- Defines a unique name for the datasource and JNDI name that you
        reference in JDBC cache store configuration.
        Enables statistics for the datasource, if required. -->
   <data-source name="ds"
                jndi-name="jdbc/postgres"
                statistics="true">
      <!-- Specifies the JDBC driver that creates connections. -->
      <connection-factory driver="org.postgresql.Driver"
                          url="jdbc:postgresql://localhost:5432/postgres"
                          username="postgres"
                          password="changeme">
         <!-- Sets optional JDBC driver-specific connection properties. -->
         <connection-property name="name">value</connection-property>
      </connection-factory>
      <!-- Defines connection pool tuning properties. -->
      <connection-pool initial-size="1"
                       max-size="10"
                       min-size="3"
                       background-validation="1000"
                       idle-removal="1"
                       blocking-timeout="1000"
                       leak-detection="10000"/>
   </data-source>
</data-sources>
```

JSON

```
{
  "data-sources": [
    {
      "name": "ds",
      "jndi-name": "jdbc/postgres",
      "statistics": true,
      "connection-factory": {
        "driver": "org.postgresql.Driver",
        "url": "jdbc:postgresql://localhost:5432/postgres",
        "username": "postgres",
        "password": "changeme",
        "connection-properties": {
          "name": "value"
        }
      },
      "connection-pool": {
        "initial-size": 1,
        "max-size": 10,
        "min-size": 3,
        "background-validation": 1000,
        "idle-removal": 1,
        "blocking-timeout": 1000,
        "leak-detection": 10000
      }
    }
  ]
}
```

YAML

```yaml
data-sources:
  - name: ds
    jndi-name: 'jdbc/postgres'
    statistics: true
    connection-factory:
      driver: "org.postgresql.Driver"
      url: "jdbc:postgresql://localhost:5432/postgres"
      username: "postgres"
      password: "changeme"
      connection-properties:
        name: value
    connection-pool:
      initial-size: 1
      max-size: 10
      min-size: 3
      background-validation: 1000
      idle-removal: 1
      blocking-timeout: 1000
      leak-detection: 10000
```

Connection pool tuning properties

You can tune connection pools with the following parameters:

Parameter	Description
`initial-size`	Initial number of connections the pool should hold.
`max-size`	Maximum number of connections in the pool.
`min-size`	Minimum number of connections the pool should hold.
`blocking-timeout`	Maximum time in milliseconds to block while waiting for a connection before throwing an exception. This will never throw an exception if creating a new connection takes an inordinately long period of time. Default is `0` meaning that a call will wait indefinitely.
`background-validation`	Time in milliseconds between background validation runs. A duration of `0` means that this feature is disabled.
`validate-on-acquisition`	Connections idle for longer than this time, specified in milliseconds, are validated before being acquired (foreground validation). A duration of `0` means that this feature is disabled.
`idle-removal`	Time in minutes a connection has to be idle before it can be removed.
`leak-detection`	Time in milliseconds a connection has to be held before a leak warning.

6.9.2. Configuring JDBC connection pools with Agroal properties

You can use a properties file to configure pooled connection factories for JDBC string-based cache

stores.

Procedure

1. Specify JDBC connection pool configuration with `org.infinispan.agroal.*` properties, 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_DELA
Y=-1
org.infinispan.agroal.driverClassName=org.h2.Driver.class
org.infinispan.agroal.principal=sa
org.infinispan.agroal.credential=sa
```

2. Configure Infinispan to use your properties file with the `properties-file` attribute or the `PooledConnectionFactoryConfiguration.propertyFile()` method.

XML

```
<connection-pool properties-file="path/to/agroal.properties"/>
```

JSON

```
"persistence": {
  "connection-pool": {
    "properties-file": "path/to/agroal.properties"
  }
}
```

YAML

```
persistence:
  connectionPool:
    propertiesFile: path/to/agroal.properties
```

ConfigurationBuilder

```
.connectionPool().propertyFile("path/to/agroal.properties")
```

Additional resources

- Agroal

6.10. SQL cache stores

SQL cache stores let you load Infinispan caches from existing database tables. Infinispan offers two types of SQL cache store:

Table
: Infinispan loads entries from a single database table.

Query
: Infinispan uses SQL queries to load entries from single or multiple database tables, including from sub-columns within those tables, and perform insert, update, and delete operations.

