

{brandname} 10.0 User Guide

Table of Contents

1. Introduction	1
1.1. What is {brandname} ?	1
1.2. Why use {brandname} ?	1
1.2.1. As a local cache	1
1.2.2. As a clustered cache	1
1.2.3. As a clustering building block for your applications	1
1.2.4. As a remote cache	1
1.2.5. As a data grid	2
1.2.6. As a geographical backup for your data	2
2. The Embedded CacheManager	3
2.1. Configuration	3
2.1.1. Configuring caches declaratively	4
2.1.2. Configuring caches programmatically	7
2.1.3. Configuration Migration Tools	12
2.1.4. Clustered Configuration	12
2.1.5. Inline JGroups configurations	14
2.2. Obtaining caches	17
2.3. Clustering Information	18
2.3.1. Member Information	18
2.3.2. Other methods	18
3. The Cache API	19
3.1. The Cache interface	19
3.1.1. Performance Concerns of Certain Map Methods	19
3.1.2. Mortal and Immortal Data	19
3.1.3. Expiration and Mortal Data	19
3.1.4. putForExternalRead operation	20
3.2. The AdvancedCache interface	21
3.2.1. Flags	21
3.2.2. Custom Interceptors	21
3.3. Listeners and Notifications	21
3.3.1. Cache-level notifications	22
3.3.2. Cache manager-level notifications	24
3.3.3. Synchronicity of events	24
3.4. Asynchronous API	25
3.4.1. Why use such an API?	26
3.4.2. Which processes actually happen asynchronously?	26
3.4.3. Notifying futures	26
3.4.4. Further reading	27

3.5. Invocation Flags	27
3.5.1. Examples	27
3.6. Functional Map API	28
3.6.1. Asynchronous and Lazy	28
3.6.2. Function transparency	29
3.6.3. Constructing Functional Maps	29
3.6.4. Read-Only Map API	29
3.6.5. Write-Only Map API	30
3.6.6. Read-Write Map API	32
3.6.7. Metadata Parameter Handling	33
3.6.8. Invocation Parameter	35
3.6.9. Functional Listeners	36
3.6.10. Marshalling of Functions	39
3.6.11. Use Cases for Functional API	42
3.7. Encoding	42
3.7.1. Overview	42
3.7.2. Default encoders	43
3.7.3. Overriding programmatically	44
3.7.4. Defining custom Encoders	44
3.7.5. MediaType	46
4. Eviction and Data Container	50
4.1. Enabling Eviction	50
4.1.1. Eviction strategy	50
4.1.2. Eviction types	51
4.1.3. Storage type	51
4.1.4. More defaults	51
4.2. Expiration	52
4.2.1. Difference between Eviction and Expiration	53
4.3. Expiration details	53
4.3.1. Maximum Idle Expiration	53
4.3.2. Configuration	54
4.3.3. Memory Based Eviction Configuration	55
4.3.4. Default values	55
4.3.5. Using expiration	55
4.4. Expiration designs	56
5. Persistence	57
5.1. Configuration	57
5.2. Cache Passivation	61
5.2.1. Limitations	61
5.2.2. Cache Loader Behavior with Passivation Disabled vs Enabled	61
5.3. Cache Loaders and transactional caches	62

5.4. Write-Through And Write-Behind Caching	62
5.4.1. Write-Through (Synchronous).....	63
5.4.2. Write-Behind (Asynchronous).....	63
5.4.3. Segmented Stores	64
5.5. Filesystem based cache stores	64
5.5.1. Single File Store	65
5.5.2. Soft-Index File Store	66
5.6. JDBC String based Cache Store	67
5.6.1. Connection management (pooling)	67
5.6.2. Sample configurations	68
5.7. Remote store	70
5.7.1. Segmentation support	70
5.7.2. Sample Usage	70
5.8. Cluster cache loader	71
5.9. Command-Line Interface cache loader	72
5.10. RocksDB Cache Store	72
5.10.1. Introduction	73
5.10.2. Segmentation support	73
5.10.3. Configuration	73
5.10.4. Additional References	75
5.11. JPA Cache Store	75
5.11.1. Sample Usage	75
5.11.2. Configuration	77
5.11.3. Additional References	78
5.12. Custom Cache Stores	78
5.12.1. HotRod Deployment	79
5.13. Store Migrator	79
5.13.1. Migrating Cache Stores	79
5.13.2. Store Migrator Properties	82
5.14. SPI	84
5.15. More implementations	86
6. Clustering	87
6.1. Which cache mode should I use?	87
6.2. Local Mode	88
6.2.1. Simple Cache	88
6.3. Invalidation Mode	89
6.4. Replicated Mode	92
6.5. Distribution Mode	94
6.5.1. Read consistency	96
6.5.2. Key Ownership	97
6.5.3. Initial cluster size	99

