Red Hat Data Grid 8.4

Configuring Data Grid Caches

Configure Data Grid caches to customize your deployment

Last Updated: 2024-04-19

Red Hat Data Grid 8.4 Configuring Data Grid Caches

Configure Data Grid caches to customize your deployment

Legal Notice

Abstract

Configure Data Grid deployments to use features and capabilities that suit your business requirements.

Table of Contents

RED HAT DATA GRID

Data Grid is a high-performance, distributed in-memory data store.

Schemaless data structure

Flexibility to store different objects as key-value pairs.

Grid-based data storage

Designed to distribute and replicate data across clusters.

Elastic scaling

Dynamically adjust the number of nodes to meet demand without service disruption.

Data interoperability

Store, retrieve, and query data in the grid from different endpoints.

DATA GRID DOCUMENTATION

Documentation for Data Grid is available on the Red Hat customer portal.

- Data Grid 8.4 Documentation
- Data Grid 8.4 Component Details
- Supported Configurations for Data Grid 8.4
- Data Grid 8 Feature Support
- Data Grid Deprecated Features and Functionality

DATA GRID DOWNLOADS

Access the Data Grid Software Downloads on the Red Hat customer portal.

NOTE

You must have a Red Hat account to access and download Data Grid software.

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

CHAPTER 1. DATA GRID CACHES

Data Grid 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 Data Grid 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 entry point for interacting with Data Grid. Cache Managers control cache lifecycle; creating, modifying, and deleting cache instances. Cache Managers also provide cluster management and monitoring along with the ability to execute code across nodes.

Data Grid provides two **CacheManager** implementations:

EmbeddedCacheManager

Entry point for caches when running Data Grid inside the same Java Virtual Machine (JVM) as the client application.

RemoteCacheManager

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

NOTE

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

1.3. CACHE MODES

TIP

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

Local

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

Replicated

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

Distributed

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

Invalidation

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

Scattered

Data Grid stores cache entries across a subset of nodes.
By default Data Grid assigns a primary owner and a backup owner to each cache entry in scattered caches.
Data Grid assigns primary owners in the same way as with distributed caches, while backup owners are always the nodes that initiate the write operations.
Data Grid 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:

Cache mode	Clustered?	Read performance	Write performance	Capacity	Availability	Capabilities
Local	*No*	**High** (local)	**High** (local)	*Single node*	*Single node*	**Complete**
Simple	*No*	**Highest** (local)	**Highest** (local)	*Single node*	*Single node*	*Partial:* no transactions, persistence, or indexing.
Invalidation	**Yes**	**High** (local)	*Low* (all nodes, no data)	*Single node*	*Single node*	*Partial:* no indexing.
Replicated	**Yes**	**High** (local)	*Lowest* (all nodes)	*Smallest node*	**All nodes**	**Complete**
Distributed	**Yes**	*Medium* (owners)	*Medium* (owner nodes)	Sum of all nodes capacity divided by the number of owners.	*Owner nodes*	**Complete**

Cache mode	Clustered?	Read performance	Write performance	Capacity	Availability	Capabilities
Scattered	**Yes**	*Medium* (primary)	*Higher* (single RPC)	Sum of all nodes capacity divided by 2.	*Owner nodes*	*Partial:* no transactions.

1.4. LOCAL CACHES

Data Grid 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 Data Grid **Cache** API extends the **ConcurrentMap** API in Java, making it easy to migrate from a map to a Data Grid cache.

Local cache configuration

XML

```
<local-cache name="mycache"
             statistics="true">
  <encoding media-type="application/x-protostream"/>
</local-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 Data Grid 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, Data Grid 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 Data Grid clusters that replicate data across nodes.

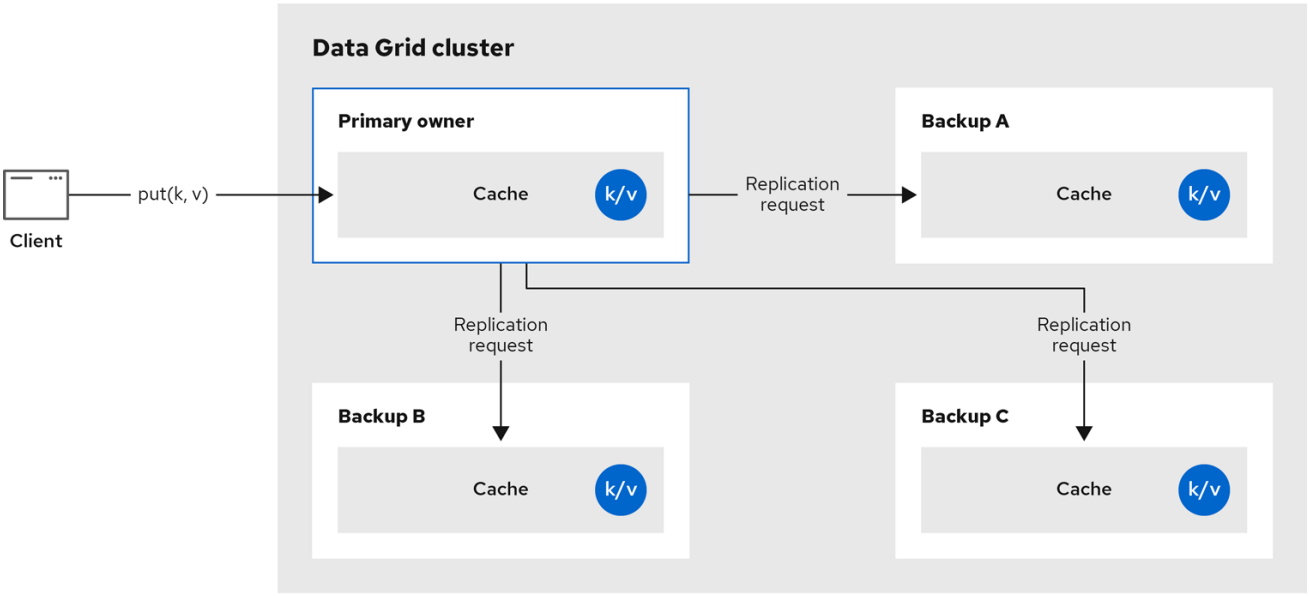
2.1. REPLICATED CACHES

Data Grid 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 2.1. 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

Data Grid 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.

Data Grid 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 2.2. Cluster replication

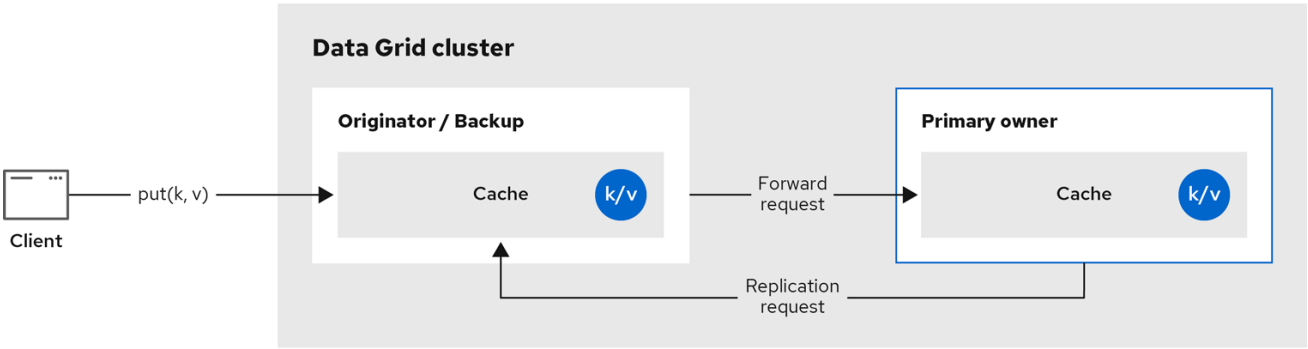
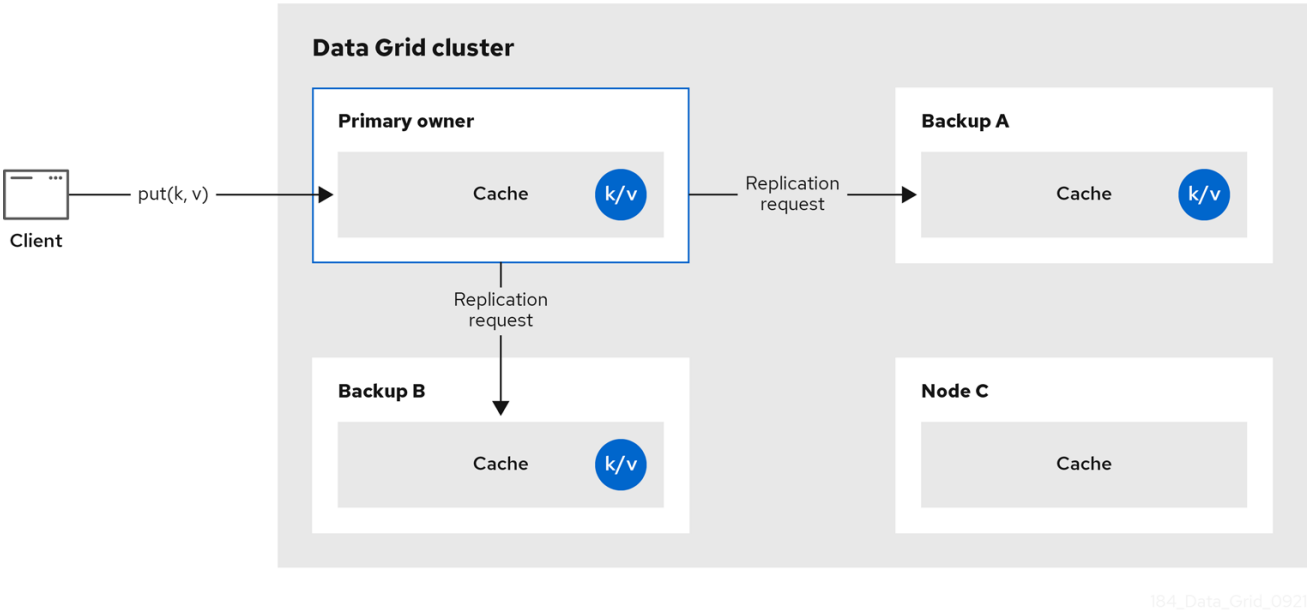


Figure 2.3. Distributed cache



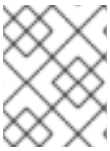
Read operations

Read operations request the value from the primary owner. If the primary owner does not respond in a reasonable amount of time, Data Grid 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.



NOTE

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 Data Grid 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 Data Grid 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.

2.2.3.1. 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

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

JSON

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

YAML

```
infinispan:
  cacheContainer:
    zeroCapacityNode: "true"
```

ConfigurationBuilder

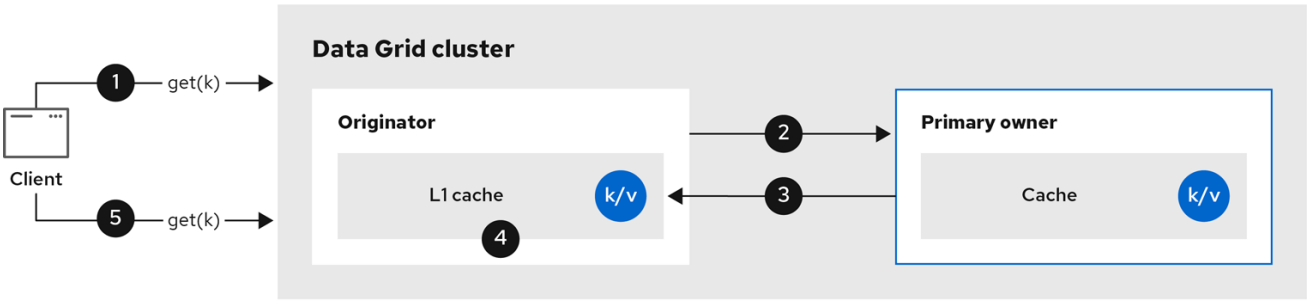
```
new GlobalConfigurationBuilder().zeroCapacityNode(true);
```

2.2.4. Level one (L1) caches

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

The following diagram illustrates how L1 caches work:

Figure 2.4. 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 Data Grid 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.

NOTE

Data Grid 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

Server hinting increases availability of data in distributed caches by replicating entries across as many servers, racks, and data centers as possible.

NOTE

Server hinting applies only to distributed caches.

When Data Grid distributes the copies of your data, it follows the order of precedence: site, rack, machine, and node. All of the configuration attributes are optional. For example, when you specify only the rack IDs, then Data Grid distributes the copies across different racks and nodes.

Server hinting can impact cluster rebalancing operations by moving more segments than necessary if the number of segments for the cache is too low.

TIP

An alternative for clusters in multiple data centers is cross-site replication.

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

```
cacheContainer:
  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");
```

Additional resources

- org.infinispan.configuration.global.TransportConfigurationBuilder

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, Data Grid 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, Data Grid 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;
   }
   }
}
```

NOTE

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 Data Grid 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>
```

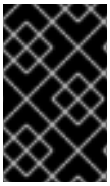
Advanced API
AdvancedCache has two group-specific methods:

- **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

Invalidation cache mode in Data Grid is designed to optimize systems that perform high volumes of read operations to a shared permanent data store. You can use invalidation mode to reduce the number of database writes when state changes occur.



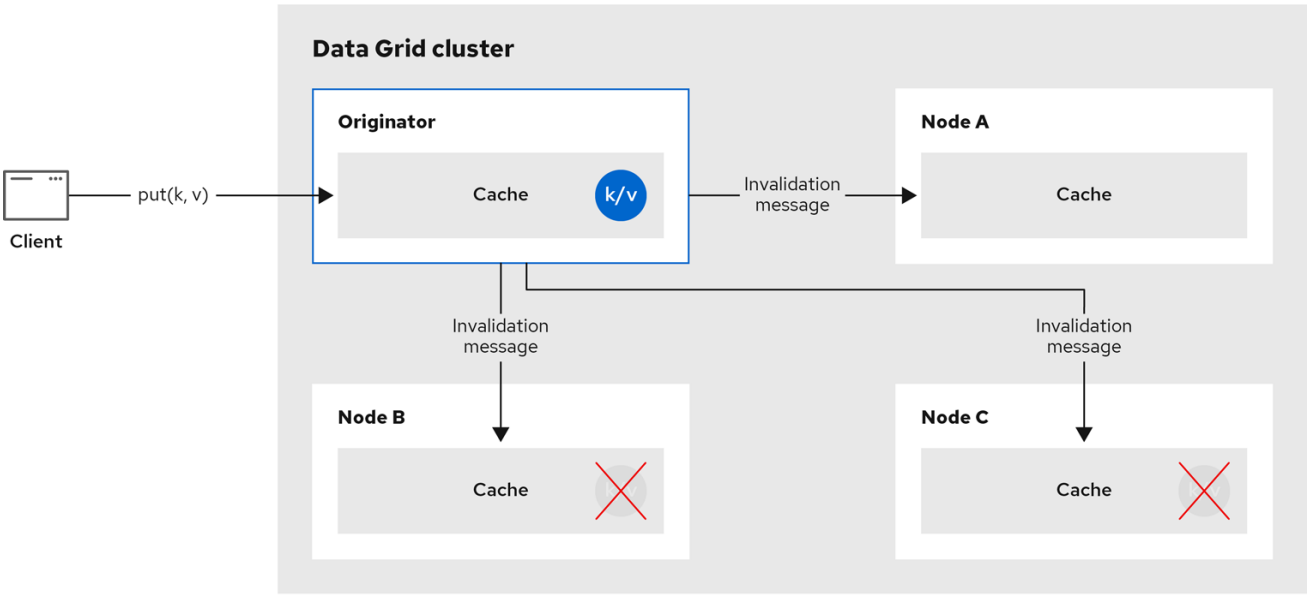
IMPORTANT

Invalidation cache mode is deprecated for Data Grid remote deployments. Use invalidation cache mode with embedded caches that are stored in shared cache stores only.

Invalidation cache mode is effective only when you have a permanent data store, such as a database, and are only using Data Grid as an optimization in a read-heavy system to prevent hitting the database for every read.

When a cache is configured for invalidation, each data change in a cache triggers a message to other caches in the cluster, informing them that their data is now stale and should be removed from memory. Invalidation messages remove stale values from other nodes' memory. The 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. The update to the shared store is typically handled by user application code or Hibernate.

Figure 2.5. 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)**.

IMPORTANT

Invalidation mode is suitable only for shared stores where all nodes can access the same data. Using invalidation mode without a persistent store is impractical, as updated values need to be read from a shared store for consistency across nodes.

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. This can lead to storing stale values, so only use it if you have a high tolerance for stale values.

Synchronous or asynchronous replication

When synchronous, a write operation 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.

NOTE

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 Data Grid 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

Data Grid handles cluster topology changes dynamically. This means that nodes do not need to wait for other nodes to join the cluster before Data Grid 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 Data Grid 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 Data Grid configuration.

Initial cluster size configuration

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:
  cacheContainer:
    transport:
      initialClusterSize: "4"
      initialClusterTimeout: "30000"
```

ConfigurationBuilder

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

CHAPTER 3. DATA GRID CACHE CONFIGURATION

Cache configuration controls how Data Grid stores your data.

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

Your configuration also defines the characteristics of your caches and enables the Data Grid capabilities that you want to use when handling data. For instance, you can configure how Data Grid 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, JSON, and YAML format, according to the Data Grid 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 Data Grid Operator for clusters running on OpenShift.

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.

NOTE

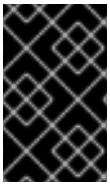
Data Grid Server configuration extends **infinispan.xml** to include cluster transport mechanisms, security realms, and endpoint configuration. If you declare caches as part of your Data Grid Server configuration you should use management tooling, such as Ansible or Chef, to keep it synchronized across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

To dynamically synchronize remote caches across Data Grid 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 Data Grid schema. Configuration in JSON format must follow the structure of an XML configuration, elements correspond to objects and attributes correspond to fields.



IMPORTANT

Data Grid restricts characters to a maximum of **255** for a cache name or a cache template name. If you exceed this character limit, Data Grid throws an exception. Write succinct cache names and cache template names.

IMPORTANT

A file system might set a limitation for the length of a file name, so ensure that a cache's name does not exceed this limitation. If a cache name exceeds a file system's naming limitation, general operations or initialing operations towards that cache might fail. Write succinct file names.