Both SQL table and query stores:

- Allow read and write operations to persistent storage.
- Can be read-only and act as a cache loader.
- Support keys and values that correspond to a single database column or a composite of multiple database columns.

  For composite keys and values, you must provide Infinispan with Protobuf schema (`.proto` files) that describe the keys and values. With Infinispan Server you can add schema through the Infinispan Console or Command Line Interface (CLI) with the `schema` command.

SQL cache stores do not support expiration or segmentation.

Additional resources

- DatabaseType Enum lists supported database dialects
- Infinispan SQL store configuration reference

6.10.1. Loading Infinispan caches from database tables

Add a SQL table cache store to your configuration if you want Infinispan to load data from a database table. When it connects to the database, Infinispan uses metadata from the table to detect column names and data types. Infinispan also automatically determines which columns in the database are part of the primary key.

Prerequisites

- Have JDBC connection details.
  You can add JDBC connection factories directly to your cache configuration.

For remote caches in production environments, you should add managed datasources to Infinispan Server configuration and specify the JNDI name in the cache configuration.

- Remote caches: Copy database drivers to the `server/lib` directory in your Infinispan Server installation.

Use the `install` command with the Infinispan Command Line Interface (CLI) to download the required drivers to the `server/lib` directory, for example:

```
install org.postgresql:postgresql:42.1.3
```

- Embedded caches: Add the `infinispan-cachestore-sql` dependency to your `pom` file.

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cachestore-sql</artifactId>
</dependency>
```

Procedure

1. Create a cache store configuration that loads from a single database table.

   Declarative

   ```
   table-jdbc-store xmlns="urn:infinispan:config:store:sql:13.0"
   ```

   Programmatic

   ```
   persistence().addStore(TableJdbcStoreConfigurationBuilder.class)
   ```

2. Specify the database dialect with either `dialect=""` or `dialect()`, for example `dialect="H2"` or `dialect="postgres"`.
3. Configure the SQL cache store with the properties you require, for example:
   - To use the same cache store across your cluster, set `shared="true"` or `shared(true)`.
   - To create a read only cache store, set `read-only="true"` or `.ignoreModifications(true)`.
4. Name the database table that loads the cache with `table-name="<database_table_name>"` or `table.name("<database_table_name>")`.
5. Add the `schema` element or the `.schemaJdbcConfigurationBuilder()` method and provide your Protobuf schema details if your database tables contain composite keys or values.

Additional resources

- JDBC connection factories
- DatabaseType Enum lists supported database dialects
- Infinispan SQL store configuration reference

6.10.2. Using SQL queries to load data and manipulate persistent storage

SQL query cache stores let you load caches from multiple database tables, including from sub-columns in database tables, and perform insert, update, and delete operations.

Prerequisites

- Have JDBC connection details.
  You can add JDBC connection factories directly to your cache configuration.
  For remote caches in production environments, you should add managed datasources to Infinispan Server configuration and specify the JNDI name in the cache configuration.
- Remote caches: Copy database drivers to the `server/lib` directory in your Infinispan Server installation.

Use the `install` command with the Infinispan Command Line Interface (CLI) to download the required drivers to the `server/lib` directory, for example:

```
install org.postgresql:postgresql:42.1.3
```

- Embedded caches: Add the `infinispan-cachestore-sql` dependency to your `pom` file and make sure database drivers are on your application classpath.

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cachestore-sql</artifactId>
</dependency>
```

Procedure

1. Create a cache store configuration that uses SQL queries.

   Declarative

   ```
   query-jdbc-store xmlns="urn:infinispan:config:store:jdbc:13.0"
   ```

   Programmatic

   ```
   persistence().addStore(QueriesJdbcStoreConfigurationBuilder.class)
   ```

2. Specify the database dialect with either `dialect=""` or `dialect()`, for example `dialect="H2"` or `dialect="postgres"`.
3. Configure the SQL cache store with the properties you require, for example:
   - To use the same cache store across your cluster, set `shared="true"` or `shared(true)`.
   - To create a read only cache store, set `read-only="true"` or `.ignoreModifications(true)`.
4. Define SQL query statements that load caches with data and modify database tables with the

`queries` element or the `queries()` method.

Query statement	Description
`SELECT`	Loads a single entry into caches. You can use wildcards but must specify parameters for keys. You can use labelled expressions.
`SELECT ALL`	Loads multiple entries into caches. You can use the `*` wildcard if the number of columns returned match the key and value columns. You can use labelled expressions.
`SIZE`	Counts the number of entries in the cache.
`DELETE`	Deletes a single entry from the cache.
`DELETE ALL`	Deletes all entries from the cache.
`UPSERT`	Modifies entries in the cache.