6.5.4. L1 Caching	99
6.5.5. Server Hinting	101
6.5.6. Key affinity service	101
6.5.7. The Grouping API	103
6.6. Scattered Mode	106
6.7. Asynchronous Options	107
6.7.1. Asynchronous Communications	107
6.7.2. Asynchronous API	107
6.7.3. Return Values	107
6.8. Partition handling	108
6.8.1. Split brain	109
6.8.2. Successive nodes stopped	111
6.8.3. Conflict Manager	112
6.8.4. Usage	113
6.8.5. Configuring partition handling	114
6.8.6. Monitoring and administration	115
7. Marshalling	117
7.1. The Role Of JBoss Marshalling	117
7.2. Support For Non-Serializable Objects	117
7.2.1. Store As Binary	118
7.3. Advanced Configuration	119
7.3.1. Troubleshooting	119
7.4. User Defined Externalizers	123
7.4.1. Benefits of Externalizers	123
7.4.2. User Friendly Externalizers	124
7.4.3. Advanced Externalizers	125
8. Transactions	131
8.1. Configuring transactions	132
8.2. Isolation levels	134
8.3. Transaction locking	134
8.3.1. Pessimistic transactional cache	134
8.3.2. Optimistic transactional cache	135
8.3.3. What do I need - pessimistic or optimistic transactions?	135
8.4. Write Skews	136
8.4.1. Forcing write locks on keys in pessimitic transactions	136
8.5. Dealing with exceptions	137
8.6. Enlisting Synchronizations	137
8.7. Batching	137
8.7.1. API	138
8.7.2. Batching and JTA	138
8.8. Transaction recovery	139

8.8.1. When to use recovery	139
8.8.2. How does it work	139
8.8.3. Configuring recovery	139
8.8.4. Recovery cache	139
8.8.5. Integration with the transaction manager	140
8.8.6. Reconciliation	140
8.8.7. Want to know more?	142
8.9. Total Order based commit protocol	142
8.9.1. Overview	143
8.9.2. Configuration	146
8.9.3. When to use it?	147
9. Locking and Concurrency	148
9.1. Locking implementation details	148
9.1.1. How does it work in clustered caches?	148
9.1.2. Transactional caches	149
9.1.3. Isolation levels	149
9.1.4. The LockManager	149
9.1.5. Lock striping	149
9.1.6. Concurrency levels	149
9.1.7. Lock timeout	150
9.1.8. Consistency	150
9.2. Data Versioning	150
10. Executing code in the Grid	152
10.1. Cluster Executor	152
10.1.1. Filtering execution nodes	152
10.1.2. Timeout	153
10.1.3. Single Node Submission	153
10.1.4. Example: PI Approximation	154
10.2. Streams	155
10.2.1. Common stream operations	156
10.2.2. Key filtering	156
10.2.3. Segment based filtering	156
10.2.4. Local/Invalidation	156
10.2.5. Example	156
10.3. Distribution/Replication/Scattered	157
10.3.1. Rehash Aware	157
10.3.2. Serialization	157
10.3.3. Parallel Computation	160
10.3.4. Task timeout	161
10.3.5. Injection	161
10.3.6. Distributed Stream execution	161

10.3.7. Key based rehash aware operators	163
10.3.8. Intermediate operation exceptions	163
10.3.9. Examples	164
10.4. Locked Streams	167
11. Indexing and Querying	168
11.1. Overview	168
11.2. Embedded Querying	168
11.2.1. Quick example	168
11.2.2. Indexing	171
11.2.3. Querying APIs	186
11.3. Remote Querying	204
11.3.1. Storing Protobuf encoded entities	204
11.3.2. Using annotations	207
11.3.3. Indexing of Protobuf encoded entries	208
11.3.4. A remote query example	208
11.3.5. Analysis	209
11.4. Statistics	211
11.5. Performance Tuning	212
11.5.1. Batch writing in SYNC mode	212
11.5.2. Writing using async mode	212
11.5.3. Index reader async strategy	213
11.5.4. Lucene Options	213
12. Clustered Counters	214
12.1. Installation and Configuration	214
12.1.1. List counter names	217
12.2. The CounterManager interface	217
12.2.1. Remove a counter via CounterManager	218
12.3. The Counter	218
12.3.1. The StrongCounter interface: when the consistency or bounds matters.	219
12.3.2. The WeakCounter interface: when speed is needed	223
12.4. Notifications and Events	224
13. Clustered Lock	226
13.1. Installation	226
13.2. ClusteredLock Configuration	226
13.2.1. Ownership	226
13.2.2. Reentrancy	227
13.3. ClusteredLockManager Interface	227
13.4. ClusteredLock Interface	228
13.4.1. Usage Examples	229
13.5. ClusteredLockManager Configuration	229
14. Multimap Cache	231