Distributed caches

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"/>
    <indexed-entities>
      <indexed-entity>org.infinispan.Person</indexed-entity>
    </indexed-entities>
  </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"
          },
          "indexed-entities": [
            "org.infinispan.Person"
          ]
    },
    "partition-handling" : {
      "when-split" : "ALLOW_READ_WRITES",
      "merge-policy" : "PREFERRED_NON_NULL"
    },
    "persistence" : {
      "passivation" : false
    }
  }
}
```

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"
    indexedEntities:
      - "org.infinispan.Person"
  partitionHandling:
    whenSplit: "ALLOW_READ_WRITES"
    mergePolicy: "PREFERRED_NON_NULL"
  persistence:
    passivation: "false"
    # Persistent storage configuration.
```

Replicated caches

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"/>
    <indexed-entities>
      <indexed-entity>org.infinispan.Person</indexed-entity>
    </indexed-entities>
  </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"
        },
      "indexed-entities": [
        "org.infinispan.Person"
      ]
    },
    "partition-handling" : {
      "when-split" : "ALLOW_READ_WRITES",
      "merge-policy" : "PREFERRED_NON_NULL"
    },
    "persistence" : {
      "passivation" : false
    }
  }
}
```

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"
    indexedEntities:
      - "org.infinispan.Person"
  partitionHandling:
    whenSplit: "ALLOW_READ_WRITES"
    mergePolicy: "PREFERRED_NON_NULL"
  persistence:
    passivation: "false"
    # Persistent storage configuration.
```

Multiple caches

XML

```
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:14.0 https://infinispan.org/schemas/infinispan-config-
14.0.xsd
                          urn:infinispan:server:14.0 https://infinispan.org/schemas/infinispan-server-14.0.xsd"
      xmlns="urn:infinispan:config:14.0"
      xmlns:server="urn:infinispan:server:14.0">
  <cache-container name="default"
                   statistics="true">
    <distributed-cache name="mycacheone"
                       mode="ASYNC"
                       statistics="true">
      <encoding media-type="application/x-protostream"/>
      <expiration lifespan="300000"/>
      <memory max-size="400MB"
              when-full="REMOVE"/>
    </distributed-cache>
    <distributed-cache name="mycachetwo"
                       mode="SYNC"
                       statistics="true">
      <encoding media-type="application/x-protostream"/>
      <expiration lifespan="300000"/>
      <memory max-size="400MB"
              when-full="REMOVE"/>
    </distributed-cache>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "name" : "default",
      "statistics" : "true",
      "caches" : {
        "mycacheone" : {
          "distributed-cache" : {
```

```
            "mode": "ASYNC",
            "statistics": "true",
            "encoding": {
              "media-type": "application/x-protostream"
            },
            "expiration" : {
              "lifespan" : "300000"
            },
            "memory": {
              "max-size": "400MB",
              "when-full": "REMOVE"
            }
          }
        },
        "mycachetwo" : {
          "distributed-cache" : {
            "mode": "SYNC",
            "statistics": "true",
            "encoding": {
              "media-type": "application/x-protostream"
            },
            "expiration" : {
              "lifespan" : "300000"
            },
            "memory": {
              "max-size": "400MB",
              "when-full": "REMOVE"
            }
          }
        }
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    name: "default"
    statistics: "true"
    caches:
      mycacheone:
       distributedCache:
          mode: "ASYNC"
          statistics: "true"
          encoding:
            mediaType: "application/x-protostream"
          expiration:
            lifespan: "300000"
          memory:
            maxSize: "400MB"
            whenFull: "REMOVE"
      mycachetwo:
        distributedCache:
```

```
    mode: "SYNC"
    statistics: "true"
    encoding:
      mediaType: "application/x-protostream"
    expiration:
      lifespan: "300000"
    memory:
      maxSize: "400MB"
      whenFull: "REMOVE"
```

Additional resources

- Data Grid configuration schema reference
- infinispan-config-14.0.xsd

3.2. ADDING CACHE TEMPLATES

The Data Grid 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 Data Grid 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 Data Grid configuration.

Cache template example

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

```
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.

TIP

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

Prerequisites

- Add at least one cache template to the Cache Manager.

Procedure

1. Open your Data Grid configuration for editing.
2. Specify the template from which the cache inherits with the **configuration** attribute or field.

3. Save and close your Data Grid configuration.

Cache configuration inherited from a template

XML

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

JSON

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

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

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

JSON

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

YAML

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

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, Data Grid applies the configuration template.

NOTE

Data Grid throws exceptions if cache names match more than one wildcard.

Template wildcard example

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

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

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 Data Grid 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).

NOTE

Data Grid 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>
```

Data Grid also provides an **infinispan-config-fragment-14.0.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:14.0 https://infinispan.org/schemas/infinispan-config-fragment-14.0.xsd"
             xmlns="urn:infinispan:config:14.0"
             name="mycache"/>
```

Additional resources

- XInclude specification

3.3. CREATING REMOTE CACHES

When you create remote caches at runtime, Data Grid 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:

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

3.3.1. Default Cache Manager

Data Grid Server provides a default Cache Manager that controls the lifecycle of remote caches. Starting Data Grid Server automatically instantiates the Cache Manager so you can create and delete remote caches and other resources like Protobuf schema.

After you start Data Grid Server and add user credentials, you can view details about the Cache Manager and get cluster information from Data Grid 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.

Default Cache Manager configuration

XML

```
<infinispan>
  <!-- Creates a Cache Manager named "default" and enables 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:}"/>
    <!-- Requires user permission to access caches and perform operations. -->
    <security>
      <authorization/>
    </security>
  </cache-container>
</infinispan>
```

JSON

```
{
  "infinispan" : {
    "jgroups" : {
      "transport" : "org.infinispan.remoting.transport.jgroups.JGroupsTransport"
    },
    "cache-container" : {
      "name" : "default",
      "statistics" : "true",
      "transport" : {
        "cluster" : "cluster",
        "node-name" : "",
        "stack" : "tcp"
      },
      "security" : {
        "authorization" : {}
      }
    }
  }
}
```

YAML

```
infinispan:
  jgroups:
    transport: "org.infinispan.remoting.transport.jgroups.JGroupsTransport"
  cacheContainer:
    name: "default"
    statistics: "true"
    transport:
      cluster: "cluster"
      nodeName: ""
      stack: "tcp"
    security:
      authorization: ~
```

3.3.2. Creating caches with Data Grid Console

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

Prerequisites

- Create a Data Grid user with **admin** permissions.
- Start at least one Data Grid Server instance.
- Have a Data Grid cache configuration.

Procedure

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

3.3.3. Creating remote caches with the Data Grid CLI

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

Prerequisites

- Create a Data Grid user with **admin** permissions.
- Start at least one Data Grid Server instance.
- Have a Data Grid cache configuration.

Procedure

1. Start the CLI.

```
bin/cli.sh
```

2. Run the **connect** command and enter your username and password when prompted.

3. Use the **create cache** command to create remote caches.
   For example, create a cache named "mycache" from a file named **mycache.xml** as follows:

```
create cache --file=mycache.xml mycache
```

Verification

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

```
ls caches
mycache
```

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

```
describe caches/mycache
```

3.3.4. Creating remote caches from Hot Rod clients

Use the Data Grid Hot Rod API to create remote caches on Data Grid 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 Data Grid Tutorials.

Prerequisites

- Create a Data Grid user with **admin** permissions.
- Start at least one Data Grid Server instance.
- Have a Data Grid cache configuration.

Procedure

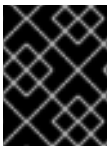
- Invoke the **remoteCache()** method as part of your the **ConfigurationBuilder**.
- Set the **configuration** or **configuration_uri** properties in the **hotrod-client.properties** file on your classpath.

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/infinispan.xml
```



IMPORTANT

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 Data Grid REST API to create remote caches on Data Grid Server from any suitable HTTP client.

Prerequisites

- Create a Data Grid user with **admin** permissions.
- Start at least one Data Grid Server instance.
- Have a Data Grid 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

Data Grid provides an **EmbeddedCacheManager** API that lets you control both the Cache Manager and embedded cache lifecycles programmatically.

3.4.1. Adding Data Grid to your project

Add Data Grid to your project to create embedded caches in your applications.

Prerequisites

- Configure your project to get Data Grid 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. Creating and using embedded caches

Data Grid provides a **GlobalConfigurationBuilder** API that controls the Cache Manager and a **ConfigurationBuilder** API that configures caches.

Prerequisites

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

Procedure

1. Initialize a **CacheManager**.

NOTE

You must always call the **cacheManager.start()** method to initialize a **CacheManager** before you can create caches. Default constructors do this for you but there are overloaded versions of the constructors that do not.

Cache Managers are also heavyweight objects and Data Grid recommends instantiating only one instance per JVM.

2. Use the **ConfigurationBuilder** API to define cache configuration.
3. Obtain caches with **getCache()**, **createCache()**, or **getOrCreateCache()** methods. Data Grid recommends using the **getOrCreateCache()** method because it either creates a cache on all nodes or returns an existing cache.
4. If necessary use the **PERMANENT** flag for caches to survive restarts.
5. Stop the **CacheManager** by calling the **cacheManager.stop()** method to release JVM resources and gracefully shutdown any caches.

```
// 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).getOrCreateC
ache("myCache", builder.build());
// Stop the Cache Manager.
cacheManager.stop();
```

getCache() method

Invoke the **getCache(String)** method to obtain caches, as follows:

```
Cache<String, String> myCache = manager.getCache("myCache");
```

The preceding operation creates a cache named **myCache**, if it does not already exist, and returns it.

Using the **getCache()** method creates the cache only on the node where you invoke the method. In other words, it performs a local operation that must be invoked on each node across the cluster. Typically, applications deployed across multiple nodes obtain caches during initialization to ensure that caches are *symmetric* and exist on each node.

createCache() method

Invoke the **createCache()** method to create caches dynamically across the entire cluster.

```
Cache<String, String> myCache = manager.administration().createCache("myCache",
"myTemplate");
```

The preceding operation also automatically creates caches on any nodes that subsequently join the cluster.

Caches that you create with the **createCache()** method are ephemeral by default. If the entire cluster shuts down, the cache is not automatically created again when it restarts.

PERMANENT flag

Use the PERMANENT flag to ensure that caches can survive restarts.

```
Cache<String, String> myCache =
manager.administration().withFlags(AdminFlag.PERMANENT).createCache("myCache",
"myTemplate");
```

For the PERMANENT flag to take effect, you must enable global state and set a configuration storage provider.

For more information about configuration storage providers, see GlobalStateConfigurationBuilder#configurationStorage().

Additional resources

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

3.4.3. Cache API

Data Grid provides a Cache interface that exposes simple methods for adding, retrieving and removing entries, including atomic mechanisms exposed by the JDK's ConcurrentMap interface. Based on the cache mode used, invoking these methods will trigger a number of things to happen, potentially even

including replicating an entry to a remote node or looking up an entry from a remote node, or potentially a cache store.

For simple usage, using the Cache API should be no different from using the JDK Map API, and hence migrating from simple in-memory caches based on a Map to Data Grid's Cache should be trivial.

Performance Concerns of Certain Map Methods

Certain methods exposed in Map have certain performance consequences when used with Data Grid, such as size() , values() , keySet() and entrySet() . Specific methods on the **keySet**, **values** and **entrySet** are fine for use please see their Javadoc for further details.

Attempting to perform these operations globally would have large performance impact as well as become a scalability bottleneck. As such, these methods should only be used for informational or debugging purposes only.

It should be noted that using certain flags with the withFlags() method can mitigate some of these concerns, please check each method's documentation for more details.

Mortal and Immortal Data

Further to simply storing entries, Data Grid's cache API allows you to attach mortality information to data. For example, simply using put(key, value) would create an *immortal* entry, i.e., an entry that lives in the cache forever, until it is removed (or evicted from memory to prevent running out of memory). If, however, you put data in the cache using put(key, value, lifespan, timeunit) , this creates a *mortal* entry, i.e., an entry that has a fixed lifespan and expires after that lifespan.

In addition to *lifespan* , Data Grid also supports *maxIdle* as an additional metric with which to determine expiration. Any combination of lifespans or maxIdles can be used.

putForExternalRead operation

Data Grid's Cache class contains a different 'put' operation called putForExternalRead . This operation is particularly useful when Data Grid is used as a temporary cache for data that is persisted elsewhere. Under heavy read scenarios, contention in the cache should not delay the real transactions at hand, since caching should just be an optimization and not something that gets in the way.

To achieve this, **putForExternalRead()** acts as a put call that only operates if the key is not present in the cache, and fails fast and silently if another thread is trying to store the same key at the same time. In this particular scenario, caching data is a way to optimise the system and it's not desirable that a failure in caching affects the on-going transaction, hence why failure is handled differently. **putForExternalRead()** is considered to be a fast operation because regardless of whether it's successful or not, it doesn't wait for any locks, and so returns to the caller promptly.

To understand how to use this operation, let's look at basic example. Imagine a cache of Person instances, each keyed by a PersonId , whose data originates in a separate data store. The following code shows the most common pattern of using putForExternalRead within the context of this example:

```
// Id of the person to look up, provided by the application
PersonId id = ...;

// Get a reference to the cache where person instances will be stored
Cache<PersonId, Person> cache = ...;

// First, check whether the cache contains the person instance
// associated with with the given id
Person cachedPerson = cache.get(id);
```

```
if (cachedPerson == null) {
   // The person is not cached yet, so query the data store with the id
   Person person = dataStore.lookup(id);

   // Cache the person along with the id so that future requests can
   // retrieve it from memory rather than going to the data store
   cache.putForExternalRead(id, person);
} else {
   // The person was found in the cache, so return it to the application
   return cachedPerson;
}
```

Note that putForExternalRead should never be used as a mechanism to update the cache with a new Person instance originating from application execution (i.e. from a transaction that modifies a Person's address). When updating cached values, please use the standard put operation, otherwise the possibility of caching corrupt data is likely.

3.4.3.1. AdvancedCache API

In addition to the simple Cache interface, Data Grid offers an AdvancedCache interface, geared towards extension authors. The AdvancedCache offers the ability to access certain internal components and to apply flags to alter the default behavior of certain cache methods. The following code snippet depicts how an AdvancedCache can be obtained:

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

3.4.3.1.1. Flags

Flags are applied to regular cache methods to alter the behavior of certain methods. For a list of all available flags, and their effects, see the Flag enumeration. Flags are applied using AdvancedCache.withFlags() . This builder method can be used to apply any number of flags to a cache invocation, for example:

```
advancedCache.withFlags(Flag.CACHE_MODE_LOCAL, Flag.SKIP_LOCKING)
   .withFlags(Flag.FORCE_SYNCHRONOUS)
   .put("hello", "world");
```

3.4.3.2. Asynchronous API

In addition to synchronous API methods like Cache.put() , Cache.remove() , etc., Data Grid also has an asynchronous, non-blocking API where you can achieve the same results in a non-blocking fashion.

These methods are named in a similar fashion to their blocking counterparts, with "Async" appended. E.g., Cache.putAsync() , Cache.removeAsync() , etc. These asynchronous counterparts return a CompletableFuture that contains the actual result of the operation.

For example, in a cache parameterized as **Cache<String, String>**, **Cache.put(String key, String value)** returns **String** while **Cache.putAsync(String key, String value)** returns **CompletableFuture<String>**.

3.4.3.2.1. Why use such an API?

Non-blocking APIs are powerful in that they provide all of the guarantees of synchronous communications - with the ability to handle communication failures and exceptions - with the ease of

not having to block until a call completes. This allows you to better harness parallelism in your system. For example:

```
Set<CompletableFuture<?>> futures = new HashSet<>();
futures.add(cache.putAsync(key1, value1)); // does not block
futures.add(cache.putAsync(key2, value2)); // does not block
futures.add(cache.putAsync(key3, value3)); // does not block

// the remote calls for the 3 puts will effectively be executed
// in parallel, particularly useful if running in distributed mode
// and the 3 keys would typically be pushed to 3 different nodes
// in the cluster

// check that the puts completed successfully
for (CompletableFuture<?> f: futures) f.get();
```

3.4.3.2.2. Which processes actually happen asynchronously?

There are 4 things in Data Grid that can be considered to be on the critical path of a typical write operation. These are, in order of cost:

- network calls
- marshalling
- writing to a cache store (optional)
- locking

Using the async methods will take the network calls and marshalling off the critical path. For various technical reasons, writing to a cache store and acquiring locks, however, still happens in the caller's thread.

CHAPTER 4. ENABLING AND CONFIGURING DATA GRID STATISTICS AND JMX MONITORING

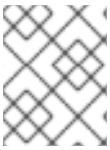
Data Grid can provide Cache Manager and cache statistics as well as export JMX MBeans.

4.1. CONFIGURING DATA GRID METRICS

Data Grid generates metrics that are compatible with any monitoring system.

- 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, Data Grid generates gauges when you enable statistics but you can also configure it to generate histograms.



NOTE

Data Grid metrics are provided at the **vendor** scope. Metrics related to the JVM are provided in the **base** scope.

Procedure

1. Open your Data Grid 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"
```

Additional resources

- Micrometer Prometheus

4.2. REGISTERING JMX MBEANS

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

Procedure

1. Open your Data Grid 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

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

4.2.1. Enabling JMX remote ports

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

You can enable remote JMX ports using one of the following approaches:

- Enable remote JMX ports that require authentication to one of the Data Grid Server security realms.
- Enable remote JMX ports manually using the standard Java management configuration options.

Prerequisites

- For remote JMX with authentication, define JMX specific user roles using the default security realm. Users must have **controlRole** with read/write access or the **monitorRole** with read-only access to access any JMX resources.

Procedure

Start Data Grid Server with a remote JMX port enabled using one of the following ways:

- Enable remote JMX through port **9999**.

```
bin/server.sh --jmx 9999
```

WARNING

Using remote JMX with SSL disabled is not intended for production environments.

- Pass the following system properties to Data Grid Server at startup.

```
bin/server.sh -Dcom.sun.management.jmxremote.port=9999 -
Dcom.sun.management.jmxremote.authenticate=false -
Dcom.sun.management.jmxremote.ssl=false
```

WARNING

Enabling remote JMX with no authentication or SSL is not secure and not recommended in any environment. Disabling authentication and SSL allows unauthorized users to connect to your server and access the data hosted there.

Additional resources

- Creating security realms

4.2.2. Data Grid MBeans

Data Grid 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 Data Grid cache and cluster health statistics.

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

Additional resources

- Data Grid JMX Components

4.2.3. Registering MBeans in custom MBean servers

Data Grid 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 Data Grid to register JMX MBeans.

Procedure

1. Open your Data Grid 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

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

JSON

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

YAML

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

4.3. EXPORTING METRICS DURING A STATE TRANSFER OPERATION

You can export time metrics for clustered caches that Data Grid redistributes across nodes.

A state transfer operation occurs when a clustered cache topology changes, such as a node joining or leaving a cluster. During a state transfer operation, Data Grid exports metrics from each cache, so that you can determine a cache's status. A state transfer exposes attributes as properties, so that Data Grid can export metrics from each cache.

NOTE

You cannot perform a state transfer operation in invalidation mode.

Data Grid generates time metrics that are compatible with the REST API and the JMX API.

Prerequisites

- Configure Data Grid metrics.
- Enable metrics for your cache type, such as embedded cache or remote cache.
- Initiate a state transfer operation by changing your clustered cache topology.

Procedure

- Choose one of the following methods:
  - Configure Data Grid to use the REST API to collect metrics.
  - Configure Data Grid to use the JMX API to collect metrics.

Additional resources

- Enabling and configuring Data Grid statistics and JMX monitoring (Data Grid caches)
- **StateTransferManager** (Data Grid 14.0 API)

4.4. MONITORING THE STATUS OF CROSS-SITE REPLICATION

Monitor the site status of your backup locations to detect interruptions in the communication between the sites. When a remote site status changes to **offline**, Data Grid stops replicating your data to the backup location. Your data become out of sync and you must fix the inconsistencies before bringing the clusters back online.

Monitoring cross-site events is necessary for early problem detection. Use one of the following monitoring strategies:

- Monitoring cross-site replication with the REST API
- Monitoring cross-site replication with the Prometheus metrics or any other monitoring system

Monitoring cross-site replication with the REST API

Monitor the status of cross-site replication for all caches using the REST endpoint. You can implement a custom script to poll the REST endpoint or use the following example.

Prerequisites

- Enable cross-site replication.

Procedure

1. Implement a script to poll the REST endpoint.
   The following example demonstrates how you can use a Python script to poll the site status every five seconds.