`DELETE`, `DELETE ALL`, and `UPSERT` statements do not apply to read only cache stores but are required if cache stores allow modifications.

Parameters in `DELETE` statements must match parameters in `SELECT` statements exactly.

Variables in `UPSERT` statements must have the same number of uniquely named variables that `SELECT` and `SELECT ALL` statements return. For example, if `SELECT` returns `foo` and `bar` this statement must take only `:foo` and `:bar` as variables. However you can apply the same named variable more than once in a statement.

SQL queries can include `JOIN`, `ON`, and any other clauses that the database supports.

5. Add the `schema` element or the `.schemaJdbcConfigurationBuilder()` method and provide your Protobuf schema details if your database tables contain composite keys or values.

Additional resources

- JDBC connection factories
- DatabaseType Enum lists supported database dialects
- Infinispan SQL store configuration reference

6.10.3. Protobuf schema for SQL cache stores

Infinispan loads keys and values from columns in database tables via SQL cache stores. If the tables in your database have composite primary keys or composite values, you need to provide Infinispan with Protobuf schema (`.proto` files) that provide a structured representation of your data.

To create Protobuf schema, Infinispan recommends annotating your Java classes and creating serialization context initializers.

For more information about creating serialization context initializers for remote and embedded caches, see the *Cache encoding and marshalling* documentation.

Keys and values with single column

The following `CREATE` statement adds a table named "books" that has two columns, `isbn` and `title`:

```
CREATE TABLE books (
    isbn NUMBER(13),
    title varchar(120)
    PRIMARY KEY(isbn)
);
```

When you use this table with a SQL cache store, Infinispan adds an entry to the cache using the `isbn` column as the key and the `title` column as the value.

Composite values

Database tables can have composite values, such as the following example where the value is composed of the `title` and `author` columns:

```
CREATE TABLE books (
    isbn NUMBER(13),
    title varchar(120),
    author varchar(80)
    PRIMARY KEY(isbn)
);
```

When you use this table with a SQL cache store, Infinispan adds an entry to the cache using the `isbn` column as the key. For the value of the entry, Infinispan requires a Protobuf schema that maps and the `title` column and the `author` column such as the following:

```
package library;

message books_value {
    optional string title = 1;
    optional string author = 2;
}
```

Composite keys and values

Database tables can have composite keys and values, such as the following example:

```
CREATE TABLE books (
    isbn NUMBER(13),
    reprint INT,
    title varchar(120),
    author varchar(80)
    PRIMARY KEY(isbn, reprint)
);
```

To use this table with a SQL cache store, you must provide Infinispan with a Protobuf schema that maps the columns to keys and values such as the following:

```
package library;

message books_key {
    required string isbn = 1;
    required int32 reprint = 2;
}

message books_value {
    optional string title = 1;
    optional string author = 2;
}
```

SQL types to Protobuf types

The following table contains default mappings of SQL data types to Protobuf data types:

SQL type	Protobuf type
int4	int32
int8	int64
float4	float
float8	double
numeric	double
bool	bool
char	string
varchar	string
text, tinytext, mediumtext, longtext	string
bytea, tinyblob, blob, mediumblob, longblob	bytes

Additional resources

- Cache encoding and marshalling
- Infinispan SQL store configuration reference

6.10.4. SQL cache store configuration examples

Declarative and programmatic examples of SQL cache store configuration.

SQL table store configuration

XML

```
<distributed-cache>
  <persistence>
    <table-jdbc-store xmlns="urn:infinispan:config:store:sql:13.0"
                      dialect="H2"
                      shared="true"
                      table-name="authors">
      <schema message-name="Author"
              package="library"
              embedded-key="true"/>
    </table-jdbc-store>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "table-jdbc-store": {
        "dialect": "H2",
        "shared": "true",
        "table-name": "authors",
        "schema": {
          "message-name": "Author",
          "package": "library",
          "embedded-key": "true"
        }
      }
    }
  }
}
```