14.1. Installation and configuration	231
14.2. MultimapCache API	231
14.2.1. CompletableFuture<Void> put(K key, V value)	232
14.2.2. CompletableFuture<Collection<V>> get(K key)	232
14.2.3. CompletableFuture<Boolean> remove(K key)	232
14.2.4. CompletableFuture<Boolean> remove(K key, V value)	232
14.2.5. CompletableFuture<Void> remove(Predicate<? super V> p)	232
14.2.6. CompletableFuture<Boolean> containsKey(K key)	232
14.2.7. CompletableFuture<Boolean> containsValue(V value)	233
14.2.8. CompletableFuture<Boolean> containsEntry(K key, V value)	233
14.2.9. CompletableFuture<Long> size()	233
14.2.10. boolean supportsDuplicates()	233
14.3. Creating a Multimap Cache	233
14.3.1. Embedded mode	233
14.3.2. Server mode	233
14.4. Limitations	233
14.4.1. Support for duplicates	234
14.4.2. Eviction	234
14.4.3. Transactions	234
15. CDI Support	235
15.1. Maven Dependencies	235
15.2. Embedded cache integration	235
15.2.1. Inject an embedded cache	235
15.2.2. Override the default embedded cache manager and configuration	237
15.2.3. Configure the transport for clustered use	238
15.3. Remote cache integration	239
15.3.1. Inject a remote cache	239
15.3.2. Override the default remote cache manager	240
15.4. Use a custom remote/embedded cache manager for one or more cache	241
15.5. Use JCache caching annotations	241
15.6. Use Cache events and CDI	243
16. JCache (JSR-107) provider	244
16.1. Dependencies	244
16.2. Create a local cache	244
16.3. Create a remote cache	245
16.4. Store and retrieve data	245
16.5. Comparing java.util.concurrent.ConcurrentMap and javax.cache.Cache APIs	246
16.6. Clustering JCache instances	247
17. Management Tooling	249
17.1. JMX	249
17.1.1. Understanding The Exposed MBeans	249

17.1.2. Enabling JMX Statistics	250
17.1.3. Monitoring cluster health	251
17.1.4. Multiple JMX Domains	251
17.1.5. Registering MBeans In Non-Default MBean Servers	251
17.1.6. Available MBeans	252
17.2. Command-Line Interface (CLI)	252
17.2.1. Commands	254
17.2.2. upgrade	260
17.2.3. version	260
17.2.4. Data Types	260
17.2.5. Time Values	261
17.3. Hawt.io	261
17.4. Writing plugins for other management tools	261
18. Custom Interceptors	262
18.1. Adding custom interceptors declaratively	262
18.2. Adding custom interceptors programmatically	262
18.3. Custom interceptor design	263
19. Running on Cloud Services	264
19.1. Generic Discovery protocols	264
19.1.1. TCPping	264
19.1.2. GossipRouter	265
19.2. Amazon Web Services	265
19.2.1. S3_PING	265
19.2.2. JDBC_PING	265
19.3. Microsoft Azure	266
19.3.1. AZURE_PING	266
19.4. Google Compute Engine	266
19.4.1. GOOGLE_PING	266
19.5. Kubernetes and OpenShift	266
19.5.1. Using Kubernetes and OpenShift Rolling Updates	267
19.5.2. Rolling upgrades with Kubernetes and OpenShift	269
20. Client/Server	270
20.1. Why Client/Server?	270
20.2. Why use embedded mode?	274
20.3. Server Modules	274
20.4. Which protocol should I use?	275
20.5. Using Hot Rod Server	275
20.6. Java Hot Rod client	276
20.6.1. Configuration	276
20.6.2. Authentication	277
20.6.3. Encryption	281