```
#!/usr/bin/python3
import time
import requests
from requests.auth import HTTPDigestAuth


class InfinispanConnection:

    def __init__(self, server: str = 'http://localhost:11222', cache_manager: str = 'default',
                 auth: tuple = ('admin', 'change_me')) -> None:
        super().__init__()
        self.__url = f'{server}/rest/v2/cache-managers/{cache_manager}/x-site/backups/'
        self.__auth = auth
        self.__headers = {
            'accept': 'application/json'
        }

    def get_sites_status(self):
        try:
            rsp = requests.get(self.__url, headers=self.__headers, auth=HTTPDigestAuth(self.__auth[0],
self.__auth[1]))
            if rsp.status_code != 200:
                return None
            return rsp.json()
        except:
            return None


# Specify credentials for Data Grid user with permission to access the REST endpoint
USERNAME = 'admin'
PASSWORD = 'change_me'
# Set an interval between cross-site status checks
POLL_INTERVAL_SEC = 5
# Provide a list of servers
SERVERS = [
    InfinispanConnection('http://127.0.0.1:11222', auth=(USERNAME, PASSWORD)),
    InfinispanConnection('http://127.0.0.1:12222', auth=(USERNAME, PASSWORD))
]
#Specify the names of remote sites
REMOTE_SITES = [
    'nyc'
]
```

```
#Provide a list of caches to monitor
CACHES = [
   'work',
   'sessions'
]


def on_event(site: str, cache: str, old_status: str, new_status: str):
   # TODO implement your handling code here
   print(f'site={site} cache={cache} Status changed {old_status} -> {new_status}')


def __handle_mixed_state(state: dict, site: str, site_status: dict):
   if site not in state:
      state[site] = {c: 'online' if c in site_status['online'] else 'offline' for c in CACHES}
      return

   for cache in CACHES:
      __update_cache_state(state, site, cache, 'online' if cache in site_status['online'] else 'offline')


def __handle_online_or_offline_state(state: dict, site: str, new_status: str):
   if site not in state:
      state[site] = {c: new_status for c in CACHES}
      return

   for cache in CACHES:
      __update_cache_state(state, site, cache, new_status)


def __update_cache_state(state: dict, site: str, cache: str, new_status: str):
   old_status = state[site].get(cache)
   if old_status != new_status:
      on_event(site, cache, old_status, new_status)
      state[site][cache] = new_status


def update_state(state: dict):
   rsp = None
   for conn in SERVERS:
      rsp = conn.get_sites_status()
      if rsp:
         break
   if rsp is None:
      print('Unable to fetch site status from any server')
      return

   for site in REMOTE_SITES:
      site_status = rsp.get(site, {})
      new_status = site_status.get('status')
      if new_status == 'mixed':
         __handle_mixed_state(state, site, site_status)
      else:
         __handle_online_or_offline_state(state, site, new_status)
```

```
if __name__ == '__main__':
    _state = {}
    while True:
        update_state(_state)
        time.sleep(POLL_INTERVAL_SEC)
```

When a site status changes from **online** to **offline** or vice-versa, the function **on_event** is invoked.

If you want to use this script, you must specify the following variables:

- **USERNAME** and **PASSWORD**: The username and password of Data Grid user with permission to access the REST endpoint.
- **POLL_INTERVAL_SEC**: The number of seconds between polls.
- **SERVERS**: The list of Data Grid Servers at this site. The script only requires a single valid response but the list is provided to allow fail over.
- **REMOTE_SITES**: The list of remote sites to monitor on these servers.
- **CACHES**: The list of cache names to monitor.

Additional resources

- REST API: Getting status of backup locations

Monitoring cross-site replication with the Prometheus metrics

Prometheus, and other monitoring systems, let you configure alerts to detect when a site status changes to **offline**.

TIP

Monitoring cross-site latency metrics can help you to discover potential issues.

Prerequisites

- Enable cross-site replication.

Procedure

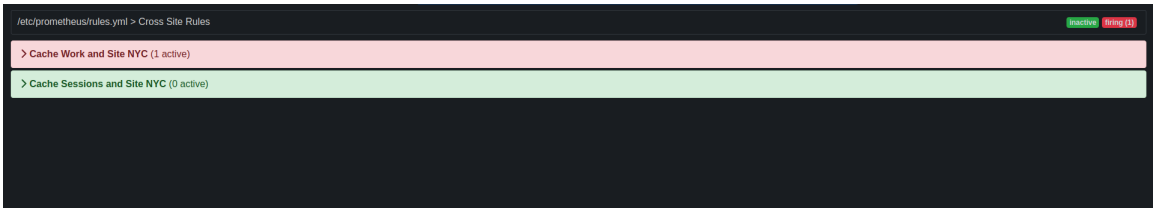
1. Configure Data Grid metrics.
2. Configure alerting rules using the Prometheus metrics format.
   - For the site status, use **1** for **online** and **0** for **offline**.
   - For the **expr** filed, use the following format:
     vendor_cache_manager_default_cache_<cache name>_x_site_admin_<site name>_status.
     In the following example, Prometheus alerts you when the **NYC** site gets **offline** for cache named **work** or **sessions**.

     ```
     groups:
     - name: Cross Site Rules
       rules:
     ```

```
- alert: Cache Work and Site NYC
  expr: vendor_cache_manager_default_cache_work_x_site_admin_nyc_status == 0
- alert: Cache Sessions and Site NYC
  expr: vendor_cache_manager_default_cache_sessions_x_site_admin_nyc_status ==
0
```

The following image shows an alert that the **NYC** site is **offline** for cache **work**.

Figure 4.1. Prometheus Alert



Additional resources

- Configuring Data Grid metrics
- Prometheus Alerting Overview
- Grafana Alerting Documentation
- Openshift Managing Alerts

CHAPTER 5. CONFIGURING JVM MEMORY USAGE

Control how Data Grid 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 Data Grid to store data in off-heap, native memory.

5.1. DEFAULT MEMORY CONFIGURATION

By default Data Grid 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 Data Grid is not the primary data store, then entries become out of date which means your caches contain stale data.

XML

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

JSON

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

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 Data Grid 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. Data Grid 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 Data Grid nodes.

✓ Expiration takes place across Data Grid 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 DATA GRID 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.

IMPORTANT

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

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

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

5.3.1. Eviction strategies

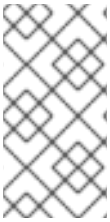
When you configure Data Grid eviction you specify:

- 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 Data Grid 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.



NOTE

Data Grid 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, Data Grid uses a custom implementation of the Least Recently Used (LRU) algorithm.

Additional resources

- Caffeine
- Data Grid configuration schema reference

5.3.2. Configuring maximum count eviction

Limit the size of Data Grid caches to a total number of entries.

Procedure

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

Maximum count eviction
In the following example, Data Grid 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

- Data Grid configuration schema reference
- org.infinispan.configuration.cache.MemoryConfigurationBuilder

5.3.3. Configuring maximum size eviction

Limit the size of Data Grid caches to a maximum amount of memory.

Procedure

1. Open your Data Grid 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 Data Grid 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 Data Grid removes entries with either the **when-full** attribute or **whenFull()** method.
   - **REMOVE** Data Grid performs eviction. This is the default strategy.
   - **MANUAL** You perform eviction manually for embedded caches.
   - **EXCEPTION** Data Grid throws an exception instead of evicting entries.
6. Save and close your Data Grid configuration.

Maximum size eviction
In the following example, Data Grid 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);
```

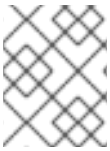
Additional resources

- Data Grid 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, Data Grid 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 Data Grid with the **REMOVE** or **EXCEPTION** eviction strategy.



NOTE

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

XML

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

JSON

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

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 Data Grid 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 Data Grid to remove entries from caches when they reach one of the following time limits:

Lifespan

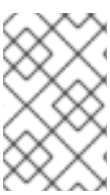
Sets the maximum amount of time that entries can exist.

Maximum idle

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

IMPORTANT

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



NOTE

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, Data Grid 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.

Data Grid 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 Data Grid 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, Data Grid detects that the entry is expired and removes it from the data container. The **Cache.get(k1)** request returns **null**.

Data Grid 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, Data Grid cannot expire entries because loaders can only read from external storage.

NOTE

Data Grid 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

Data Grid 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 Data Grid **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.

IMPORTANT

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

Data Grid 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, Data Grid sends touch commands to all owners when keys are accessed.

The touch commands that Data Grid 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 Data Grid uses for eviction.
- With scattered cache mode, Data Grid 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 Data Grid 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 Data Grid configuration.

Expiration for Data Grid caches

In the following example, Data Grid 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

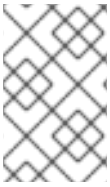
```
replicatedCache:
  expiration:
    lifespan: "5000"
    maxIdle: "1000"
```

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.



NOTE

When you explicitly define lifespan and maximum idle time values for cache entries, Data Grid replicates those values across the cluster along with the cache entries. Likewise, Data Grid 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 Data Grid Console.
  With the Data Grid 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

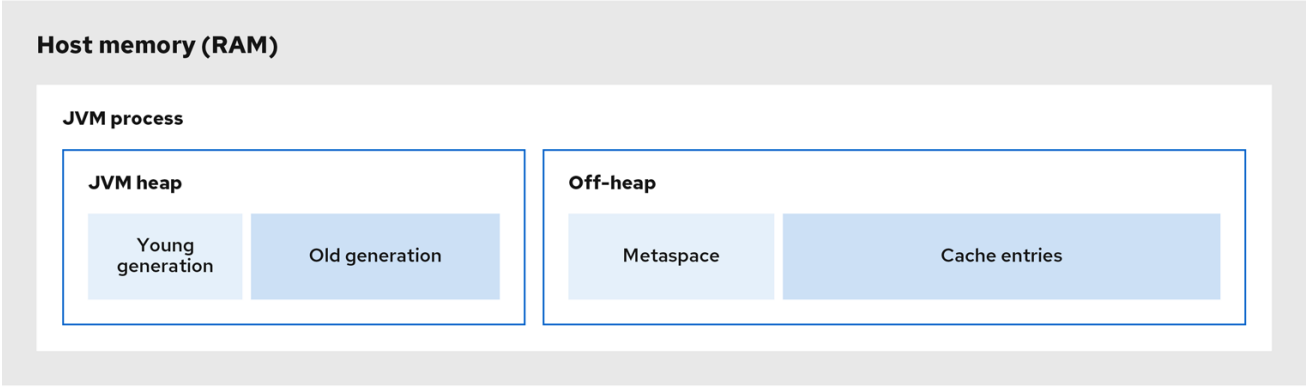
- org.infinispan.configuration.cache.ExpirationConfigurationBuilder
- org.infinispan.expiration.ExpirationManager

5.5. JVM HEAP AND OFF-HEAP MEMORY

Data Grid stores cache entries in JVM heap memory by default. You can configure Data Grid 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 Data Grid is running:

Figure 5.1. 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 Data Grid 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 Data Grid cache entries.

Off-heap memory:

- Uses less memory per entry.

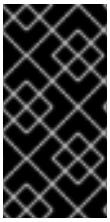
- Improves overall JVM performance by avoiding Garbage Collector (GC) runs.

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, Data Grid dynamically allocates native memory to your data.

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



IMPORTANT

Even though Data Grid 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

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

Data consistency

Data Grid 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 Data Grid to store cache entries in native memory outside the JVM heap space.

Procedure

1. Open your Data Grid 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 Data Grid configuration.

Off-heap storage

Data Grid stores cache entries as bytes in native memory. Eviction happens when there are 100 entries in the data container and Data Grid 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" : "OFF_HEAP",
      "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

- Data Grid configuration schema reference
- org.infinispan.configuration.cache.MemoryConfigurationBuilder

CHAPTER 6. CONFIGURING PERSISTENT STORAGE

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

Data overflow

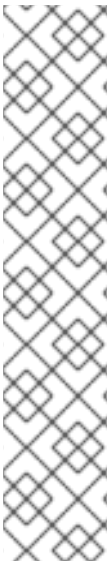
Using eviction and passivation techniques ensures that Data Grid keeps only frequently used data in-memory and writes older entries to persistent storage.

6.1. PASSIVATION

Passivation configures Data Grid to write entries to cache stores when it evicts those entries from memory. In this way, passivation 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 Data Grid evicts an entry from the cache, it notifies cache listeners that the entry is passivated then stores the entry in the cache store. When Data Grid 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 while keeping the value still in the store.



NOTE

- Passivation uses the first cache loader in the Data Grid 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 Data Grid 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, Data Grid throws an exception.

6.1.1. How passivation works

Passivation disabled

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

If Data Grid 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. This is recommended when you require highest consistency as the store will be able to be read again after a crash.

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

Passivation enabled

Data Grid adds data to persistent storage only when it evicts data from memory, an entry is removed or upon shutting down the node.

When Data Grid activates entries, it restores data in memory but keeps the data in the store still. This allows for writes to be just as fast as without a store, and still maintains consistency. When an entry is created or updated only the in memory will be updated and thus the store will be outdated for the time being.

NOTE

Passivation is not supported when a store is also configured as shared. This is due to entries can become out of sync between nodes depending on when a write is evicted versus read.

To gurantee data consistency any store that is not shared should always have **purgeOnStartup** enabled. This is true for both passivation enabled or disabled since a store could hold an outdated entry while down and resurrect it at a later point.

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

Operation	Passivation disabled	Passivation enabled
Insert k1.	**Memory:** k1 **Disk:** k1	**Memory:** k1 **Disk:** -
Insert k2.	**Memory:** k1, k2 **Disk:** k1, k2	**Memory:** k1, k2 **Disk:** -
Eviction thread runs and evicts k1.	**Memory:** k2 **Disk:** k1, k2	**Memory:** k2 **Disk:** k1
Read k1.	**Memory:** k1, k2 **Disk:** k1, k2	**Memory:** k1, k2 **Disk:** k1
Eviction thread runs and evicts k2.	**Memory:** k1 **Disk:** k1, k2	**Memory:** k1 **Disk:** k1, k2
Remove k2.	**Memory:** k1 **Disk:** k1	**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()**,

Data Grid 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

Data Grid 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>
      <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, Data Grid adds those operations to a modification queue. Data Grid 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, Data Grid adds those additional operations to the queue. However, those operations do not complete until Data Grid 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 Data Grid 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

```
<distributed-cache>
  <persistence>
```

```
    <table-jdbc-store xmlns="urn:infinispan:config:store:sql:14.0"
                      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

```
{
  "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

```
distributedCache:
  persistence:
    tableJdbcStore:
      dialect: "H2"
      shared: "true"
      tableName: "books"
      connectionPool:
        connectionUrl: "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 Data Grid logs WARN messages and rejects write operations.
- If **fail-silently="false"** then Data Grid throws exceptions if it detects the cache store is unavailable during a write operation. Likewise if the modification queue becomes full, Data Grid throws an exception.
  In some cases, data loss can occur if Data Grid 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 Data Grid 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.

IMPORTANT

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

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

6.5. SHARED CACHE STORES

Data Grid 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 should use "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.

IMPORTANT

Purging deletes data, which is not typically the desired behavior for persistent storage.

Local cache store

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

Shared cache store

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

Additional resources

- Data Grid Configuration Schema

6.6. TRANSACTIONS WITH PERSISTENT CACHE STORES

Data Grid 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, Data Grid 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

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

Remote caches

Data Grid Server saves cluster state to the **$RHDG_HOME/server/data** directory.

IMPORTANT

You should never delete or modify the **server/data** directory or its content. Data Grid 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

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

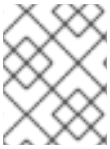
For clustered embedded caches, such as replicated or distributed, you should always enable and configure a global persistent location to restore cluster topology.

You should never configure an absolute path for a file-based cache store that is outside the global persistent location. If you do, Data Grid 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 Data Grid stores global state for clustered embedded caches.



NOTE

Data Grid 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 Data Grid to your project.

Procedure

1. Enable global state in one of the following ways:
   - Add the **global-state** element to your Data Grid 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 Data Grid 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 Data Grid uses **/opt/data/global/state** as the global persistent location.

Global persistent location configuration

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

- Data Grid 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 Data Grid is running. For clustered caches, file-based cache stores are unique to each Data Grid node.

WARNING

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

NOTE

Using SoftIndexFileStore in a clustered cache should enable purge on startup to ensure stale entries are not resurrected.

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.

SoftIndexFileStore creates a B+ tree per configured cache segment. This provides an additional "index" as it only has so many elements and provides additional parallelism for index updates. Currently we allow for a parallel amount based on one sixteenth of the number of cache segments.

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.

File limits

SoftIndexFileStore will use two plus the configured openFilesLimit amount of files at a given time. The two additional file pointers are reserved for the log appender for newly updated data and another for the compactor which writes compacted entries into a new file.

The amount of open allocated files allocated for indexing is one tenth of the total number of the configured openFilesLimit. This number has a minimum of 1 or the number of cache segments. Any number remaning from configured limit is allocated for open data files themselves.

Segmentation

Soft-index file stores are always segmented. The append log(s) are not directly segmented and segmentation is handled directly by the index.

Expiration

The SoftIndexFileStore has full support for expired entries and their requirements.

Single File Cache Stores

NOTE

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

Single File cache stores, **SingleFileStore**, persist data to file. Data Grid 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 Data Grid 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 Data Grid 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 Data Grid 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

```
<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

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

YAML

```
distributedCache:
  persistence:
    passivation: "true"
    fileStore:
      shared: "false"
      data:
        path: "data"
      index:
        path: "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 Data Grid to create single file stores.

IMPORTANT

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"/>
  </persistence>
</distributed-cache>
```

JSON

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

YAML

```
distributedCache:
  persistence:
    passivation: "true"
    singleFileStore:
      shared: "false"
      preload: "true"
```

ConfigurationBuilder

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

6.9. JDBC CONNECTION FACTORIES

Data Grid 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 Data Grid 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

```
distributedCache:
  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

```
distributedCache:
  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

```
distributedCache:
  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 managed datasources

Create managed datasources as part of your Data Grid Server configuration to optimize connection pooling and performance for JDBC database connections. You can then specify the JDNI name of the managed datasources in your caches, which centralizes JDBC connection configuration for your deployment.

Prerequisites

- Copy database drivers to the **server/lib** directory in your Data Grid Server installation.

TIP

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

```
install org.postgresql:postgresql:42.4.3
```

Procedure

1. Open your Data Grid 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.

   TIP

   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 Data Grid Server nodes pool and reuse connections with connection pool tuning properties in the **connection-pool** section.
8. Save the changes to your configuration.

Verification

Use the Data Grid 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.

```
server datasource ls
```

3. Test a datasource connection.

```
server datasource test my-datasource
```

Managed datasource configuration

XML

```
<server xmlns="urn:infinispan:server:14.0">
  <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>
</server>
```

JSON

```
{
  "server": {
    "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

```
server:
  dataSources:
    - name: ds
      jndiName: 'jdbc/postgres'
      statistics: true
      connectionFactory:
        driver: "org.postgresql.Driver"
        url: "jdbc:postgresql://localhost:5432/postgres"
        username: "postgres"
        password: "changeme"
        connectionProperties:
          name: value
      connectionPool:
        initialSize: 1
        maxSize: 10
        minSize: 3
        backgroundValidation: 1000
        idleRemoval: 1
        blockingTimeout: 1000
        leakDetection: 10000
```

6.9.1.1. Configuring caches with JNDI names

When you add a managed datasource to Data Grid Server you can add the JNDI name to a JDBC-based cache store configuration.