YAML

```yaml
distributed-cache:
  persistence:
    table-jdbc-store:
      dialect: "H2"
      shared: "true"
      table-name: "authors"
      schema:
        message-name: "Author"
        package: "library"
        embedded-key: "true"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(TableJdbcStoreConfigurationBuilder.class)
      .dialect(DatabaseType.H2)
      .shared("true")
      .tableName("authors")
      .schemaJdbcConfigurationBuilder()
         .messageName("Author")
         .packageName("library")
         .embeddedKey(true);
```

Additional resources

- Infinispan SQL store configuration reference

SQL query store configuration

XML

```
<distributed-cache>
  <persistence>
    <query-jdbc-store xmlns="urn:infinispan:config:store:jdbc:13.0"
                      dialect="POSTGRES"
                      shared="true">
      <queries key-columns="isbn">
        <select-single>SELECT t1.name, t1.picture, t1.sex, t1.birthdate,
t1.accepted_tos, t2.street, t2.city, t2.zip FROM Person t1 JOIN Address t2 ON t1.name
= t2.name WHERE t1.name = :name</select-single>
        <select-all>SELECT t1.name, t1.picture, t1.sex, t1.birthdate, t1.accepted_tos,
t2.street, t2.city, t2.zip FROM Person t1 JOIN Address t2 ON t1.name =
t2.name</select-all>
        <delete-single>DELETE FROM Person t1 WHERE t1.name = :name; DELETE FROM
Address t2 where t2.name = :name</delete-single>
        <delete-all>DELETE FROM Person; DELETE FROM Address</delete-all>
        <upsert>INSERT INTO Person (name,  picture, sex, birthdate, accepted_tos)
VALUES (:name, :picture, :sex, :birthdate, :accepted_tos); INSERT INTO Address(name,
street, city, zip) VALUES (:name, :street, :city, :zip)</upsert>
        <size>SELECT COUNT(*) FROM Person</size>
      </queries>
      <schema message-name="Person"
              key-message-name="PersonID"
              package="com.example"
              embedded-key="true"/>
    </query-jdbc-store>
  </persistence>
<distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "query-jdbc-store": {
        "dialect": "POSTGRES",
        "shared": "true",
        "key-columns": "isbn",
        "queries": {
          "select-single": "SELECT t1.name, t1.picture, t1.sex, t1.birthdate,
t1.accepted_tos, t2.street, t2.city, t2.zip FROM Person t1 JOIN Address t2 ON t1.name
= t2.name WHERE t1.name = :name",
          "select-all": "SELECT t1.name, t1.picture, t1.sex, t1.birthdate,
t1.accepted_tos, t2.street, t2.city, t2.zip FROM Person t1 JOIN Address t2 ON t1.name
= t2.name",
          "delete-single": "DELETE FROM Person t1 WHERE t1.name = :name; DELETE FROM
Address t2 where t2.name = :name",
          "delete-all": "DELETE FROM Person; DELETE FROM Address",
          "upsert": "INSERT INTO Person (name,  picture, sex, birthdate, accepted_tos)
VALUES (:name, :picture, :sex, :birthdate, :accepted_tos); INSERT INTO Address(name,
street, city, zip) VALUES (:name, :street, :city, :zip)",
          "size": "SELECT COUNT(*) FROM Person"
        },
        "schema": {
          "message-name": "Books",
          "key-message-name": "BooksID",
          "package": "library",
          "embedded-key": "true"
        }
      }
    }
  }
}
```