20.6.4. Basic API	284
20.6.5. RemoteCache(keySet .entrySet .values)	284
20.6.6. Remote Iterator	285
20.6.7. Versioned API	287
20.6.8. Async API	288
20.6.9. Streaming API	288
20.6.10. Creating Event Listeners	289
20.6.11. Removing Event Listeners	291
20.6.12. Filtering Events	291
20.6.13. Customizing Events	293
20.6.14. Filter and Custom Events	297
20.6.15. Event Marshalling	298
20.6.16. Listener State Handling	299
20.6.17. Listener Failure Handling	300
20.6.18. Near Caching	300
20.6.19. Unsupported methods	301
20.6.20. Return values	302
20.6.21. Hot Rod Transactions	302
20.6.22. Client Intelligence	309
20.6.23. Request Balancing	309
20.6.24. Persistent connections	310
20.6.25. Marshalling data	310
20.6.26. Reading data in different data formats	311
20.6.27. Statistics	312
20.6.28. Multi-Get Operations	312
20.6.29. Failover capabilities	312
20.6.30. Site Cluster Failover	312
20.6.31. Manual Site Cluster Switch	313
20.6.32. Monitoring the Hot Rod client	313
20.6.33. Concurrent Updates	313
20.6.34. Javadocs	316
20.7. REST Server	316
20.7.1. Running the REST server	316
20.7.2. Supported protocols	317
20.7.3. REST API	317
20.7.4. CORS	324
20.7.5. Client side code	325
20.8. Memcached Server	330
20.8.1. Client Encoding	330
20.8.2. Command Clarifications	330
20.8.3. Unsupported Features	330

20.8.4. Talking To {brandname} Memcached Servers From Non-Java Clients	332
20.9. Executing code in the Remote Grid	333
20.10. Scripting	333
20.10.1. Installing scripts	333
20.10.2. Script metadata	333
20.10.3. Script bindings	334
20.10.4. Script parameters	334
20.10.5. Running Scripts using the Hot Rod Java client	335
20.10.6. Distributed execution	335
20.11. Server Tasks	335
21. Compatibility Mode	338
21.1. Enable Compatibility Mode	338
21.1.1. Optional: Configuring Compatibility Marshaller	338
21.2. Code examples	339
22. Protocol Interoperability	340
22.1. Considerations with Media Types and Endpoint Interoperability	340
22.2. REST, Hot Rod, and Memcached Interoperability with Text-Based Storage	340
22.3. REST, Hot Rod, and Memcached Interoperability with Custom Java Objects	341
22.4. Java and Non-Java Client Interoperability with Protobuf	342
22.5. Custom Code Interoperability	343
22.5.1. Converting Data On Demand	343
22.5.2. Storing Data as POJOs	344
22.6. Deploying Entity Classes	345
22.7. Trying the Interoperability Demo	345
23. Security	346
23.1. Embedded Security	346
23.1.1. Embedded Permissions	346
23.1.2. Embedded API	347
23.1.3. Embedded Configuration	348
23.2. Security Audit	350
23.3. Cluster security	351
24. Integrations	353
24.1. Apache Spark	353
24.2. Apache Hadoop	353
24.3. Apache Lucene	353
24.3.1. Lucene compatibility	353
24.3.2. Maven dependencies	353
24.3.3. How to use it	354
24.3.4. Configuration	356
24.3.5. Using a CacheLoader	357
24.3.6. Storing the index in a database	357

24.3.7. Loading an existing Lucene Index	358
24.3.8. Architectural limitations	358
24.3.9. Suggestions for optimal performance	359
24.3.10. Demo	360
24.3.11. Additional Links	360
24.4. Directory Provider for Hibernate Search	360
24.4.1. Maven dependencies	360
24.4.2. How to use it	360
24.4.3. Configuration	361
24.4.4. Architecture considerations	361
24.5. JPA/Hibernate 2L Cache	361
24.5.1. Deployment Scenarios	363
24.5.2. Configuration Reference	366
24.5.3. Cache Strategies	372
24.5.4. Using minimal puts	373
24.6. JPA / Hibernate OGM	373
24.7. Using {brandname} with Spring	375
24.7.1. Spring Boot Starter	375
24.7.2. Setting Up {brandname} as a Spring Cache Provider	375
24.7.3. Adding Caching to Your Application	377
24.7.4. Configuring Timeouts for Cache Operations	378
24.7.5. Externalizing Sessions Using Spring Session	379
24.8. {brandname} modules for WildFly / EAP	380
24.8.1. Installation	380
24.8.2. Application Dependencies	381
24.8.3. Usage	383
24.8.4. Troubleshooting	386
25. Grid File System	387
25.1. WebDAV demo	388
26. Cross site replication	389
26.1. Sample deployment	389
26.1.1. Local cluster's jgroups.xml configuration	392
26.1.2. RELAY2 configuration file	392
26.2. Data replication	393
26.2.1. Non transactional caches	393
26.2.2. Transactional caches	393
26.3. Taking a site offline	394
26.3.1. Configuration	394
26.3.2. Bringing Sites Back Online	395
26.4. Pushing State Transfer to Sites	395
26.4.1. Handling join/leave nodes	396