Prerequisites

- Configure Data Grid Server with a managed datasource.

Procedure

1. Open your cache configuration for editing.
2. Add the **data-source** element or field to the JDBC-based cache store configuration.
3. Specify the JNDI name of the managed datasource as the value of the **jndi-url** attribute.
4. Configure the JDBC-based cache stores as appropriate.
5. Save the changes to your configuration.

JNDI name in cache configuration

XML

```
<distributed-cache>
 <persistence>
  <jdbc:string-keyed-jdbc-store>
   <!-- Specifies the JNDI name of a managed datasource on Data Grid Server. -->
   <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>
```

JSON

```
{
 "distributed-cache": {
  "persistence": {
   "string-keyed-jdbc-store": {
    "data-source": {
     "jndi-url": "jdbc/postgres"
     },
    "string-keyed-table": {
     "prefix": "TBL",
     "drop-on-exit": true,
     "create-on-start": true,
     "id-column": {
      "name": "ID",
      "type": "VARCHAR(255)"
     },
     "data-column": {
      "name": "DATA",
      "type": "BYTEA"
     },
     "timestamp-column": {
      "name": "TS",
      "type": "BIGINT"
     },
     "segment-column": {
      "name": "S",
      "type": "INT"
     }
    }
   }
  }
 }
}
```

YAML

```
distributedCache:
  persistence:
    stringKeyedJdbcStore:
      dataSource:
        jndi-url: "jdbc/postgres"
      stringKeyedTable:
        prefix: "TBL"
        dropOnExit: true
        createOnStart: true
        idColumn:
          name: "ID"
          type: "VARCHAR(255)"
        dataColumn:
          name: "DATA"
          type: "BYTEA"
        timestampColumn:
          name: "TS"
          type: "BIGINT"
        segmentColumn:
          name: "S"
          type: "INT"
```

6.9.1.2. Connection pool tuning properties

You can tune JDBC connection pools for managed datasources in your Data Grid Server configuration.

Property	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.

Property	Description
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_DELAY
=-1
org.infinispan.agroal.driverClassName=org.h2.Driver.class
org.infinispan.agroal.principal=sa
org.infinispan.agroal.credential=sa
```

2. Configure Data Grid 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 Data Grid caches from existing database tables. Data Grid offers two types of SQL cache store:

Table

Data Grid loads entries from a single database table.

Query

Data Grid 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.

TIP

Visit the code tutorials to try a SQL cache store in action. See the Persistence code tutorial with remote caches.

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 Data Grid with Protobuf schema (**.proto** files) that describe the keys and values. With Data Grid Server you can add schema through the Data Grid Console or Command Line Interface (CLI) with the **schema** command.

WARNING

The SQL cache store is intended for use with an existing database table. As a result, it does not store any metadata, which includes expiration, segments, and, versioning metadata. Due to the absence of version storage, SQL store does not support optimistic transactional caching and asynchronous cross-site replication. This limitation also extends to Hot Rod versioned operations.

TIP

Use expiration with the SQL cache store when it is configured as read only. Expiration removes stale values from memory, causing the cache to fetch the values from the database again and cache them anew.

Additional resources

- DatabaseType Enum lists supported database dialects
- Data Grid SQL store configuration reference

6.10.1. Data types for keys and values

Data Grid loads keys and values from columns in database tables via SQL cache stores, automatically using the appropriate data types. The following **CREATE** statement adds a table named "books" that has two columns, **isbn** and **title**:

Database table with two columns

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

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

Additional resources

- Data Grid SQL store configuration reference

6.10.1.1. Composite keys and values

You can use SQL stores with database tables that contain composite primary keys or composite values.

To use composite keys or values, you must provide Data Grid with Protobuf schema that describe the data types. You must also add **schema** configuration to your SQL store and specify the message names for keys and values.

TIP

Data Grid recommends generating Protobuf schema with the ProtoStream processor. You can then upload your Protobuf schema for remote caches through the Data Grid Console, CLI, or REST API.

Composite values

The following database table holds a composite value of the **title** and **author** columns:

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

Data Grid adds an entry to the cache using the **isbn** column as the key. For the value, Data Grid requires a Protobuf schema that maps the **title** column and the **author** columns:

```
package library;

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

Composite keys and values

The following database table holds a composite primary key and a composite value, with two columns each:

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

For both the key and the value, Data Grid requires a Protobuf schema that maps the columns to keys and values:

```
package library;

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

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

Additional resources

- Cache encoding and marshalling: Generate Protobuf schema and register them with Data Grid
- Data Grid SQL store configuration reference

6.10.1.2. Embedded keys

Protobuf schema can include keys within values, as in the following example:

Protobuf schema with an embedded key

```
package library;

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

message books_value {
   required string isbn = 1;
   required string reprint = 2;
   optional string title = 3;
   optional string author = 4;
}
```

To use embedded keys, you must include the **embedded-key="true"** attribute or **embeddedKey(true)** method in your SQL store configuration.

6.10.1.3. 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**

SQL type	Protobuf type
bytea, **tinyblob**, **blob**, **mediumblob**, **longblob**	**bytes**

Additional resources

- Cache encoding and marshalling

6.10.2. Loading Data Grid caches from database tables

Add a SQL table cache store to your configuration if you want Data Grid to load data from a database table. When it connects to the database, Data Grid uses metadata from the table to detect column names and data types. Data Grid 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 Data Grid Server configuration and specify the JNDI name in the cache configuration.
- Generate Protobuf schema for any composite keys or composite values and register your schemas with Data Grid.

  TIP

  Data Grid recommends generating Protobuf schema with the ProtoStream processor. For remote caches, you can register your schemas by adding them through the Data Grid Console, CLI, or REST API.

Procedure

1. Add database drivers to your Data Grid deployment.
   - Remote caches: Copy database drivers to the **server/lib** directory in your Data Grid Server installation.

     TIP

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

     ```
     install org.postgresql:postgresql:42.4.3
     ```

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

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

2. Open your Data Grid configuration for editing.
3. Add a SQL table cache store.

   Declarative

   ```
   table-jdbc-store xmlns="urn:infinispan:config:store:sql:14.0"
   ```

   Programmatic

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

4. Specify the database dialect with either **dialect=""** or **dialect()**, for example **dialect="H2"** or **dialect="postgres"**.
5. 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)**.
6. Name the database table that loads the cache with **table-name="<database_table_name>"** or **table.name("<database_table_name>")**.
7. Add the **schema** element or the **.schemaJdbcConfigurationBuilder()** method and add Protobuf schema configuration for composite keys or values.
   a. Specify the package name with the **package** attribute or **package()** method.
   b. Specify composite values with the **message-name** attribute or **messageName()** method.
   c. Specify composite keys with the **key-message-name** attribute or **keyMessageName()** method.
   d. Set a value of **true** for the **embedded-key** attribute or **embeddedKey()** method if your schema includes keys within values.
8. Save the changes to your configuration.

SQL table store configuration

The following example loads a distributed cache from a database table named "books" using composite values defined in a Protobuf schema:

XML

```
<distributed-cache>
  <persistence>
    <table-jdbc-store xmlns="urn:infinispan:config:store:sql:14.0"
                      dialect="H2"
                      shared="true"
                      table-name="books">
      <schema message-name="books_value"
              package="library"/>
```

```
    </table-jdbc-store>
  </persistence>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "persistence": {
      "table-jdbc-store": {
        "dialect": "H2",
        "shared": "true",
        "table-name": "books",
        "schema": {
          "message-name": "books_value",
          "package": "library"
        }
      }
    }
  }
}
```

YAML

```
distributedCache:
  persistence:
    tableJdbcStore:
      dialect: "H2"
      shared: "true"
      tableName: "books"
      schema:
        messageName: "books_value"
        package: "library"
```

ConfigurationBuilder

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

Additional resources

- Cache encoding and marshalling: Generate Protobuf schema and register them with Data Grid
- Persistence code tutorial with remote caches
- JDBC connection factories

- DatabaseType Enum lists supported database dialects
- Data Grid SQL store configuration reference

6.10.3. Using SQL queries to load data and perform operations

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 Data Grid Server configuration and specify the JNDI name in the cache configuration.
- Generate Protobuf schema for any composite keys or composite values and register your schemas with Data Grid.

  TIP

  Data Grid recommends generating Protobuf schema with the ProtoStream processor. For remote caches, you can register your schemas by adding them through the Data Grid Console, CLI, or REST API.

Procedure

1. Add database drivers to your Data Grid deployment.
   - Remote caches: Copy database drivers to the **server/lib** directory in your Data Grid Server installation.

     TIP

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

     ```
     install org.postgresql:postgresql:42.4.3
     ```

   - 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>
     ```

2. Open your Data Grid configuration for editing.
3. Add a SQL query cache store.

   Declarative

```
query-jdbc-store xmlns="urn:infinispan:config:store:sql:14.0"
```

Programmatic

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

4. Specify the database dialect with either **dialect=""** or **dialect()**, for example **dialect="H2"** or **dialect="postgres"**.
5. 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)**.
6. 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.

NOTE

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.

7. Add the **schema** element or the **.schemaJdbcConfigurationBuilder()** method and add Protobuf schema configuration for composite keys or values.
   a. Specify the package name with the **package** attribute or **package()** method.
   b. Specify composite values with the **message-name** attribute or **messageName()** method.
   c. Specify composite keys with the **key-message-name** attribute or **keyMessageName()** method.
   d. Set a value of **true** for the **embedded-key** attribute or **embeddedKey()** method if your schema includes keys within values.
8. Save the changes to your configuration.

Additional resources

- Cache encoding and marshalling: Generate Protobuf schema and register them with Data Grid
- Persistence code tutorial with remote caches
- JDBC connection factories
- DatabaseType Enum lists supported database dialects
- Data Grid SQL store configuration reference

6.10.3.1. SQL query store configuration

This section provides an example configuration for a SQL query cache store that loads a distributed cache with data from two database tables: "person" and "address".

SQL statements
The following examples show SQL data definition language (DDL) statements for the "person" and "address" tables. The data types described in the example are only valid for PostgreSQL database.

SQL statement for the "person" table

```
CREATE TABLE Person (
```

```
  name VARCHAR(255) NOT NULL,
  picture BYTEA,
  sex VARCHAR(255),
  birthdate TIMESTAMP,
  accepted_tos BOOLEAN,
  notused VARCHAR(255),
  PRIMARY KEY (name)
);
```

SQL statement for the "address" table

```
CREATE TABLE Address (
  name VARCHAR(255) NOT NULL,
  street VARCHAR(255),
  city VARCHAR(255),
  zip INT,
  PRIMARY KEY (name)
);
```

Protobuf schemas

Protobuf schema for the "person" and "address" tables are as follows:

Protobuf schema for the "address" table

```
package com.example;

message Address {
   optional string street = 1;
   optional string city = 2 [default = "San Jose"];
   optional int32 zip = 3 [default = 0];
}
```

Protobuf schema for the "person" table

```
package com.example;

import "/path/to/address.proto";

enum Sex {
   FEMALE = 1;
   MALE = 2;
}

message Person {
   optional string name = 1;
   optional Address address = 2;
   optional bytes picture = 3;
   optional Sex sex = 4;
   optional fixed64 birthDate = 5 [default = 0];
   optional bool accepted_tos = 6 [default = false];
}
```

Cache configuration

The following example loads a distributed cache from the "person" and "address" tables using a SQL query that includes a **JOIN** clause:

XML

```
<distributed-cache>
  <persistence>
    <query-jdbc-store xmlns="urn:infinispan:config:store:sql:14.0"
                      dialect="POSTGRES"
                      shared="true" key-columns="name">
          <connection-pool driver="org.postgresql.Driver"
                           connection-url="jdbc:postgresql://localhost:5432/postgres"
                           username="postgres"
                           password="changeme"/>
      <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 = :name AND t2.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="Person"
               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": "name",
        "connection-pool": {
          "username": "postgres",
          "password": "changeme",
          "driver": "org.postgresql.Driver",
          "connection-url": "jdbc:postgresql://localhost:5432/postgres"
        },
        "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 = :name AND t2.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": "Person",
        "package": "com.example",
        "embedded-key": "true"
       }
     }
   }
  }
}
```

YAML

```
distributedCache:
  persistence:
    queryJdbcStore:
      dialect: "POSTGRES"
      shared: "true"
      keyColumns: "name"
      connectionPool:
        username: "postgres"
        password: "changeme"
        driver: "org.postgresql.Driver"
        connectionUrl: "jdbc:postgresql://localhost:5432/postgres"
      queries:
        selectSingle: "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 = :name AND t2.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"
        deleteSingle: "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"
      schema:
        messageName: "Person"
        package: "com.example"
        embeddedKey: "true"
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence().addStore(QueriesJdbcStoreConfigurationBuilder.class)
      .dialect(DatabaseType.POSTGRES)
      .shared("true")
```

```
      .keyColumns("name")
      .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 = :name AND t2.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("Person")
        .packageName("com.example")
        .embeddedKey(true);
```

Additional resources

- Data Grid SQL store configuration reference

6.10.4. 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
```

Data Grid 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.
  Data Grid 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 Data Grid version and configuration.

NOTE

By default Data Grid 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 Data Grid 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 Data Grid 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:14.0"
     ```

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

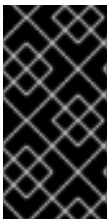
     ```
     persistence().addStore(JdbcStringBasedStoreConfigurationBuilder.class)
     ```

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 Data Grid can connect to the database.

5. Add a database table that stores cache entries.



IMPORTANT

Configuring the **string-keyed-jdbc-store** with inappropriate data type can lead to exceptions during loading or storing cache entries. For more information and a list of data types that are tested as part of the Data Grid release, see Tested database settings for Data Grid string-keyed-jdbc-store persistence (Login required).

JDBC string-based cache store configuration

XML

```
<distributed-cache>
  <persistence>
    <string-keyed-jdbc-store xmlns="urn:infinispan:config:store:jdbc:14.0"
                             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

```
distributedCache:
  persistence:
    stringKeyedJdbcStore:
      dialect: "H2"
      stringKeyedTable:
        prefix: "ISPN_STRING_TABLE"
        createOnStart: true
        idColumn:
          name: "ID_COLUMN"
          type: "VARCHAR(255)"
        dataColumn:
          name: "DATA_COLUMN"
          type: "BINARY"
        timestampColumn:
          name: "TIMESTAMP_COLUMN"
          type: "BIGINT"
        segmentColumn:
          name: "SEGMENT_COLUMN"
          type: "INT"
      connectionPool:
        connectionUrl: "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 Data Grid expires from memory.

Table 6.1. 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, Data Grid flushes the expired into the RocksDB cache store.

Parameter	Description
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:kSnappyCo
mpression: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.

NOTE

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:14.0"
                   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

```
localCache:
  persistence:
    rocksdbStore:
      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

Reference

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

6.13. REMOTE CACHE STORES

Remote cache stores, **RemoteStore**, use the Hot Rod protocol to store data on Data Grid clusters.

NOTE

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 Data Grid Hot Rod protocol version 2.3 or later.

WARNING

When you enable segmentation for **RemoteStore**, it uses the number of segments that you define in your Data Grid 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:14.0"
                  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

```
distributedCache:
  remoteStore:
    cache: "mycache"
    rawValues: "true"
    remoteServer:
      - host: "one"
        port: "12111"
      - host: "two"
    connectionPool:
      maxActive: "10"
      exhaustedAction: "CREATE_NEW"
```

ConfigurationBuilder

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence().addStore(RemoteStoreConfigurationBuilder.class)
      .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. CLUSTER CACHE LOADERS

ClusterCacheLoader retrieves data from other Data Grid cluster members but does not persist data. In other words, **ClusterCacheLoader** is not a cache store.

WARNING

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**).
- 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

```
distributedCache:
  persistence:
    clusterLoader:
      preload: "true"
      remoteTimeout: "500"
```

ConfigurationBuilder

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

Additional resources

- Data Grid configuration schema
- ClusterLoader
- ClusterLoaderConfiguration

6.15. CREATING CUSTOM CACHE STORE IMPLEMENTATIONS

You can create custom cache stores through the Data Grid persistent SPI.

6.15.1. Data Grid Persistence SPI

The Data Grid 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

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

Simplified transaction integration

Data Grid 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

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

Reduced serialization resulting in less CPU usage

Data Grid exposes stored entries in a serialized format that can be transmitted remotely. For this reason, Data Grid 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

6.15.2. Creating cache stores

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

Procedure

1. Implement the appropriate Data Grid 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.

NOTE

If a configuration does not declare the **@ConfigurationFor** annotation, a warning message is logged when Data Grid initializes the cache.

6.15.3. Examples of custom cache store configuration

The following are examples show how to configure Data Grid 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

```
distributedCache:
  persistence:
    store:
      class: "org.infinispan.persistence.example.MyInMemoryStore"
```

ConfigurationBuilder

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

6.15.4. Deploying custom cache stores

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

Prerequisites

- Stop Data Grid Server if it is running.
  Data Grid 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 Data Grid Server installation.

6.16. MIGRATING DATA BETWEEN CACHE STORES

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

6.16.1. Cache store migrator

Data Grid provides the **StoreMigrator.java** utility that recreates data for the latest Data Grid cache store implementations.

StoreMigrator takes a cache store from a previous version of Data Grid 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.

IMPORTANT

StoreMigrator cannot migrate data from segmented cache stores to:

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

6.16.2. Getting the cache store migrator

StoreMigrator is available as part of the Data Grid 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>
```

6.16.3. Configuring the cache store migrator

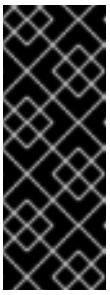
Use the **migrator.properties** file to configure properties for source and target cache stores.

Procedure

1. Create a **migrator.properties** file.
2. Configure properties for source and target cache store using the **migrator.properties** file.
   a. Add the **source.** prefix to all configuration properties for the source cache store.

   Example source cache store

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

   IMPORTANT

   For migrating data from segmented cache stores, you must also configure the number of segments using the **source.segment_count** property. The number of segments must match **clustering.hash.numSegments** in your Data Grid configuration. If the number of segments for a cache store does not match the number of segments for the corresponding cache, Data Grid cannot read data from the cache store.

   b. Add the **target.** prefix to all configuration properties for the target cache store.