YAML

```yaml
distributed-cache:
  persistence:
    query-jdbc-store:
      dialect: "POSTGRES"
      shared: "true"
      key-columns: "isbn"
      queries:
        select-single: "SELECT t1.name, t1.picture, t1.sex, t1.birthdate,
t1.accepted_tos, t2.street, t2.city, t2.zip FROM Person t1 JOIN Address t2 ON t1.name
= t2.name WHERE t1.name = :name"
        select-all: "SELECT t1.name, t1.picture, t1.sex, t1.birthdate,
t1.accepted_tos, t2.street, t2.city, t2.zip FROM Person t1 JOIN Address t2 ON t1.name
= t2.name"
        delete-single: "DELETE FROM Person t1 WHERE t1.name = :name; DELETE FROM
Address t2 where t2.name = :name"
        delete-all: "DELETE FROM Person; DELETE FROM Address"
        upsert: "INSERT INTO Person (name,  picture, sex, birthdate, accepted_tos)
VALUES (:name, :picture, :sex, :birthdate, :accepted_tos); INSERT INTO Address(name,
street, city, zip) VALUES (:name, :street, :city, :zip)"
        size: "SELECT COUNT(*) FROM Person"
      schema:
        message-name: "Books"
        key-message-name: "BooksID"
        package: "library"
        embedded-key: "true"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(QueriesJdbcStoreConfigurationBuilder.class)
      .dialect(DatabaseType.POSTGRES)
      .shared("true")
      .keyColumns("isbn")
      .queriesJdbcConfigurationBuilder()
         .select("SELECT t1.name, t1.picture, t1.sex, t1.birthdate, t1.accepted_tos,
t2.street, t2.city, t2.zip FROM Person t1 JOIN Address t2 ON t1.name = t2.name WHERE
t1.name = :name")
         .selectAll("SELECT t1.name, t1.picture, t1.sex, t1.birthdate,
t1.accepted_tos, t2.street, t2.city, t2.zip FROM Person t1 JOIN Address t2 ON t1.name
= t2.name")
         .delete("DELETE FROM Person t1 WHERE t1.name = :name; DELETE FROM Address t2
where t2.name = :name")
         .deleteAll("DELETE FROM Person; DELETE FROM Address")
         .upsert("INSERT INTO Person (name,  picture, sex, birthdate, accepted_tos)
VALUES (:name, :picture, :sex, :birthdate, :accepted_tos); INSERT INTO Address(name,
street, city, zip) VALUES (:name, :street, :city, :zip)")
         .size("SELECT COUNT(*) FROM Person")
      .schemaJdbcConfigurationBuilder()
         .messageName("Books")
         .keyMessageName("BooksID")
         .packageName("library")
         .embeddedKey(true);
```

Additional resources

- Infinispan SQL store configuration reference

6.10.5. SQL cache store troubleshooting

Find out about common issues and errors with SQL cache stores and how to troubleshoot them.

```
ISPN008064: No primary keys found for table <table_name>, check case sensitivity
```

Infinispan logs this message in the following cases:

- The database table does not exist.
- The database table name is case sensitive and needs to be either all lower case or all upper case, depending on the database provider.
- The database table does not have any primary keys defined.

To resolve this issue you should:

1. Check your SQL cache store configuration and ensure that you specify the name of an existing table.

2. Ensure that the database table name conforms to an case sensitivity requirements.
3. Ensure that your database tables have primary keys that uniquely identify the appropriate rows.

6.11. JDBC string-based cache stores

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

JDBC String-Based cache stores:

- Store each entry in its own row in the table to increase throughput for concurrent loads.
- Use 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.

In addition to the data table used to store cache entries, the store also creates a `_META` table for storing metadata. This table is used to ensure that any existing database content is compatible with the current Infinispan version and configuration.

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

Segmentation

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

Additional resources

- DatabaseType Enum lists supported database dialects

6.11.1. Configuring JDBC string-based cache stores

Configure Infinispan caches with JDBC string-based cache stores that can connect to databases.

Prerequisites

- Remote caches: Copy database drivers to the `server/lib` directory in your Infinispan Server installation.
- Embedded caches: Add the `infinispan-cachestore-jdbc` dependency to your `pom` file.

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cachestore-jdbc</artifactId>
</dependency>
```

Procedure

1. Create a JDBC string-based cache store configuration in one of the following ways:
   - Declaratively, add the `persistence` element or field then add `string-keyed-jdbc-store` with the following schema namespace:

     ```
     xmlns="urn:infinispan:config:store:jdbc:13.0"
     ```

   - Programmatically, add the following methods to your `ConfigurationBuilder`:

     ```
     persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
     ```

2. Specify the dialect of the database with either the `dialect` attribute or the `dialect()` method.
3. Configure any properties for the JDBC string-based cache store as appropriate.

   For example, specify if the cache store is shared with multiple cache instances with either the `shared` attribute or the `shared()` method.

4. Add a JDBC connection factory so that Infinispan can connect to the database.
5. Add a database table that stores cache entries.

JDBC string-based cache store configuration

XML