   Example target cache store

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

6.16.3.1. Configuration properties for the cache store migrator

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

Table 6.2. Cache Store Type Property

Property	Description	Required/Optional
type	Specifies the type of cache store for a source or target cache store. **.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

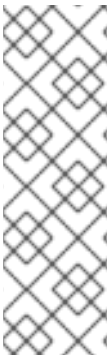
Table 6.3. Common Properties

Property	Description	Example Value	Required/Optional
cache_name	The name of the cache that you want to back up.	**.cache_name=myCache**	Required
segment_count	The number of segments for target cache stores that can use segmentation. The number of segments must match **clustering.hash.numSegments** in the Data Grid configuration. If the number of segments for a cache store does not match the number of segments for the corresponding cache, Data Grid cannot read data from the cache store.	**.segment_count=256**	Optional

Table 6.4. JDBC Properties

Property	Description	Required/Optional
dialect	Specifies the dialect of the underlying database.	Required
version	Specifies the marshaller version for source cache stores. Set one of the following values: * **8** for Data Grid 7.2.x * **9** for Data Grid 7.3.x * **10** for Data Grid 8.0.x * **11** for Data Grid 8.1.x * **12** for Data Grid 8.2.x * **13** for Data Grid 8.3.x	Required for source stores only.
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
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.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

Property	Description	Required/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



NOTE

To migrate from Binary cache stores in older Data Grid 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.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 6.5. RocksDB Properties

Property	Description	Required/Optional
location	Sets the database directory.	Required

Property	Description	Required/Optional
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 6.6. 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 6.7. 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
```

6.16.4. Migrating Data Grid cache stores

You can use the **StoreMigrator** to migrate data between cache stores with different Data Grid versions or to migrate data from one type of cache store to another.

Prerequisites

- Have a **infinispan-tools.jar**.
- Have the source and target cache store configured in the **migrator.properties** file.

Procedure

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

```
mvn exec:java
```

CHAPTER 7. CONFIGURING DATA GRID TO HANDLE NETWORK PARTITIONS

Data Grid clusters can split into network partitions in which subsets of nodes become isolated from each other. This condition results in loss of availability or consistency for clustered caches. Data Grid automatically detects crashed nodes and resolves conflicts to merge caches back together.

7.1. SPLIT CLUSTERS AND NETWORK PARTITIONS

Network partitions are the result of error conditions in the running environment, such as when a network router crashes. When a cluster splits into partitions, nodes create a JGroups cluster view that includes only the nodes in that partition. This condition means that nodes in one partition can operate independently of nodes in the other partition.

Detecting a split

To automatically detect network partitions, Data Grid uses the **FD_ALL** protocol in the default JGroups stack to determine when nodes leave the cluster abruptly.

NOTE

Data Grid cannot detect what causes nodes to leave abruptly. This can happen not only when there is a network failure but also for other reasons, such as when Garbage Collection (GC) pauses the JVM.

Data Grid suspects that nodes have crashed after the following number of milliseconds:

```
FD_ALL[2|3].timeout + FD_ALL[2|3].interval + VERIFY_SUSPECT[2].timeout +
GMS.view_ack_collection_timeout
```

When it detects that the cluster is split into network partitions, Data Grid uses a strategy for handling cache operations. Depending on your application requirements Data Grid can:

- Allow read and/or write operations for availability
- Deny read and write operations for consistency

Merging partitions together

To fix a split cluster, Data Grid merges the partitions back together. During the merge, Data Grid uses the **.equals()** method for values of cache entries to determine if any conflicts exist. To resolve any conflicts between replicas it finds on partitions, Data Grid uses a merge policy that you can configure.

7.1.1. Data consistency in a split cluster

Network outages or errors that cause Data Grid clusters to split into partitions can result in data loss or consistency issues regardless of any handling strategy or merge policy.

Between the split and detection

If a write operation takes place on a node that is in a minor partition when a split occurs, and before Data Grid detects the split, that value is lost when Data Grid transfers state to that minor partition during the merge.

In the event that all partitions are in the **DEGRADED** mode that value is not lost because no state

transfer occurs but the entry can have an inconsistent value. For transactional caches write operations that are in progress when the split occurs can be committed on some nodes and rolled back on other nodes, which also results in inconsistent values.

During the split and the time that Data Grid detects it, it is possible to get stale reads from a cache in a minor partition that has not yet entered **DEGRADED** mode.

During the merge

When Data Grid starts removing partitions nodes reconnect to the cluster with a series of merge events. Before this merge process completes it is possible that write operations on transactional caches succeed on some nodes but not others, which can potentially result in stale reads until the entries are updated.

7.2. CACHE AVAILABILITY AND DEGRADED MODE

To preserve data consistency, Data Grid can put caches into **DEGRADED** mode if you configure them to use either the **DENY_READ_WRITES** or **ALLOW_READS** partition handling strategy.

Data Grid puts caches in a partition into **DEGRADED** mode when the following conditions are true:

- At least one segment has lost all owners.
  This happens when a number of nodes equal to or greater than the number of owners for a distributed cache have left the cluster.

- There is not a majority of nodes in the partition.
  A majority of nodes is any number greater than half the total number of nodes in the cluster from the most recent stable topology, which was the last time a cluster rebalancing operation completed successfully.

When caches are in **DEGRADED** mode, Data Grid:

- Allows read and write operations only if all replicas of an entry reside in the same partition.

- Denies read and write operations and throws an **AvailabilityException** if the partition does not include all replicas of an entry.

NOTE

With the **ALLOW_READS** strategy, Data Grid allows read operations on caches in **DEGRADED** mode.

DEGRADED mode guarantees consistency by ensuring that write operations do not take place for the same key in different partitions. Additionally **DEGRADED** mode prevents stale read operations that happen when a key is updated in one partition but read in another partition.

If all partitions are in **DEGRADED** mode then the cache becomes available again after merge only if the cluster contains a majority of nodes from the most recent stable topology and there is at least one replica of each entry. When the cluster has at least one replica of each entry, no keys are lost and Data Grid can create new replicas based on the number of owners during cluster rebalancing.

In some cases a cache in one partition can remain available while entering **DEGRADED** mode in another partition. When this happens the available partition continues cache operations as normal and Data Grid attempts to rebalance data across those nodes. To merge the cache together Data Grid always transfers state from the available partition to the partition in **DEGRADED** mode.

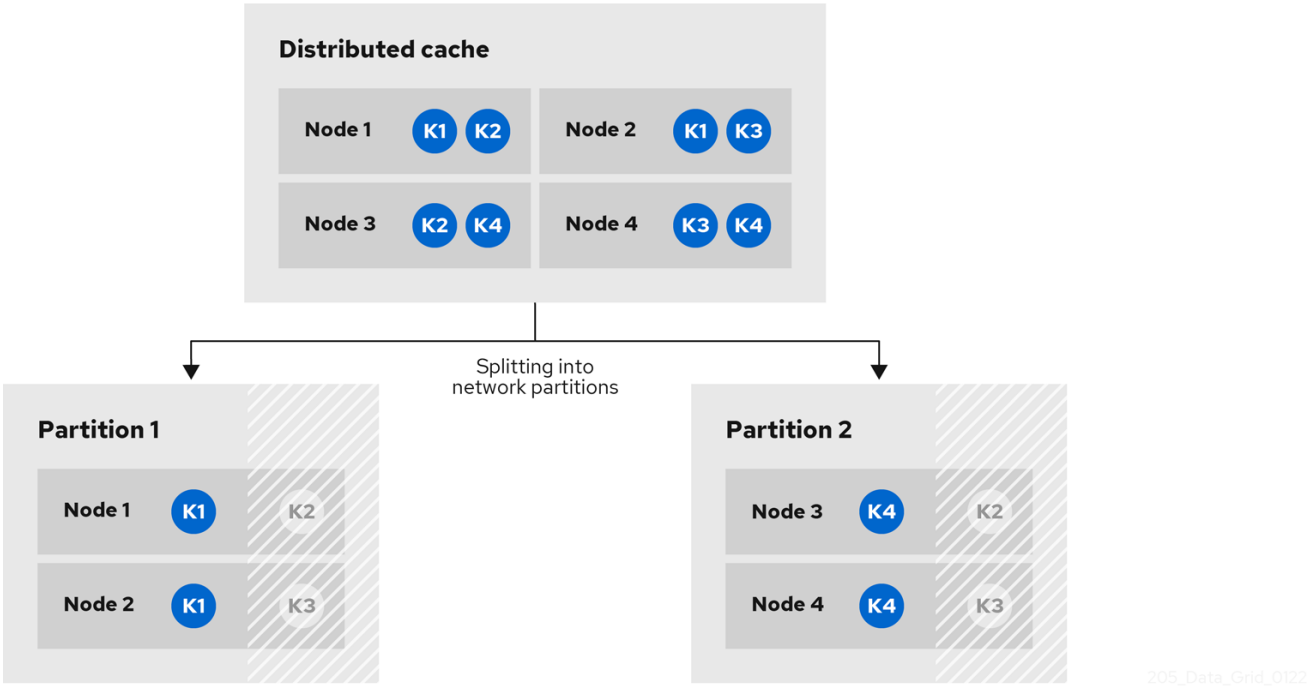
7.2.1. Degraded cache recovery example

This topic illustrates how Data Grid recovers from split clusters with caches that use the **DENY_READ_WRITES** partition handling strategy.

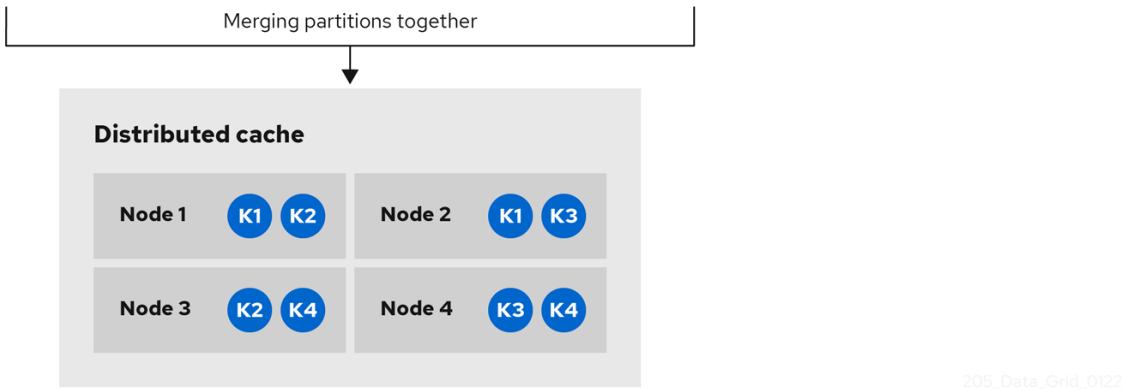
As an example, a Data Grid cluster has four nodes and includes a distributed cache with two replicas for each entry (**owners=2**). There are four entries in the cache, **k1**, **k2**, **k3** and **k4**.

With the **DENY_READ_WRITES** strategy, if the cluster splits into partitions, Data Grid allows cache operations only if all replicas of an entry are in the same partition.

In the following diagram, while the cache is split into partitions, Data Grid allows read and write operations for **k1** on partition 1 and **k4** on partition 2. Because there is only one replica for **k2** and **k3** on either partition 1 or partition 2, Data Grid denies read and write operations for those entries.



When network conditions allow the nodes to re-join the same cluster view, Data Grid merges the partitions without state transfer and restores normal cache operations.



7.2.2. Verifying cache availability during network partitions

Determine if caches on Data Grid clusters are in **AVAILABLE** mode or **DEGRADED** mode during a network partition.

When Data Grid clusters split into partitions, nodes in those partitions can enter **DEGRADED** mode to guarantee data consistency. In **DEGRADED** mode clusters do not allow cache operations resulting in loss of availability.

Procedure

Verify availability of clustered caches in network partitions in one of the following ways:

- Check Data Grid logs for **ISPN100011** messages that indicate if the cluster is available or if at least one cache is in **DEGRADED** mode.
- Get the availability of remote caches through the Data Grid Console or with the REST API.
  - Open the Data Grid Console in any browser, select the **Data Container** tab, and then locate the availability status in the **Health** column.
  - Retrieve cache health from the REST API.

```
GET /rest/v2/cache-managers/<cacheManagerName>/health
```

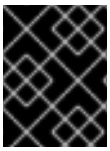
- Programmatically retrieve the availability of embedded caches with the **getAvailability()** method in the **AdvancedCache** API.

Additional resources

- REST API: Getting cluster health
- org.infinispan.AdvancedCache.getAvailability
- Enum AvailabilityMode

7.2.3. Making caches available

Make caches available for read and write operations by forcing them out of **DEGRADED** mode.



IMPORTANT

You should force clusters out of **DEGRADED** mode only if your deployment can tolerate data loss and inconsistency.

Procedure

Make caches available in one of the following ways:

- Open the Data Grid Console and select the **Make available** option.
- Change the availability of remote caches with the REST API.

```
POST /rest/v2/caches/<cacheName>?action=set-availability&availability=AVAILABLE
```

- Programmatically change the availability of embedded caches with the **AdvancedCache** API.

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

```
// Retrieve cache availability
boolean available = ac.getAvailability() == AvailabilityMode.AVAILABLE;
// Make the cache available
if (!available) {
   ac.setAvailability(AvailabilityMode.AVAILABLE);
}
```

Additional resources

- REST API: Setting cache availability
- org.infinispan.AdvancedCache

7.3. CONFIGURING PARTITION HANDLING

Configure Data Grid to use a partition handling strategy and merge policy so it can resolve split clusters when network issues occur. By default Data Grid uses a strategy that provides availability at the cost of lowering consistency guarantees for your data. When a cluster splits due to a network partition clients can continue to perform read and write operations on caches.

If you require consistency over availability, you can configure Data Grid to deny read and write operations while the cluster is split into partitions. Alternatively you can allow read operations and deny write operations. You can also specify custom merge policy implementations that configure Data Grid to resolve splits with custom logic tailored to your requirements.

Prerequisites

- Have a Data Grid cluster where you can create either a replicated or distributed cache.

NOTE

Partition handling configuration applies only to replicated and distributed caches.

Procedure

1. Open your Data Grid configuration for editing.
2. Add partition handling configuration to your cache with either the **partition-handling** element or **partitionHandling()** method.
3. Specify a strategy for Data Grid to use when the cluster splits into partitions with the **when-split** attribute or **whenSplit()** method.
   The default partition handling strategy is **ALLOW_READ_WRITES** so caches remain available.
   If your use case requires data consistency over cache availability, specify the **DENY_READ_WRITES** strategy.
4. Specify a policy that Data Grid uses to resolve conflicting entries when merging partitions with the **merge-policy** attribute or **mergePolicy()** method.
   By default Data Grid does not resolve conflicts on merge.
5. Save the changes to your Data Grid configuration.

Partition handling configuration

XML

```
<distributed-cache>
  <partition-handling when-split="DENY_READ_WRITES"
                      merge-policy="PREFERRED_ALWAYS"/>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "partition-handling" : {
      "when-split": "DENY_READ_WRITES",
      "merge-policy": "PREFERRED_ALWAYS"
    }
  }
}
```

YAML

```
distributedCache:
  partitionHandling:
    whenSplit: DENY_READ_WRITES
    mergePolicy: PREFERRED_ALWAYS
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clustering().cacheMode(CacheMode.DIST_SYNC)
       .partitionHandling()
       .whenSplit(PartitionHandling.DENY_READ_WRITES)
       .mergePolicy(MergePolicy.PREFERRED_NON_NULL);
```

7.4. PARTITION HANDLING STRATEGIES

Partition handling strategies control if Data Grid allows read and write operations when a cluster is split. The strategy you configure determines whether you get cache availability or data consistency.

Table 7.1. Partition handling strategies

Strategy	Description	Availability or consistency
ALLOW_READ_WRITES	Data Grid allows read and write operations on caches while a cluster is split into network partitions. Nodes in each partition remain available and function independently of each other. This is the default partition handling strategy.	Availability

Strategy	Description	Availability or consistency
DENY_READ_WRITES	Data Grid allows read and write operations only if all replicas of an entry are in the partition. If a partition does not include all replicas of an entry, Data Grid prevents cache operations for that entry.	Consistency
ALLOW_READS	Data Grid allows read operations for entries and prevents write operations unless the partition includes all replicas of an entry.	Consistency with read availability

7.5. MERGE POLICIES

Merge policies control how Data Grid resolves conflicts between replicas when bringing cluster partitions together. You can use one of the merge policies that Data Grid provides or you can create a custom implementation of the **EntryMergePolicy** API.

Table 7.2. Data Grid merge policies

Merge policy	Description	Considerations
NONE	Data Grid does not resolve conflicts when merging split clusters. This is the default merge policy.	Nodes drop segments for which they are not the primary owner, which can result in data loss.
PREFERRED_ALWAYS	Data Grid finds the value that exists on the majority of nodes in the cluster and uses it to resolve conflicts.	Data Grid could use stale values to resolve conflicts. Even if an entry is available the majority of nodes, the last update could happen on the minority partition.
PREFERRED_NON_NULL	Data Grid uses the first non-null value that it finds on the cluster to resolve conflicts.	Data Grid could restore deleted entries.
REMOVE_ALL	Data Grid removes any conflicting entries from the cache.	Results in loss of any entries that have different values when merging split clusters.

7.6. CONFIGURING CUSTOM MERGE POLICIES

Configure Data Grid to use custom implementations of the **EntryMergePolicy** API when handling network partitions.

Prerequisites

- Implement the **EntryMergePolicy** API.

```
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 resolves the conflict

    return the_solved_CacheEntry;
  }
```

Procedure

1. Deploy your merge policy implementation to Data Grid Server if you use remote caches.
   a. Package your classes as a JAR file that includes a **META-INF/services/org.infinispan.conflict.EntryMergePolicy** file that contains the fully qualified class name of your merge policy.

```
# List implementations of EntryMergePolicy with the full qualified class name
org.example.CustomMergePolicy
```

   b. Add the JAR file to the **server/lib** directory.

   TIP

   Use the **install** command with the Data Grid Command Line Interface (CLI) to download the JAR to the **server/lib** directory.

2. Open your Data Grid configuration for editing.
3. Configure cache encoding with the **encoding** element or **encoding()** method as appropriate. For remote caches, if you use only object metadata for comparison when merging entries then you can use **application/x-protostream** as the media type. In this case Data Grid returns entries to the **EntryMergePolicy** as **byte[]**.

   If you require the object itself when merging conflicts then you should configure caches with the **application/x-java-object** media type. In this case you must deploy the relevant ProtoStream marshallers to Data Grid Server so it can perform **byte[]** to object transformations if clients use Protobuf encoding.
4. Specify your custom merge policy with the **merge-policy** attribute or **mergePolicy()** method as part of the partition handling configuration.
5. Save your changes.

Custom merge policy configuration

XML

```
<distributed-cache name="mycache">
```

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

JSON

```
{
  "distributed-cache": {
    "partition-handling" : {
      "when-split": "DENY_READ_WRITES",
      "merge-policy": "org.example.CustomMergePolicy"
    }
  }
}
```

YAML

```
distributedCache:
  partitionHandling:
    whenSplit: DENY_READ_WRITES
    mergePolicy: org.example.CustomMergePolicy
```

ConfigurationBuilder

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clustering().cacheMode(CacheMode.DIST_SYNC)
       .partitionHandling()
       .whenSplit(PartitionHandling.DENY_READ_WRITES)
       .mergePolicy(new CustomMergePolicy());
```

Additional resources

- org.infinispan.conflict.EntryMergePolicy

7.7. MANUALLY MERGING PARTITIONS IN EMBEDDED CACHES

Detect and resolve conflicting entries to manually merge embedded caches after network partitions occur.

Procedure

- Retrieve the **ConflictManager** from the **EmbeddedCacheManager** to detect and resolve conflicting entries in a cache, as in the following example:

```
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 a 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);
```

NOTE

Although the **ConflictManager::getConflicts** stream is processed per entry, the underlying spliterator lazily loads cache entries on a per segment basis.

CHAPTER 8. SECURITY AUTHORIZATION WITH ROLE-BASED ACCESS CONTROL

Role-based access control (RBAC) capabilities use different permissions levels to restrict user interactions with Data Grid.