```
<distributed-cache>
  <persistence>
    <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:13.0"
                             dialect="H2">
      <connection-pool connection-url="jdbc:h2:mem:infinispan"
                       username="sa"
                       password="changeme"
                       driver="org.h2.Driver"/>
      <string-keyed-table create-on-start="true"
                          prefix="ISPN_STRING_TABLE">
        <id-column name="ID_COLUMN"
                   type="VARCHAR(255)" />
        <data-column name="DATA_COLUMN"
                     type="BINARY" />
        <timestamp-column name="TIMESTAMP_COLUMN"
                          type="BIGINT" />
        <segment-column name="SEGMENT_COLUMN"
                        type="INT"/>
      </string-keyed-table>
    </string-keyed-jdbc-store>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "string-keyed-jdbc-store": {
        "dialect": "H2",
        "string-keyed-table": {
          "prefix": "ISPN_STRING_TABLE",
          "create-on-start": true,
          "id-column": {
            "name": "ID_COLUMN",
            "type": "VARCHAR(255)"
          },
          "data-column": {
            "name": "DATA_COLUMN",
            "type": "BINARY"
          },
          "timestamp-column": {
            "name": "TIMESTAMP_COLUMN",
            "type": "BIGINT"
          },
          "segment-column": {
            "name": "SEGMENT_COLUMN",
            "type": "INT"
          }
        },
        "connection-pool": {
          "connection-url": "jdbc:h2:mem:infinispan",
          "driver": "org.h2.Driver",
          "username": "sa",
          "password": "changeme"
        }
      }
    }
  }
}
```

YAML

```yaml
distributed-cache:
  persistence:
    string-keyed-jdbc-store:
      dialect: "H2"
      string-keyed-table:
        prefix: "ISPN_STRING_TABLE"
        create-on-start: true
        id-column:
          name: "ID_COLUMN"
          type: "VARCHAR(255)"
        data-column:
          name: "DATA_COLUMN"
          type: "BINARY"
        timestamp-column:
          name: "TIMESTAMP_COLUMN"
          type: "BIGINT"
        segment-column:
          name: "SEGMENT_COLUMN"
          type: "INT"
      connection-pool:
        connection-url: "jdbc:h2:mem:infinispan"
        driver: "org.h2.Driver"
        username: "sa"
        password: "changeme"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
      .dialect(DatabaseType.H2)
      .table()
         .dropOnExit(true)
         .createOnStart(true)
         .tableNamePrefix("ISPN_STRING_TABLE")
         .idColumnName("ID_COLUMN").idColumnType("VARCHAR(255)")
         .dataColumnName("DATA_COLUMN").dataColumnType("BINARY")
         .timestampColumnName("TIMESTAMP_COLUMN").timestampColumnType("BIGINT")
         .segmentColumnName("SEGMENT_COLUMN").segmentColumnType("INT")
      .connectionPool()
         .connectionUrl("jdbc:h2:mem:infinispan")
         .username("sa")
         .password("changeme")
         .driverClass("org.h2.Driver");
```

Additional resources

- JDBC connection factories

6.12. 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.

Table 1. Configuration parameters

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

Tuning parameters

You can also specify the following RocksDB tuning parameters:

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

Configuration properties

Optionally set properties in the configuration as follows:

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

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

Segmentation

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

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

RocksDB cache store configuration

XML

```
<local-cache>
   <persistence>
      <rocksdb-store xmlns="urn:infinispan:config:store:rocksdb:13.0"
                     path="rocksdb/data">
         <expiration path="rocksdb/expired"/>
      </rocksdb-store>
   </persistence>
</local-cache>
```

JSON

```
{
  "local-cache": {
    "persistence": {
      "rocksdb-store": {
        "path": "rocksdb/data",
        "expiration": {
          "path": "rocksdb/expired"
        }
      }
    }
  }
}
```

YAML

```yaml
local-cache:
  persistence:
    rocksdb-store:
      path: "rocksdb/data"
      expiration:
        path: "rocksdb/expired"
```

ConfigurationBuilder

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

ConfigurationBuilder with properties

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

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

Reference

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

6.13. Remote cache stores

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

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

Segmentation

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

When you enable segmentation for `RemoteStore`, it uses the number of segments that you define in your Infinispan server configuration.

If the source cache is segmented and uses a different number of segments than `RemoteStore`, then incorrect values are returned for bulk operations. In this case, you should disable segmentation for `RemoteStore`.

Remote cache store configuration

XML

```
<distributed-cache>
  <persistence>
    <remote-store xmlns="urn:infinispan:config:store:remote:13.0"
                  cache="mycache"
                  raw-values="true">
      <remote-server host="one"
                     port="12111" />
      <remote-server host="two" />
      <connection-pool max-active="10"
                       exhausted-action="CREATE_NEW" />
    </remote-store>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "remote-store": {
      "cache": "mycache",
      "raw-values": "true",
      "remote-server": [
        {
          "host": "one",
          "port": "12111"
        },
        {
          "host": "two"
        }
      ],
      "connection-pool": {
        "max-active": "10",
        "exhausted-action": "CREATE_NEW"
      }
    }
  }
}
```

YAML