NOTE

For information on creating users and configuring authorization specific to remote or embedded caches, see:

- Configuring user roles and permissions with Data Grid Server
- Programmatically configuring user roles and permissions

8.1. DATA GRID USER ROLES AND PERMISSIONS

Data Grid includes several roles that provide users with permissions to access caches and Data Grid resources.

Role	Permissions	Description
admin	ALL	Superuser with all permissions including control of the Cache Manager lifecycle.
deployer	ALL_READ, ALL_WRITE, LISTEN, EXEC, MONITOR, CREATE	Can create and delete Data Grid resources in addition to **application** permissions.
application	ALL_READ, ALL_WRITE, LISTEN, EXEC, MONITOR	Has read and write access to Data Grid resources in addition to **observer** permissions. Can also listen to events and execute server tasks and scripts.
observer	ALL_READ, MONITOR	Has read access to Data Grid resources in addition to **monitor** permissions.
monitor	MONITOR	Can view statistics via JMX and the **metrics** endpoint.

Additional resources

- org.infinispan.security.AuthorizationPermission Enum
- Data Grid configuration schema reference

8.1.1. Permissions

User roles are sets of permissions with different access levels.

Table 8.1. Cache Manager permissions

Permission	Function	Description
CONFIGURATION	**defineConfiguration**	Defines new cache configurations.
LISTEN	**addListener**	Registers listeners against a Cache Manager.
LIFECYCLE	**stop**	Stops the Cache Manager.
CREATE	**createCache**, **removeCache**	Create and remove container resources such as caches, counters, schemas, and scripts.
MONITOR	**getStats**	Allows access to JMX statistics and the **metrics** endpoint.
ALL	-	Includes all Cache Manager permissions.

Table 8.2. Cache permissions

Permission	Function	Description
READ	**get**, **contains**	Retrieves entries from a cache.
WRITE	**put**, **putIfAbsent**, **replace**, **remove**, **evict**	Writes, replaces, removes, evicts data in a cache.
EXEC	**distexec**, **streams**	Allows code execution against a cache.
LISTEN	**addListener**	Registers listeners against a cache.
BULK_READ	**keySet**, **values**, **entrySet**, **query**	Executes bulk retrieve operations.
BULK_WRITE	**clear**, **putAll**	Executes bulk write operations.
LIFECYCLE	**start**, **stop**	Starts and stops a cache.

ADMIN	**getVersion**, **addInterceptor***, **removeInterceptor**, **getInterceptorChain**, **getEvictionManager**, **getComponentRegistry**, **getDistributionManager**, **getAuthorizationManager**, **evict**, **getRpcManager**, **getCacheConfiguration**, **getCacheManager**, **getInvocationContextContainer**, **setAvailability**, **getDataContainer**, **getStats**, **getXAResource**	Allows access to underlying components and internal structures.
MONITOR	**getStats**	Allows access to JMX statistics and the **metrics** endpoint.
ALL	-	Includes all cache permissions.
ALL_READ	-	Combines the READ and BULK_READ permissions.
ALL_WRITE	-	Combines the WRITE and BULK_WRITE permissions.

Additional resources

- Data Grid Security API

8.1.2. Role and permission mappers

Data Grid implements users as a collection of principals. Principals represent either an individual user identity, such as a username, or a group to which the users belong. Internally, these are implemented with the **javax.security.auth.Subject** class.

To enable authorization, the principals must be mapped to role names, which are then expanded into a set of permissions.

Data Grid includes the **PrincipalRoleMapper** API for associating security principals to roles, and the **RolePermissionMapper** API for associating roles with specific permissions.

Data Grid provides the following role and permission mapper implementations:

Cluster role mapper

Stores principal to role mappings in the cluster registry.

Cluster permission mapper

Stores role to permission mappings in the cluster registry. Allows you to dynamically modify user roles and permissions.

Identity role mapper

Uses the principal name as the role name. The type or format of the principal name depends on the source. For example, in an LDAP directory the principal name could be a Distinguished Name (DN).

Common name role mapper

Uses the Common Name (CN) as the role name. You can use this role mapper with an LDAP directory or with client certificates that contain Distinguished Names (DN); for example **cn=managers,ou=people,dc=example,dc=com** maps to the **managers** role.

8.1.2.1. Mapping users to roles and permissions in Data Grid

Consider the following user retrieved from an LDAP server, as a collection of DNs:

```
CN=myapplication,OU=applications,DC=mycompany
CN=dataprocessors,OU=groups,DC=mycompany
CN=finance,OU=groups,DC=mycompany
```

Using the **Common name role mapper**, the user would be mapped to the following roles:

```
dataprocessors
finance
```

Data Grid has the following role definitions:

```
dataprocessors: ALL_WRITE ALL_READ
finance: LISTEN
```

The user would have the following permissions:

```
ALL_WRITE ALL_READ LISTEN
```

Additional resources

- Data Grid Security API
- org.infinispan.security.PrincipalRoleMapper
- org.infinispan.security.RolePermissionMapper
- org.infinispan.security.mappers.IdentityRoleMapper
- org.infinispan.security.mappers.CommonNameRoleMapper

8.1.3. Configuring role mappers

Data Grid enables the cluster role mapper and cluster permission mapper by default. To use a different implementation for role mapping, you must configure the role mappers.

Procedure

1. Open your Data Grid configuration for editing.
2. Declare the role mapper as part of the security authorization in the Cache Manager configuration.

3. Save the changes to your configuration.

Role mapper configuration

XML

```
<cache-container>
  <security>
    <authorization>
      <common-name-role-mapper />
    </authorization>
  </security>
</cache-container>
```

JSON

```
{
  "infinispan" : {
    "cache-container" : {
      "security" : {
        "authorization" : {
          "common-name-role-mapper": {}
        }
      }
    }
  }
}
```

YAML

```
infinispan:
  cacheContainer:
    security:
      authorization:
        commonNameRoleMapper: ~
```

Additional resources

- Data Grid configuration schema reference

8.2. CONFIGURING CACHES WITH SECURITY AUTHORIZATION

Add security authorization to caches to enforce role-based access control (RBAC). This requires Data Grid users to have a role with a sufficient level of permission to perform cache operations.

Prerequisites

- Create Data Grid users and either grant them with roles or assign them to groups.

Procedure

1. Open your Data Grid configuration for editing.

2. Add a **security** section to the configuration.
3. Specify roles that users must have to perform cache operations with the **authorization** element.
   You can implicitly add all roles defined in the Cache Manager or explicitly define a subset of roles.
4. Save the changes to your configuration.

Implicit role configuration

The following configuration implicitly adds every role defined in the Cache Manager:

XML

```
<distributed-cache>
  <security>
    <authorization/>
  </security>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "security": {
      "authorization": {
        "enabled": true
      }
    }
  }
}
```

YAML

```
distributedCache:
  security:
    authorization:
      enabled: true
```

Explicit role configuration

The following configuration explicitly adds a subset of roles defined in the Cache Manager. In this case Data Grid denies cache operations for any users that do not have one of the configured roles.

XML

```
<distributed-cache>
  <security>
    <authorization roles="admin supervisor"/>
  </security>
</distributed-cache>
```

JSON

```
{
  "distributed-cache": {
    "security": {
      "authorization": {
        "enabled": true,
        "roles": ["admin","supervisor"]
      }
    }
  }
}
```

YAML

```
distributedCache:
  security:
    authorization:
      enabled: true
      roles: ["admin","supervisor"]
```

CHAPTER 9. CONFIGURING TRANSACTIONS

Data that resides on a distributed system is vulnerable to errors that can arise from temporary network outages, system failures, or just simple human error. These external factors are uncontrollable but can have serious consequences for quality of your data. The effects of data corruption range from lower customer satisfaction to costly system reconciliation that results in service unavailability.

Data Grid can carry out ACID (atomicity, consistency, isolation, durability) transactions to ensure the cache state is consistent.

9.1. TRANSACTIONS

Data Grid can be configured to use and to participate in JTA compliant transactions.

Alternatively, if transaction support is disabled, it is equivalent to using autocommit in JDBC calls, where modifications are potentially replicated after every change (if replication is enabled).

On every cache operation Data Grid does the following:

1. Retrieves the current Transaction associated with the thread
2. If not already done, registers XAResource with the transaction manager to be notified when a transaction commits or is rolled back.

In order to do this, the cache has to be provided with a reference to the environment's TransactionManager. This is usually done by configuring the cache with the class name of an implementation of the TransactionManagerLookup interface. When the cache starts, it will create an instance of this class and invoke its **getTransactionManager()** method, which returns a reference to the **TransactionManager**.

Data Grid ships with several transaction manager lookup classes:

Transaction manager lookup implementations

- EmbeddedTransactionManagerLookup: This provides with a basic transaction manager which should only be used for embedded mode when no other implementation is available. This implementation has some severe limitations to do with concurrent transactions and recovery.
- JBossStandaloneJTAManagerLookup: If you're running Data Grid in a standalone environment, or in JBoss AS 7 and earlier, and WildFly 8, 9, and 10, this should be your default choice for transaction manager. It's a fully fledged transaction manager based on JBoss Transactions which overcomes all the deficiencies of the **EmbeddedTransactionManager**.
- WildflyTransactionManagerLookup: If you're running Data Grid in WildFly 11 or later, this should be your default choice for transaction manager.
- GenericTransactionManagerLookup: This is a lookup class that locate transaction managers in the most popular Java EE application servers. If no transaction manager can be found, it defaults on the **EmbeddedTransactionManager**.

Once initialized, the **TransactionManager** can also be obtained from the **Cache** itself:

```
//the cache must have a transactionManagerLookupClass defined
Cache cache = cacheManager.getCache();
```

```
//equivalent with calling TransactionManagerLookup.getTransactionManager();
TransactionManager tm = cache.getAdvancedCache().getTransactionManager();
```

9.1.1. Configuring transactions

Transactions are configured at cache level. Below is the configuration that affects a transaction behaviour and a small description of each configuration attribute.

```
<locking
  isolation="READ_COMMITTED"/>
<transaction
  locking="OPTIMISTIC"
  auto-commit="true"
  complete-timeout="60000"
  mode="NONE"
  notifications="true"
  reaper-interval="30000"
  recovery-cache="__recoveryInfoCacheName__"
  stop-timeout="30000"
  transaction-manager-
lookup="org.infinispan.transaction.lookup.GenericTransactionManagerLookup"/>
```

or programmatically:

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.locking()
   .isolationLevel(IsolationLevel.READ_COMMITTED);
builder.transaction()
   .lockingMode(LockingMode.OPTIMISTIC)
   .autoCommit(true)
   .completedTxTimeout(60000)
   .transactionMode(TransactionMode.NON_TRANSACTIONAL)
   .useSynchronization(false)
   .notifications(true)
   .reaperWakeUpInterval(30000)
   .cacheStopTimeout(30000)
   .transactionManagerLookup(new GenericTransactionManagerLookup())
   .recovery()
   .enabled(false)
   .recoveryInfoCacheName("__recoveryInfoCacheName__");
```

- **isolation** - configures the isolation level. Check section Isolation Levels for more details. Default is **REPEATABLE_READ**.
- **locking** - configures whether the cache uses optimistic or pessimistic locking. Check section Transaction Locking for more details. Default is **OPTIMISTIC**.
- **auto-commit** - if enable, the user does not need to start a transaction manually for a single operation. The transaction is automatically started and committed. Default is **true**.
- **complete-timeout** - the duration in milliseconds to keep information about completed transactions. Default is **60000**.
- **mode** - configures whether the cache is transactional or not. Default is **NONE**. The available options are:

- **NONE** - non transactional cache
- **FULL_XA** - XA transactional cache with recovery enabled. Check section Transaction recovery for more details about recovery.
- **NON_DURABLE_XA** - XA transactional cache with recovery disabled.
- **NON_XA** - transactional cache with integration via Synchronization instead of XA. Check section Enlisting Synchronizations for details.
- **BATCH**- transactional cache using batch to group operations. Check section Batching for details.

- **notifications** - enables/disables triggering transactional events in cache listeners. Default is **true**.
- **reaper-interval** - the time interval in millisecond at which the thread that cleans up transaction completion information kicks in. Defaults is **30000**.
- **recovery-cache** - configures the cache name to store the recovery information. Check section Transaction recovery for more details about recovery. Default is ***recoveryInfoCacheName***.
- **stop-timeout** - the time in millisecond to wait for ongoing transaction when the cache is stopping. Default is **30000**.
- **transaction-manager-lookup** - configures the fully qualified class name of a class that looks up a reference to a **javax.transaction.TransactionManager**. Default is **org.infinispan.transaction.lookup.GenericTransactionManagerLookup**.

For more details on how Two-Phase-Commit (2PC) is implemented in Data Grid and how locks are being acquired see the section below. More details about the configuration settings are available in Configuration reference.

9.1.2. Isolation levels

Data Grid offers two isolation levels - READ_COMMITTED and REPEATABLE_READ.

These isolation levels determine when readers see a concurrent write, and are internally implemented using different subclasses of **MVCCEntry**, which have different behaviour in how state is committed back to the data container.

Here's a more detailed example that should help understand the difference between **READ_COMMITTED** and **REPEATABLE_READ** in the context of Data Grid. With **READ_COMMITTED**, if between two consecutive read calls on the same key, the key has been updated by another transaction, the second read may return the new updated value:

```
Thread1: tx1.begin()
Thread1: cache.get(k) // returns v
Thread2:                        tx2.begin()
Thread2:                        cache.get(k) // returns v
Thread2:                        cache.put(k, v2)
Thread2:                        tx2.commit()
Thread1: cache.get(k) // returns v2!
Thread1: tx1.commit()
```

With **REPEATABLE_READ**, the final get will still return **v**. So, if you're going to retrieve the same key multiple times within a transaction, you should use **REPEATABLE_READ**.

However, as read-locks are not acquired even for **REPEATABLE_READ**, this phenomena can occur:

```
cache.get("A") // returns 1
cache.get("B") // returns 1

Thread1: tx1.begin()
Thread1: cache.put("A", 2)
Thread1: cache.put("B", 2)
Thread2:                        tx2.begin()
Thread2:                        cache.get("A") // returns 1
Thread1: tx1.commit()
Thread2:                        cache.get("B") // returns 2
Thread2:                        tx2.commit()
```

9.1.3. Transaction locking

9.1.3.1. Pessimistic transactional cache

From a lock acquisition perspective, pessimistic transactions obtain locks on keys at the time the key is written.

1. A lock request is sent to the primary owner (can be an explicit lock request or an operation)
2. The primary owner tries to acquire the lock:
    a. If it succeed, it sends back a positive reply;
    b. Otherwise, a negative reply is sent and the transaction is rollback.

As an example:

```
transactionManager.begin();
cache.put(k1,v1); //k1 is locked.
cache.remove(k2); //k2 is locked when this returns
transactionManager.commit();
```

When **cache.put(k1,v1)** returns, **k1** is locked and no other transaction running anywhere in the cluster can write to it. Reading **k1** is still possible. The lock on **k1** is released when the transaction completes (commits or rollbacks).

NOTE

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

9.1.3.2. Optimistic transactional cache

With optimistic transactions locks are being acquired at transaction prepare time and are only being held up to the point the transaction commits (or rollbacks). This is different from the 5.0 default locking model where local locks are being acquire on writes and cluster locks are being acquired during prepare time.

1. The prepare is sent to all the owners.
2. The primary owners try to acquire the locks needed:
    a. If locking succeeds, it performs the write skew check.
    b. If the write skew check succeeds (or is disabled), send a positive reply.
    c. Otherwise, a negative reply is sent and the transaction is rolled back.

As an example:

```
transactionManager.begin();
cache.put(k1,v1);
cache.remove(k2);
transactionManager.commit(); //at prepare time, K1 and K2 is locked until committed/rolled back.
```

NOTE

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

9.1.3.3. What do I need - pessimistic or optimistic transactions?

From a use case perspective, optimistic transactions should be used when there is *not* a lot of contention between multiple transactions running at the same time. That is because the optimistic transactions rollback if data has changed between the time it was read and the time it was committed (with write skew check enabled).

On the other hand, pessimistic transactions might be a better fit when there is high contention on the keys and transaction rollbacks are less desirable. Pessimistic transactions are more costly by their nature: each write operation potentially involves a RPC for lock acquisition.

9.1.4. Write Skews

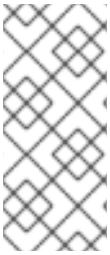
Write skews occur when two transactions independently and simultaneously read and write to the same key. The result of a write skew is that both transactions successfully commit updates to the same key but with different values.

Data Grid automatically performs write skew checks to ensure data consistency for **REPEATABLE_READ** isolation levels in optimistic transactions. This allows Data Grid to detect and roll back one of the transactions.

When operating in **LOCAL** mode, write skew checks rely on Java object references to compare differences, which provides a reliable technique for checking for write skews.

9.1.4.1. Forcing write locks on keys in pessimitic transactions

To avoid write skews with pessimistic transactions, lock keys at read-time with **Flag.FORCE_WRITE_LOCK**.



NOTE

- In non-transactional caches, **Flag.FORCE_WRITE_LOCK** does not work. The **get()** call reads the key value but does not acquire locks remotely.
- You should use **Flag.FORCE_WRITE_LOCK** with transactions in which the entity is updated later in the same transaction.

Compare the following code snippets for an example of **Flag.FORCE_WRITE_LOCK**:

```
// begin the transaction
if (!cache.getAdvancedCache().lock(key)) {
   // abort the transaction because the key was not locked
} else {
   cache.get(key);
   cache.put(key, value);
   // commit the transaction
}
```

```
// begin the transaction
try {
   // throws an exception if the key is not locked.
   cache.getAdvancedCache().withFlags(Flag.FORCE_WRITE_LOCK).get(key);
   cache.put(key, value);
} catch (CacheException e) {
   // mark the transaction rollback-only
}
// commit or rollback the transaction
```

9.1.5. Dealing with exceptions

If a CacheException (or a subclass of it) is thrown by a cache method within the scope of a JTA transaction, then the transaction is automatically marked for rollback.

9.1.6. Enlisting Synchronizations

By default Data Grid registers itself as a first class participant in distributed transactions through XAResource. There are situations where Data Grid is not required to be a participant in the transaction, but only to be notified by its lifecycle (prepare, complete): e.g. in the case Data Grid is used as a 2nd level cache in Hibernate.

Data Grid allows transaction enlistment through Synchronization. To enable it just use **NON_XA** transaction mode.

Synchronizations have the advantage that they allow **TransactionManager** to optimize 2PC with a 1PC where only one other resource is enlisted with that transaction (last resource commit optimization). E.g. Hibernate second level cache: if Data Grid registers itself with the **TransactionManager** as a **XAResource** than at commit time, the **TransactionManager** sees two **XAResource** (cache and database) and does not make this optimization. Having to coordinate between two resources it needs to write the tx log to disk. On the other hand, registering Data Grid as a **Synchronization** makes the **TransactionManager** skip writing the log to the disk (performance improvement).

9.1.7. Batching

Batching allows atomicity and some characteristics of a transaction, but not full-blown JTA or XA capabilities. Batching is often a lot lighter and cheaper than a full-blown transaction.

TIP

Generally speaking, one should use batching API whenever the only participant in the transaction is an Data Grid cluster. On the other hand, JTA transactions (involving **TransactionManager**) should be used whenever the transactions involves multiple systems. E.g. considering the "Hello world!" of transactions: transferring money from one bank account to the other. If both accounts are stored within Data Grid, then batching can be used. If one account is in a database and the other is Data Grid, then distributed transactions are required.

NOTE

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

9.1.7.1. API

Once you have configured your cache to use batching, you use it by calling **startBatch()** and **endBatch()** on **Cache**. E.g.,

```
Cache cache = cacheManager.getCache();
// not using a batch
cache.put("key", "value"); // will replicate immediately

// using a batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k2", "value");
cache.endBatch(true); // This will now replicate the modifications since the batch was started.