```
distributed-cache:
  remote-store:
    cache: "mycache"
    raw-values: "true"
    remote-server:
      - host: "one"
        port: "12111"
      - host: "two"
    connection-pool:
      max-active: "10"
      exhausted-action: "CREATE_NEW"
```

ConfigurationBuilder

```
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();
```

Reference

- Remote cache store configuration schema
- RemoteStore
- RemoteStoreConfigurationBuilder

6.14. 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 JPA cache stores, 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.
- JPA cache stores do not support segmentation.

You should use JPA cache stores with embedded Infinispan caches only.

JPA cache store configuration

XML

```
<local-cache name="vehicleCache">
   <persistence passivation="false">
      <jpa-store xmlns="urn:infinispan:config:store:jpa:13.0"
         persistence-unit="org.infinispan.persistence.jpa.configurationTest"
         entity-class="org.infinispan.persistence.jpa.entity.Vehicle">
        />
   </persistence>
</local-cache>
```

ConfigurationBuilder

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

Configuration parameters

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

Additional resources

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

6.14.1. JPA cache store example

This section provides an example for using JPA cache stores.

Prerequistes

- Configure Infinispan to marshall your JPA entities.

Procedure

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

```
<persistence-unit name="myPersistenceUnit">
    <!-- Persistence configuration goes here. -->
</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;
    ...
}
```

Additional resources

- Cache Encoding and Marshalling

6.15. 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.

`ClusterLoader` is deprecated and planned for removal in a future version.

`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.

Cluster cache loader configuration

XML

```
<distributed-cache>
  <persistence>
    <cluster-loader preload="true" remote-timeout="500"/>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence" : {
      "cluster-loader" : {
        "preload" : true,
        "remote-timeout" : "500"
      }
    }
  }
}
```

YAML

```
distributed-cache:
  persistence:
    clusterLoader:
      preload: "true"
      remoteTimeout: "500"
```

ConfigurationBuilder

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

Additional resources

- Infinispan configuration schema
- ClusterLoader
- ClusterLoaderConfiguration

6.16. Creating custom cache store implementations

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

6.16.1. Infinispan Persistence SPI

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

Portability across JCache-compliant vendors

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

Simplified transaction integration

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

Parallel iteration

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

Reduced serialization resulting in less CPU usage

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

Additional resources

- Persistence SPI
- NonBlockingStore
- JSR-107

6.16.2. Creating cache stores

Create custom cache stores with implementations of the `NonBlockingStore` API.

Procedure

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

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

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

6.16.3. Examples of custom cache store configuration

The following are examples show how to configure Infinispan with custom cache store implementations:

XML

```
<distributed-cache>
  <persistence>
    <store class="org.infinispan.persistence.example.MyInMemoryStore" />
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence" : {
      "store" : {
        "class" : "org.infinispan.persistence.example.MyInMemoryStore"
      }
    }
  }
}
```

YAML

```
distributed-cache:
  persistence:
    store:
      class: "org.infinispan.persistence.example.MyInMemoryStore"
```

ConfigurationBuilder

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

6.16.4. Deploying custom cache stores

To use your cache store implementation with Infinispan Server, you must provide it with a JAR file.

Prerequisites

- Stop Infinispan Server if it is running.

  Infinispan loads JAR files at startup only.

Procedure

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

6.17. Migrating data between cache stores

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

6.17.1. Cache store migrator

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

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

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

`StoreMigrator` also lets you migrate data from one type of cache store to another. For example, you can migrate from a JDBC string-based cache store to a RocksDB cache store.

`StoreMigrator` cannot migrate data from segmented cache stores to:

- Non-segmented cache store.
- Segmented cache stores that have a different number of segments.

6.17.2. Getting the cache store migrator

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

Procedure

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

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

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

    <dependencies>
      <dependency>
        <groupId>org.infinispan</groupId>
        <artifactId>infinispan-tools</artifactId>
      </dependency>
      <!-- Additional dependencies -->
    </dependencies>

    <build>
      <plugins>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>exec-maven-plugin</artifactId>
          <version>1.2.1</version>
          <executions>
            <execution>
              <goals>
                <goal>java</goal>
              </goals>
            </execution>
          </executions>
          <configuration>
            <mainClass>
org.infinispan.tools.store.migrator.StoreMigrator</mainClass>
            <arguments>
              <argument>path/to/migrator.properties</argument>
            </arguments>
          </configuration>
        </plugin>
      </plugins>
    </build>
</project>
```

6.17.3. Configuring the cache store migrator

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

Procedure

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

```
source.type=SOFT_INDEX_FILE_STORE
source.cache_name=myCache
source.location=/path/to/source/sifs
source.version=12
```

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

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

Configuration properties for the cache store migrator

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

Table 2. Cache Store Type Property

Property	Description	Required/Optional

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

Table 3. Common Properties

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

Table 4. JDBC Properties

Property	Description	Required/Optional
`dialect`	Specifies the dialect of the underlying database.	Required
`version`	Specifies the marshaller version for source cache stores. Set the value that matches the Infinispan major version of the source cluster. For example; set a value of `12` for Infinispan 12.x.	Required for source stores only. For example: `source.version=12`
`marshaller.class`	Specifies a custom marshaller class.	Required if using custom marshallers.
`marshaller.externalizers`	Specifies a comma-separated list of custom `AdvancedExternalizer` implementations to load in this format: `[id]:<Externalizer class>`	Optional

Property	Description	Required/Optional
connection_pool.connection_url	Specifies the JDBC connection URL.	Required
connection_pool.driver_class	Specifies the class of the JDBC driver.	Required
connection_pool.username	Specifies a database username.	Required
connection_pool.password	Specifies a password for the database username.	Required
db.major_version	Sets the database major version.	Optional
db.minor_version	Sets the database minor version.	Optional
db.disable_upsert	Disables database upsert.	Optional
db.disable_indexing	Specifies if table indexes are created.	Optional
table.string.table_name_prefix	Specifies additional prefixes for the table name.	Optional
table.string.<id|data|timestamp>.name	Specifies the column name.	Required
table.string.<id|data|timestamp>.type	Specifies the column type.	Required
key_to_string_mapper	Specifies the TwoWayKey2StringMapper class.	Optional

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

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

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

Table 5. RocksDB Properties

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

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

Table 6. SingleFileStore Properties

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

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

Table 7. SoftIndexFileStore Properties

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

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

6.17.4. Migrating Infinispan cache stores

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

Prerequisites

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

Procedure

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

```
mvn exec:java
```

Chapter 7. Setting up partition handling

7.1. Partition handling

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

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

Split brain

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

`numOwners` nodes crash in sequence

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

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

Strategy	Description	CAP
DENY_READ_WRITES	If the partition does not have all owners for a given segment, both reads and writes are denied for all keys in that segment.	Consistency

Strategy	Description	CAP
ALLOW_READS	Allows reads for a given key if it exists in this partition, but only allows writes if this partition contains all owners of a segment. This is still a consistent approach because some entries are readable if available in this partition, but from a client application perspective it is not deterministic.	Consistency
ALLOW_READ_WRITES	Allow entries on each partition to diverge, with conflict resolution attempted upon the partitions merging.	Availability

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

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

7.1.1. Split brain

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

Split strategies

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

ALLOW_READ_WRITES

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

DENY_READ_WRITES

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

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

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

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

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

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

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

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

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

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

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

As another example, the cluster could split in two partitions `O1 = {A, B, C}` and `O2 = {D}`, partition

`O1` will stay fully available (rebalancing cache entries on the remaining members). Partition `O2`, however, will detect a split and enter the degraded mode. Since it doesn't have any fully owned keys, it will reject any read or write operation with an `AvailabilityException`.

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

ALLOW_READS

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

Current limitations

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

7.1.2. Successive nodes stopped

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

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

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

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

7.1.3. Conflict manager

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

Detecting conflicts

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

Merge policies

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

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

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

Currently Infinispan provides the following implementations of EntryMergePolicy:

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

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

Usage

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

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

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

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

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

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

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

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

7.1.4. Configuring partition handling

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

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

The same can be achieved programmatically:

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

Implementing a custom merge policy

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

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

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

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

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

      return the_solved_CacheEntry;
   }
```

Deploying custom merge policies to Infinispan Server

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

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

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

7.1.5. Monitoring and administration

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

The availability mode is also accessible via the AdvancedCache interface:

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

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

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