// a new batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(false); // This will "discard" changes made in the batch
```

9.1.7.2. Batching and JTA

Behind the scenes, the batching functionality starts a JTA transaction, and all the invocations in that scope are associated with it. For this it uses a very simple (e.g. no recovery) internal **TransactionManager** implementation. With batching, you get:

1. Locks you acquire during an invocation are held until the batch completes
2. Changes are all replicated around the cluster in a batch as part of the batch completion process. Reduces replication chatter for each update in the batch.
3. If synchronous replication or invalidation are used, a failure in replication/invalidation will cause the batch to roll back.
4. All the transaction related configurations apply for batching as well.

9.1.8. Transaction recovery

Recovery is a feature of XA transactions, which deal with the eventuality of a resource or possibly even the transaction manager failing, and recovering accordingly from such a situation.

9.1.8.1. When to use recovery

Consider a distributed transaction in which money is transferred from an account stored in an external database to an account stored in Data Grid. When **TransactionManager.commit()** is invoked, both resources prepare successfully (1st phase). During the commit (2nd) phase, the database successfully applies the changes whilst Data Grid fails before receiving the commit request from the transaction manager. At this point the system is in an inconsistent state: money is taken from the account in the external database but not visible yet in Data Grid (since locks are only released during 2nd phase of a two-phase commit protocol). Recovery deals with this situation to make sure data in both the database and Data Grid ends up in a consistent state.

9.1.8.2. How does it work

Recovery is coordinated by the transaction manager. The transaction manager works with Data Grid to determine the list of in-doubt transactions that require manual intervention and informs the system administrator (via email, log alerts, etc). This process is transaction manager specific, but generally requires some configuration on the transaction manager.

Knowing the in-doubt transaction ids, the system administrator can now connect to the Data Grid cluster and replay the commit of transactions or force the rollback. Data Grid provides JMX tooling for this - this is explained extensively in the Transaction recovery and reconciliation section.

9.1.8.3. Configuring recovery

Recovery is *not* enabled by default in Data Grid. If disabled, the **TransactionManager** won't be able to work with Data Grid to determine the in-doubt transactions. The Transaction configuration section shows how to enable it.

NOTE: **recovery-cache** attribute is not mandatory and it is configured per-cache.

NOTE

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

9.1.8.3.1. Enable JMX support

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

9.1.8.4. Recovery cache

In order to track in-doubt transactions and be able to reply them, Data Grid caches all transaction state for future use. This state is held only for in-doubt transaction, being removed for successfully completed transactions after when the commit/rollback phase completed.

This in-doubt transaction data is held within a local cache: this allows one to configure swapping this info to disk through cache loader in the case it gets too big. This cache can be specified through the **recovery-cache** configuration attribute. If not specified Data Grid will configure a local cache for you.

It is possible (though not mandated) to share same recovery cache between all the Data Grid caches that have recovery enabled. If the default recovery cache is overridden, then the specified recovery cache must use a TransactionManagerLookup that returns a different transaction manager than the one used by the cache itself.

9.1.8.5. Integration with the transaction manager

Even though this is transaction manager specific, generally a transaction manager would need a reference to a **XAResource** implementation in order to invoke **XAResource.recover()** on it. In order to obtain a reference to an Data Grid **XAResource** following API can be used:

```
XAResource xar = cache.getAdvancedCache().getXAResource();
```

It is a common practice to run the recovery in a different process from the one running the transaction.

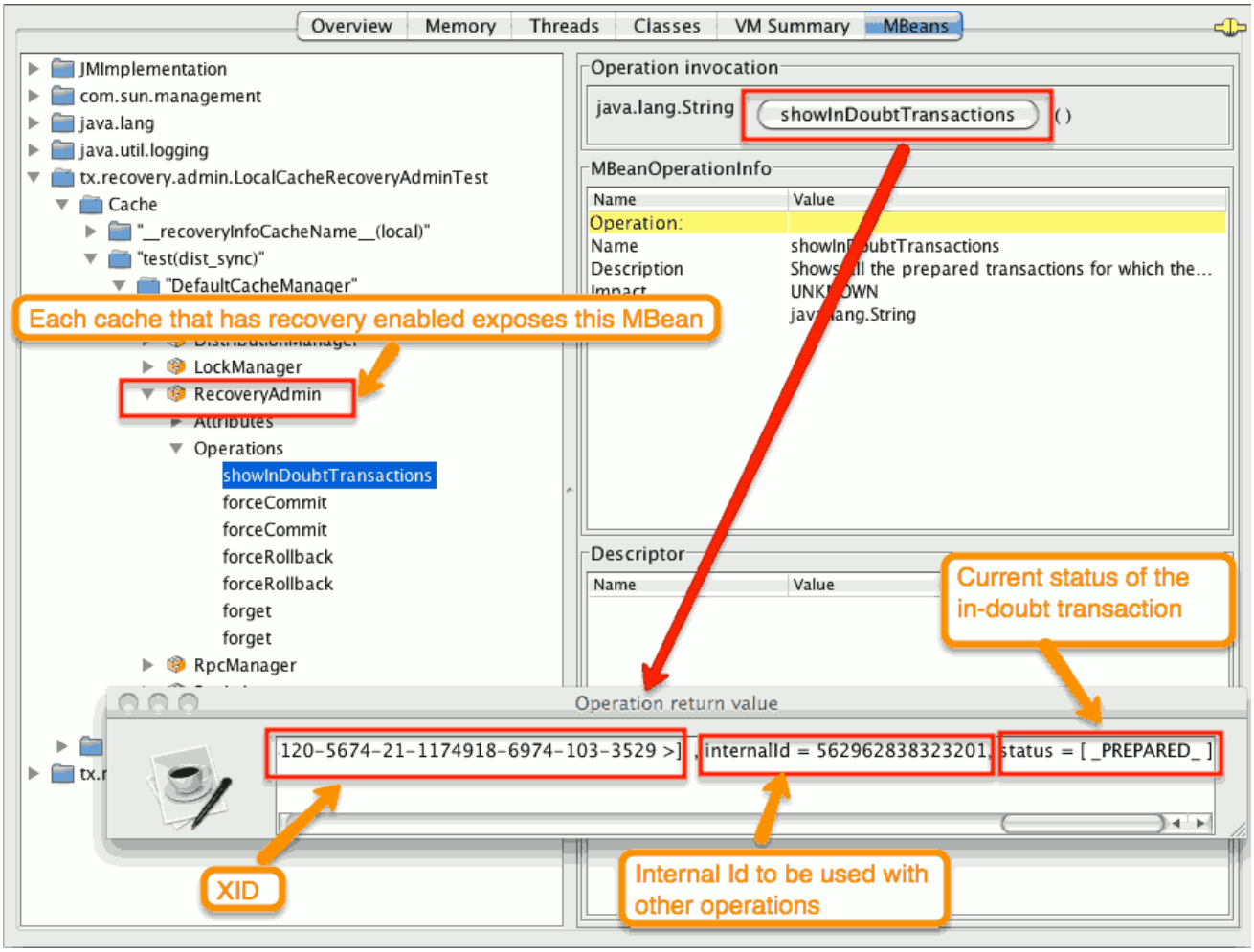
9.1.8.6. Reconciliation

The transaction manager informs the system administrator on in-doubt transaction in a proprietary way. At this stage it is assumed that the system administrator knows transaction's XID (a byte array).

A normal recovery flow is:

- **STEP 1**: The system administrator connects to an Data Grid server through JMX, and lists the in doubt transactions. The image below demonstrates JConsole connecting to an Data Grid node that has an in doubt transaction.

Figure 9.1. Show in-doubt transactions

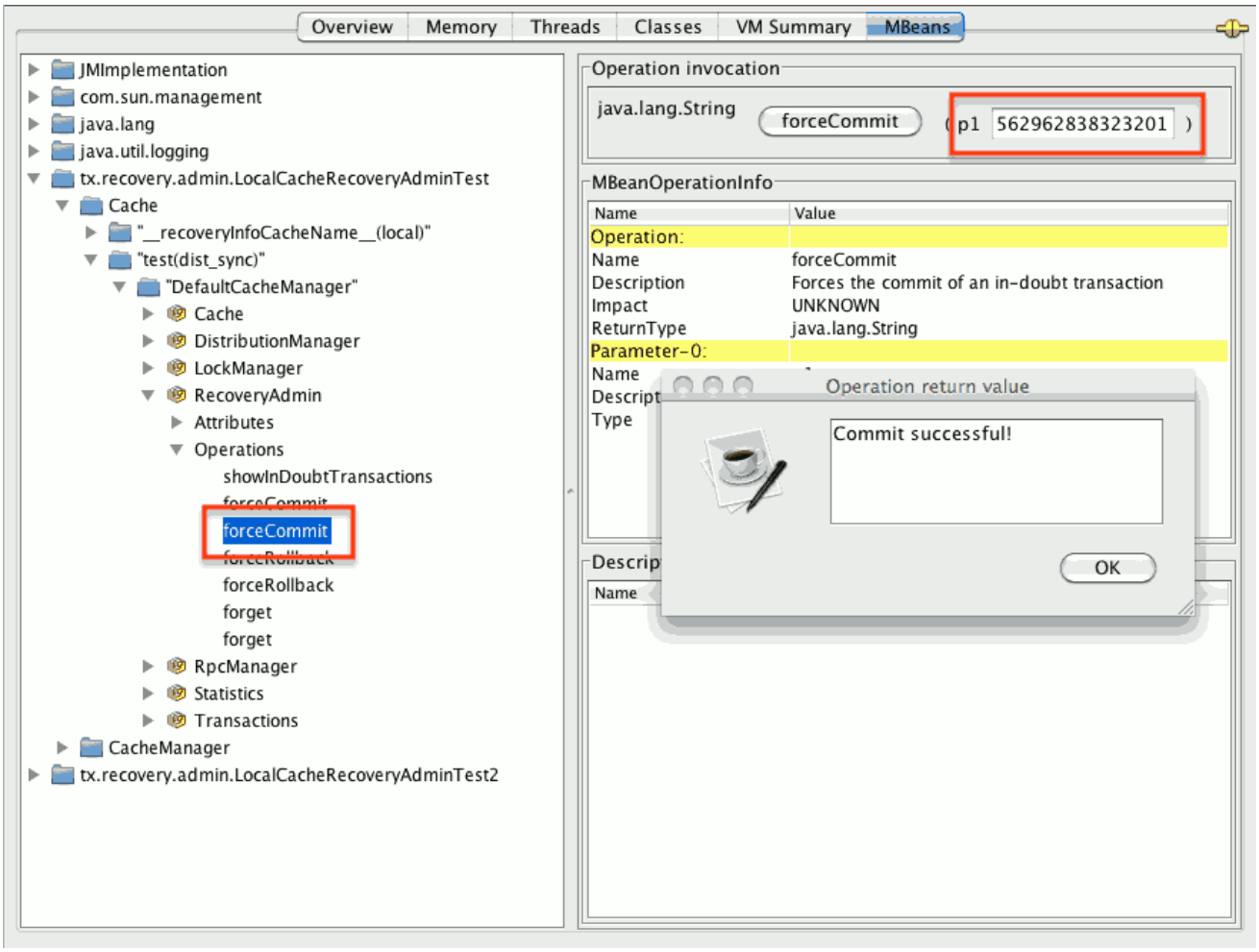


The status of each in-doubt transaction is displayed(in this example " *PREPARED* "). There might be multiple elements in the status field, e.g. "PREPARED" and "COMMITTED" in the case the transaction committed on certain nodes but not on all of them.

- **STEP 2**: The system administrator visually maps the XID received from the transaction manager to an Data Grid internal id, represented as a number. This step is needed because the XID, a byte array, cannot conveniently be passed to the JMX tool (e.g. JConsole) and then re-assembled on Data Grid's side.

- **STEP 3**: The system administrator forces the transaction's commit/rollback through the corresponding jmx operation, based on the internal id. The image below is obtained by forcing the commit of the transaction based on its internal id.

Figure 9.2. Force commit



TIP

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

9.1.8.6.1. Force commit/rollback based on XID

XID-based JMX operations for forcing in-doubt transactions' commit/rollback are available as well: these methods receive byte[] arrays describing the XID instead of the number associated with the transactions (as previously described at step 2). These can be useful e.g. if one wants to set up an automatic completion job for certain in-doubt transactions. This process is plugged into transaction manager's recovery and has access to the transaction manager's XID objects.

CHAPTER 10. CONFIGURING LOCKING AND CONCURRENCY

Data Grid uses multi-versioned concurrency control (MVCC) to improve access to shared data.

- Allowing concurrent readers and writers
- Readers and writers do not block one another
- Write skews can be detected and handled
- Internal locks can be striped

10.1. LOCKING AND CONCURRENCY

Multi-versioned concurrency control (MVCC) is a concurrency scheme popular with relational databases and other data stores. MVCC offers many advantages over coarse-grained Java synchronization and even JDK Locks for access to shared data.

Data Grid's MVCC implementation makes use of minimal locks and synchronizations, leaning heavily towards lock-free techniques such as compare-and-swap and lock-free data structures wherever possible, which helps optimize for multi-CPU and multi-core environments.

In particular, Data Grid's MVCC implementation is heavily optimized for readers. Reader threads do not acquire explicit locks for entries, and instead directly read the entry in question.

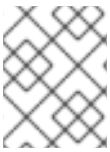
Writers, on the other hand, need to acquire a write lock. This ensures only one concurrent writer per entry, causing concurrent writers to queue up to change an entry.

To allow concurrent reads, writers make a copy of the entry they intend to modify, by wrapping the entry in an **MVCCEntry**. This copy isolates concurrent readers from seeing partially modified state. Once a write has completed, **MVCCEntry.commit()** will flush changes to the data container and subsequent readers will see the changes written.

10.1.1. Clustered caches and locks

In Data Grid clusters, primary owner nodes are responsible for locking keys.

For non-transactional caches, Data Grid forwards the write operation to the primary owner of the key so it can attempt to lock it. Data Grid either then forwards the write operation to the other owners or throws an exception if it cannot lock the key.



NOTE

If the operation is conditional and fails on the primary owner, Data Grid does not forward it to the other owners.

For transactional caches, primary owners can lock keys with optimistic and pessimistic locking modes. Data Grid also supports different isolation levels to control concurrent reads between transactions.

10.1.2. The LockManager

The **LockManager** is a component that is responsible for locking an entry for writing. The **LockManager** makes use of a **LockContainer** to locate/hold/create locks. **LockContainers** come in two broad flavours, with support for lock striping and with support for one lock per entry.

10.1.3. Lock striping

Lock striping entails the use of a fixed-size, shared collection of locks for the entire cache, with locks being allocated to entries based on the entry's key's hash code. Similar to the way the JDK's **ConcurrentHashMap** allocates locks, this allows for a highly scalable, fixed-overhead locking mechanism in exchange for potentially unrelated entries being blocked by the same lock.

The alternative is to disable lock striping - which would mean a *new* lock is created per entry. This approach *may* give you greater concurrent throughput, but it will be at the cost of additional memory usage, garbage collection churn, etc.

DEFAULT LOCK STRIPING SETTINGS

lock striping is disabled by default, due to potential deadlocks that can happen if locks for different keys end up in the same lock stripe.

The size of the shared lock collection used by lock striping can be tuned using the **concurrencyLevel** attribute of the **<locking />** configuration element.

Configuration example:

```
<locking striping="false|true"/>
```

Or

```
new ConfigurationBuilder().locking().useLockStriping(false|true);
```

10.1.4. Concurrency levels

In addition to determining the size of the striped lock container, this concurrency level is also used to tune any JDK **ConcurrentHashMap** based collections where related, such as internal to **DataContainer**s. Please refer to the JDK **ConcurrentHashMap** Javadocs for a detailed discussion of concurrency levels, as this parameter is used in exactly the same way in Data Grid.

Configuration example:

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

Or

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

10.1.5. Lock timeout

The lock timeout specifies the amount of time, in milliseconds, to wait for a contented lock.

Configuration example:

```
<locking acquire-timeout="10000"/>
```

Or

```
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10000);
//alternatively
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10, TimeUnit.SECONDS);
```

10.1.6. Consistency

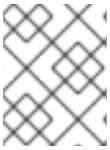
The fact that a single owner is locked (as opposed to all owners being locked) does not break the following consistency guarantee: if key **K** is hashed to nodes **{A, B}** and transaction **TX1** acquires a lock for **K**, let's say on **A**. If another transaction, **TX2**, is started on **B** (or any other node) and **TX2** tries to lock **K** then it will fail with a timeout as the lock is already held by **TX1**. The reason for this is the that the lock for a key **K** is always, deterministically, acquired on the same node of the cluster, regardless of where the transaction originates.

10.1.7. Data Versioning

Data Grid supports two forms of data versioning: simple and external. The simple versioning is used in transactional caches for write skew check.

The external versioning is used to encapsulate an external source of data versioning within Data Grid, such as when using Data Grid with Hibernate which in turn gets its data version information directly from a database.

In this scheme, a mechanism to pass in the version becomes necessary, and overloaded versions of **put()** and **putForExternalRead()** will be provided in **AdvancedCache** to take in an external data version. This is then stored on the **InvocationContext** and applied to the entry at commit time.



NOTE

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

CHAPTER 11. USING CLUSTERED COUNTERS

Data Grid provides counters that record the count of objects and are distributed across all nodes in a cluster.

11.1. CLUSTERED COUNTERS

Clustered counters are counters which are distributed and shared among all nodes in the Data Grid cluster. Counters can have different consistency levels: strong and weak.

Although a strong/weak consistent counter has separate interfaces, both support updating its value, return the current value and they provide events when its value is updated. Details are provided below in this document to help you choose which one fits best your uses-case.

11.1.1. Installation and Configuration

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

pom.xml

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

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

configuration.xml

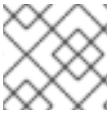
```
<infinispan>
   <cache-container ...>
      <!-- To persist counters, you need to configure the global state. -->
      <global-state>
      <!-- Global state configuration goes here. -->
      </global-state>
      <!-- Cache configuration goes here. -->
       <counters xmlns="urn:infinispan:config:counters:14.0" num-owners="3"
reliability="CONSISTENT">
          <strong-counter name="c1" initial-value="1" storage="PERSISTENT"/>
          <strong-counter name="c2" initial-value="2" storage="VOLATILE" lower-bound="0"/>
          <strong-counter name="c3" initial-value="3" storage="PERSISTENT" upper-bound="5"/>
          <strong-counter name="c4" initial-value="4" storage="VOLATILE" lower-bound="0" upper-
bound="10"/>
          <strong-counter name="c5" initial-value="0" upper-bound="100" lifespan="60000"/>
          <weak-counter name="c6" initial-value="5" storage="PERSISTENT" concurrency-level="1"/>
       </counters>
   </cache-container>
</infinispan>
```

or programmatically, in the **GlobalConfigurationBuilder**:

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...;
CounterManagerConfigurationBuilder builder =
globalConfigurationBuilder.addModule(CounterManagerConfigurationBuilder.class);
builder.numOwner(3).reliability(Reliability.CONSISTENT);
builder.addStrongCounter().name("c1").initialValue(1).storage(Storage.PERSISTENT);
builder.addStrongCounter().name("c2").initialValue(2).lowerBound(0).storage(Storage.VOLATILE);
builder.addStrongCounter().name("c3").initialValue(3).upperBound(5).storage(Storage.PERSISTENT)
;
builder.addStrongCounter().name("c4").initialValue(4).lowerBound(0).upperBound(10).storage(Storag
e.VOLATILE);
builder.addStrongCounter().name("c5").initialValue(0).upperBound(100).lifespan(60000);
builder.addWeakCounter().name("c6").initialValue(5).concurrencyLevel(1).storage(Storage.PERSIST
ENT);
```

On other hand, the counters can be configured on-demand, at any time after the **EmbeddedCacheManager** is initialized.

```
CounterManager manager = ...;
manager.defineCounter("c1",
CounterConfiguration.builder(CounterType.UNBOUNDED_STRONG).initialValue(1).storage(Storage.
PERSISTENT).build());
manager.defineCounter("c2",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(2).lowerBound(0).stora
ge(Storage.VOLATILE).build());
manager.defineCounter("c3",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(3).upperBound(5).stor
age(Storage.PERSISTENT).build());
manager.defineCounter("c4",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(4).lowerBound(0).uppe
rBound(10).storage(Storage.VOLATILE).build());
manager.defineCounter("c4",
CounterConfiguration.builder(CounterType.BOUNDED_STRONG).initialValue(0).upperBound(100).lif
espan(60000).build());
manager.defineCounter("c6",
CounterConfiguration.builder(CounterType.WEAK).initialValue(5).concurrencyLevel(1).storage(Storag
e.PERSISTENT).build());
```



NOTE

CounterConfiguration is immutable and can be reused.

The method **defineCounter()** will return **true** if the counter is successful configured or **false** otherwise. However, if the configuration is invalid, the method will throw a **CounterConfigurationException**. To find out if a counter is already defined, use the method **isDefined()**.

```
CounterManager manager = ...
if (!manager.isDefined("someCounter")) {
   manager.define("someCounter", ...);
}
```

Additional resources

- Data Grid configuration schema reference

11.1.1.1. List counter names

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

11.1.2. CounterManager interface

The **CounterManager** interface is the entry point to define, retrieve and remove counters.

Embedded deployments

CounterManager automatically listen to the creation of **EmbeddedCacheManager** and proceeds with the registration of an instance of it per **EmbeddedCacheManager**. It starts the caches needed to store the counter state and configures the default counters.

Retrieving the **CounterManager** is as simple as invoke the **EmbeddedCounterManagerFactory.asCounterManager(EmbeddedCacheManager)** as shown in the example below:

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

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

Server deployments

For Hot Rod clients, the **CounterManager** is registered in the RemoteCacheManager and can be retrieved as follows:

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

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

11.1.2.1. Remove a counter via CounterManager

There is a difference between remove a counter via the **Strong/WeakCounter** interfaces and the **CounterManager**. The **CounterManager.remove(String)** removes the counter value from the cluster and removes all the listeners registered in the counter in the local counter instance. In addition, the counter instance is no longer reusable and it may return an invalid results.

On the other side, the **Strong/WeakCounter** removal only removes the counter value. The instance can still be reused and the listeners still works.

NOTE

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

11.1.3. The Counter

A counter can be strong (**StrongCounter**) or weakly consistent (**WeakCounter**) and both is identified by

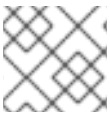
a name. They have a specific interface but they share some logic, namely, both of them are asynchronous (a **CompletableFuture** is returned by each operation), provide an update event and can be reset to its initial value.

If you don't want to use the async API, it is possible to return a synchronous counter via **sync()** method. The API is the same but without the **CompletableFuture** return value.

The following methods are common to both interfaces:

```
String getName();
CompletableFuture<Long> getValue();
CompletableFuture<Void> reset();
<T extends CounterListener> Handle<T> addListener(T listener);
CounterConfiguration getConfiguration();
CompletableFuture<Void> remove();
SyncStrongCounter sync(); //SyncWeakCounter for WeakCounter
```

- **getName()** returns the counter name (identifier).
- **getValue()** returns the current counter's value.
- **reset()** allows to reset the counter's value to its initial value.
- **addListener()** register a listener to receive update events. More details about it in the Notification and Events section.
- **getConfiguration()** returns the configuration used by the counter.
- **remove()** removes the counter value from the cluster. The instance can still be used and the listeners are kept.
- **sync()** creates a synchronous counter.



NOTE

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

11.1.3.1. The StrongCounter interface: when the consistency or bounds matters.

The strong counter provides uses a single key stored in Data Grid cache to provide the consistency needed. All the updates are performed under the key lock to updates its values. On other hand, the reads don't acquire any locks and reads the current value. Also, with this scheme, it allows to bound the counter value and provide atomic operations like compare-and-set/swap.

A **StrongCounter** can be retrieved from the **CounterManager** by using the **getStrongCounter()** method. As an example:

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

WARNING

Since every operation will hit a single key, the **StrongCounter** has a higher contention rate.

The **StrongCounter** interface adds the following method:

```
default CompletableFuture<Long> incrementAndGet() {
  return addAndGet(1L);
}

default CompletableFuture<Long> decrementAndGet() {
  return addAndGet(-1L);
}

CompletableFuture<Long> addAndGet(long delta);

CompletableFuture<Boolean> compareAndSet(long expect, long update);

CompletableFuture<Long> compareAndSwap(long expect, long update);
```

- **incrementAndGet()** increments the counter by one and returns the new value.
- **decrementAndGet()** decrements the counter by one and returns the new value.
- **addAndGet()** adds a delta to the counter's value and returns the new value.
- **compareAndSet()** and **compareAndSwap()** atomically set the counter's value if the current value is the expected.

NOTE

A operation is considered completed when the **CompletableFuture** is completed.

NOTE

The difference between compare-and-set and compare-and-swap is that the former returns true if the operation succeeds while the later returns the previous value. The compare-and-swap is successful if the return value is the same as the expected.

11.1.3.1.1. Bounded StrongCounter

When bounded, all the update method above will throw a **CounterOutOfBoundsException** when they reached the lower or upper bound. The exception has the following methods to check which side bound has been reached:

```
public boolean isUpperBoundReached();
public boolean isLowerBoundReached();
```

11.1.3.1.2. Uses cases

The strong counter fits better in the following uses cases:

- When counter's value is needed after each update (example, cluster-wise ids generator or sequences)
- When a bounded counter is needed (example, rate limiter)

11.1.3.1.3. Usage Examples

```
StrongCounter counter = counterManager.getStrongCounter("unbounded_counter");

// incrementing the counter
System.out.println("new value is " + counter.incrementAndGet().get());

// decrement the counter's value by 100 using the functional API
counter.addAndGet(-100).thenApply(v -> {
   System.out.println("new value is " + v);
   return null;
}).get();

// alternative, you can do some work while the counter is updated
CompletableFuture<Long> f = counter.addAndGet(10);
// ... do some work ...
System.out.println("new value is " + f.get());

// and then, check the current value
System.out.println("current value is " + counter.getValue().get());

// finally, reset to initial value
counter.reset().get();
System.out.println("current value is " + counter.getValue().get());

// or set to a new value if zero
System.out.println("compare and set succeeded? " + counter.compareAndSet(0, 1));
```

And below, there is another example using a bounded counter:

```
StrongCounter counter = counterManager.getStrongCounter("bounded_counter");

// incrementing the counter
try {
   System.out.println("new value is " + counter.addAndGet(100).get());
} catch (ExecutionException e) {
   Throwable cause = e.getCause();
   if (cause instanceof CounterOutOfBoundsException) {
      if (((CounterOutOfBoundsException) cause).isUpperBoundReached()) {
         System.out.println("ops, upper bound reached.");
      } else if (((CounterOutOfBoundsException) cause).isLowerBoundReached()) {
         System.out.println("ops, lower bound reached.");
      }
   }
}
```

```
// now using the functional API
counter.addAndGet(-100).handle((v, throwable) -> {
   if (throwable != null) {
      Throwable cause = throwable.getCause();
      if (cause instanceof CounterOutOfBoundsException) {
         if (((CounterOutOfBoundsException) cause).isUpperBoundReached()) {
            System.out.println("ops, upper bound reached.");
         } else if (((CounterOutOfBoundsException) cause).isLowerBoundReached()) {
            System.out.println("ops, lower bound reached.");
         }
      }
      return null;
   }
   System.out.println("new value is " + v);
   return null;
}).get();
```

Compare-and-set vs Compare-and-swap examples:

```
StrongCounter counter = counterManager.getStrongCounter("my-counter");
long oldValue, newValue;
do {
   oldValue = counter.getValue().get();
   newValue = someLogic(oldValue);
} while (!counter.compareAndSet(oldValue, newValue).get());
```

With compare-and-swap, it saves one invocation counter invocation (**counter.getValue()**)

```
StrongCounter counter = counterManager.getStrongCounter("my-counter");
long oldValue = counter.getValue().get();
long currentValue, newValue;
do {
   currentValue = oldValue;
   newValue = someLogic(oldValue);
} while ((oldValue = counter.compareAndSwap(oldValue, newValue).get()) != currentValue);
```

To use a strong counter as a rate limiter, configure **upper-bound** and **lifespan** parameters as follows:

```
// 5 request per minute
CounterConfiguration configuration =
CounterConfiguration.builder(CounterType.BOUNDED_STRONG)
      .upperBound(5)
      .lifespan(60000)
      .build();
counterManager.defineCounter("rate_limiter", configuration);
StrongCounter counter = counterManager.getStrongCounter("rate_limiter");

// on each operation, invoke
try {
   counter.incrementAndGet().get();
   // continue with operation
} catch (InterruptedException e) {
   Thread.currentThread().interrupt();
} catch (ExecutionException e) {
   if (e.getCause() instanceof CounterOutOfBoundsException) {
```

```
      // maximum rate. discard operation
      return;
    } else {
      // unexpected error, handling property
    }
}
```

NOTE

The **lifespan** parameter is an experimental capability and may be removed in a future version.

11.1.3.2. The WeakCounter interface: when speed is needed

The **WeakCounter** stores the counter's value in multiple keys in Data Grid cache. The number of keys created is configured by the **concurrency-level** attribute. Each key stores a partial state of the counter's value and it can be updated concurrently. It main advantage over the **StrongCounter** is the lower contention in the cache. On other hand, the read of its value is more expensive and bounds are not allowed.

WARNING

The reset operation should be handled with caution. It is **not** atomic and it produces intermediates values. These value may be seen by a read operation and by any listener registered.

A **WeakCounter** can be retrieved from the **CounterManager** by using the **getWeakCounter()** method. As an example:

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

11.1.3.2.1. Weak Counter Interface

The **WeakCounter** adds the following methods:

```
default CompletableFuture<Void> increment() {
  return add(1L);
}

default CompletableFuture<Void> decrement() {
  return add(-1L);
}

CompletableFuture<Void> add(long delta);
```

They are similar to the `StrongCounter's methods but they don't return the new value.

11.1.3.2.2. Uses cases

The weak counter fits best in uses cases where the result of the update operation is not needed or the counter's value is not required too often. Collecting statistics is a good example of such an use case.

11.1.3.2.3. Examples

Below, there is an example of the weak counter usage.

```
WeakCounter counter = counterManager.getWeakCounter("my_counter");

// increment the counter and check its result
counter.increment().get();
System.out.println("current value is " + counter.getValue());

CompletableFuture<Void> f = counter.add(-100);
//do some work
f.get(); //wait until finished
System.out.println("current value is " + counter.getValue().get());

//using the functional API
counter.reset().whenComplete((aVoid, throwable) -> System.out.println("Reset done " + (throwable
== null ? "successfully" : "unsuccessfully"))).get();
System.out.println("current value is " + counter.getValue().get());
```

11.1.4. Notifications and Events

Both strong and weak counter supports a listener to receive its updates events. The listener must implement **CounterListener** and it can be registered by the following method:

```
<T extends CounterListener> Handle<T> addListener(T listener);
```

The **CounterListener** has the following interface:

```
public interface CounterListener {
   void onUpdate(CounterEvent entry);
}
```

The **Handle** object returned has the main goal to remove the **CounterListener** when it is not longer needed. Also, it allows to have access to the **CounterListener** instance that is it handling. It has the following interface:

```
public interface Handle<T extends CounterListener> {
   T getCounterListener();
   void remove();
}
```

Finally, the **CounterEvent** has the previous and current value and state. It has the following interface:

```
public interface CounterEvent {
   long getOldValue();
   State getOldState();
```

```
    long getNewValue();
    State getNewState();
}
```

NOTE

The state is always **State.VALID** for unbounded strong counter and weak counter. **State.LOWER_BOUND_REACHED** and **State.UPPER_BOUND_REACHED** are only valid for bounded strong counters.

WARNING

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

CHAPTER 12. LISTENERS AND NOTIFICATIONS

Use listeners with Data Grid to get notifications when events occur for the Cache Manager or for caches.

12.1. LISTENERS AND NOTIFICATIONS

Data Grid offers a listener API, where clients can register for and get notified when events take place. This annotation-driven API applies to 2 different levels: cache level events and Cache Manager level events.

Events trigger a notification which is dispatched to listeners. Listeners are simple POJOs annotated with @Listener and registered using the methods defined in the Listenable interface.

Both Cache and CacheManager implement Listenable, which means you can attach listeners to either a cache or a Cache Manager, to receive either cache-level or Cache Manager-level notifications.

For example, the following class defines a listener to print out some information every time a new entry is added to the cache, in a non blocking fashion:

```
@Listener
public class PrintWhenAdded {
  Queue<CacheEntryCreatedEvent> events = new ConcurrentLinkedQueue<>();

  @CacheEntryCreated
  public CompletionStage<Void> print(CacheEntryCreatedEvent event) {
    events.add(event);
    return null;
  }

}
```

For more comprehensive examples, please see the Javadocs for @Listener.

12.2. CACHE-LEVEL NOTIFICATIONS

Cache-level events occur on a per-cache basis, and by default are only raised on nodes where the events occur. Note in a distributed cache these events are only raised on the owners of data being affected. Examples of cache-level events are entries being added, removed, modified, etc. These events trigger notifications to listeners registered to a specific cache.

Please see the Javadocs on the org.infinispan.notifications.cachelistener.annotation package for a comprehensive list of all cache-level notifications, and their respective method-level annotations.

NOTE

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

Cluster listeners

The cluster listeners should be used when it is desirable to listen to the cache events on a single node.

To do so all that is required is set to annotate your listener as being clustered.

```
@Listener (clustered = true)
public class MyClusterListener { .... }
```

There are some limitations to cluster listeners from a non clustered listener.

1. A cluster listener can only listen to **@CacheEntryModified**, **@CacheEntryCreated**, **@CacheEntryRemoved** and **@CacheEntryExpired** events. Note this means any other type of event will not be listened to for this listener.

2. Only the post event is sent to a cluster listener, the pre event is ignored.

Event filtering and conversion

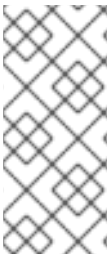
All applicable events on the node where the listener is installed will be raised to the listener. It is possible to dynamically filter what events are raised by using a KeyFilter (only allows filtering on keys) or CacheEventFilter (used to filter for keys, old value, old metadata, new value, new metadata, whether command was retried, if the event is before the event (ie. isPre) and also the command type).

The example here shows a simple **KeyFilter** that will only allow events to be raised when an event modified the entry for the key **Only Me**.

```
public class SpecificKeyFilter implements KeyFilter<String> {
   private final String keyToAccept;

   public SpecificKeyFilter(String keyToAccept) {
     if (keyToAccept == null) {
       throw new NullPointerException();
     }
     this.keyToAccept = keyToAccept;
   }

   public boolean accept(String key) {
     return keyToAccept.equals(key);
   }
}

...
cache.addListener(listener, new SpecificKeyFilter("Only Me"));
...
```

This can be useful when you want to limit what events you receive in a more efficient manner.

There is also a CacheEventConverter that can be supplied that allows for converting a value to another before raising the event. This can be nice to modularize any code that does value conversions.



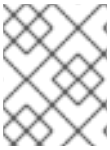
NOTE

The mentioned filters and converters are especially beneficial when used in conjunction with a Cluster Listener. This is because the filtering and conversion is done on the node where the event originated and not on the node where event is listened to. This can provide benefits of not having to replicate events across the cluster (filter) or even have reduced payloads (converter).

Initial State Events

When a listener is installed it will only be notified of events after it is fully installed.

It may be desirable to get the current state of the cache contents upon first registration of listener by having an event generated of type **@CacheEntryCreated** for each element in the cache. Any additionally generated events during this initial phase will be queued until appropriate events have been raised.



NOTE

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

Duplicate Events

It is possible in a non transactional cache to receive duplicate events. This is possible when the primary owner of a key goes down while trying to perform a write operation such as a put.

Data Grid internally will rectify the put operation by sending it to the new primary owner for the given key automatically, however there are no guarantees in regards to if the write was first replicated to backups. Thus more than 1 of the following write events (**CacheEntryCreatedEvent**, **CacheEntryModifiedEvent** & **CacheEntryRemovedEvent**) may be sent on a single operation.

If more than one event is generated Data Grid will mark the event that it was generated by a retried command to help the user to know when this occurs without having to pay attention to view changes.

```
@Listener
public class MyRetryListener {
  @CacheEntryModified
  public void entryModified(CacheEntryModifiedEvent event) {
    if (event.isCommandRetried()) {
      // Do something
    }
  }
}
```

Also when using a **CacheEventFilter** or **CacheEventConverter** the EventType contains a method **isRetry** to tell if the event was generated due to retry.

12.3. CACHE MANAGER NOTIFICATIONS

Events that occur on a Cache Manager level are cluster-wide and involve events that affect all caches created by a single Cache Manager. Examples of Cache Manager events are nodes joining or leaving a cluster, or caches starting or stopping.

See the org.infinispan.notifications.cachemanagerlistener.annotation package for a comprehensive list of all Cache Manager notifications, and their respective method-level annotations.

12.4. SYNCHRONICITY OF EVENTS

By default, all async notifications are dispatched in the notification thread pool. Sync notifications will delay the operation from continuing until the listener method completes or the CompletionStage completes (the former causing the thread to block). Alternatively, you could annotate your listener as *asynchronous* in which case the operation will continue immediately, while the notification is completed asynchronously on the notification thread pool. To do this, simply annotate your listener such:

Asynchronous Listener

```
@Listener (sync = false)
public class MyAsyncListener {
  @CacheEntryCreated
  void listen(CacheEntryCreatedEvent event) { }
}
```

Blocking Synchronous Listener

```
@Listener
public class MySyncListener {
  @CacheEntryCreated
  void listen(CacheEntryCreatedEvent event) { }
}
```

Non-Blocking Listener

```
@Listener
public class MyNonBlockingListener {
  @CacheEntryCreated
  CompletionStage<Void> listen(CacheEntryCreatedEvent event) { }
}
```

Asynchronous thread pool

To tune the thread pool used to dispatch such asynchronous notifications, use the **<listener-executor />** XML element in your configuration file.