26.4.2. Handling broken link between sites	396
26.4.3. System Administrator Operations	396
26.4.4. Configuration	397
26.5. Reference	398
27. Performing Rolling Upgrades	399
27.1. Setting Up a Target Cluster	399
27.2. Synchronizing Data from the Source Cluster	400
28. Extending {brandname}	402
28.1. Custom Commands	402
28.1.1. An Example	402
28.1.2. Preassigned Custom Command Id Ranges	402
28.2. Extending the configuration builders and parsers	403
29. Architectural Overview	404
29.1. Cache hierarchy	404
29.2. Commands	404
29.3. Visitors	405
29.4. Interceptors	405
29.5. Putting it all together	406
29.6. Subsystem Managers	406
29.6.1. DistributionManager	406
29.6.2. TransactionManager	406
29.6.3. RpcManager	406
29.6.4. LockManager	406
29.6.5. PersistenceManager	406
29.6.6. DataContainer	406
29.6.7. Configuration	407
29.7. ComponentRegistry	407

Chapter 1. Introduction

Welcome to the official {brandname} user guide. This comprehensive document will guide you through every last detail of {brandname}. Because of this, it can be a poor starting point if you are new to {brandname}.



For newbies, starting with the [Getting Started Guide](#) or one of the [Quickstarts](#) is probably a better bet.

The [Frequently Asked Questions](#) and [Glossary](#) are also useful documents to have alongside this user guide.

1.1. What is {brandname} ?

{brandname} is a distributed in-memory key/value data store with optional schema, available under the Apache License 2.0. It can be used both as an embedded Java library and as a language-independent service accessed remotely over a variety of protocols (Hot Rod, REST, Memcached and WebSockets). It offers advanced functionality such as transactions, events, querying and distributed processing as well as numerous integrations with frameworks such as the JCache API standard, CDI, Hibernate, WildFly, Spring Cache, Spring Session, Lucene, Spark and Hadoop.

1.2. Why use {brandname} ?

1.2.1. As a local cache

The primary use for {brandname} is to provide a fast in-memory cache of frequently accessed data. Suppose you have a slow data source (database, web service, text file, etc): you could load some or all of that data in memory so that it's just a memory access away from your code. Using {brandname} is better than using a simple ConcurrentHashMap, since it has additional useful features such as expiration and eviction.

1.2.2. As a clustered cache

If your data doesn't fit in a single node, or you want to invalidate entries across multiple instances of your application, {brandname} can scale horizontally to several hundred nodes.

1.2.3. As a clustering building block for your applications

If you need to make your application cluster-aware, integrate {brandname} and get access to features like topology change notifications, cluster communication and clustered execution.

1.2.4. As a remote cache

If you want to be able to scale your caching layer independently from your application, or you need to make your data available to different applications, possibly even using different languages / platforms, use {brandname} Server and its various clients.

1.2.5. As a data grid

Data you place in {brandname} doesn't have to be temporary: use {brandname} as your primary store and use its powerful features such as transactions, notifications, queries, distributed execution, distributed streams, analytics to process data quickly.

1.2.6. As a geographical backup for your data

{brandname} supports replication between clusters, allowing you to backup your data across geographically remote sites.

Chapter 2. The Embedded CacheManager

The CacheManager is {brandname}'s main entry point. You use a CacheManager to

- configure and obtain caches
- manage and monitor your nodes
- execute code across a cluster
- more...

Depending on whether you are embedding {brandname} in your application or you are using it remotely, you will be dealing with either an `EmbeddedCacheManager` or a `RemoteCacheManager`. While they share some methods and properties, be aware that there are semantic differences between them. The following chapters focus mostly on the *embedded* implementation. For details on the *remote* implementation refer to [Hot Rod Java Client](#).

CacheManagers are heavyweight objects, and we foresee no more than one CacheManager being used per JVM (unless specific setups require more than one; but either way, this would be a minimal and finite number of instances).

The simplest way to create a CacheManager is:

```
EmbeddedCacheManager manager = new DefaultCacheManager();
```

which starts the most basic, local mode, non-clustered cache manager with no caches. CacheManagers have a lifecycle and the default constructors also call `start()`. Overloaded versions of the constructors are available, that do not start the CacheManager, although keep in mind that CacheManagers need to be started before they can be used to create Cache instances.

Once constructed, CacheManagers should be made available to any component that require to interact with it via some form of application-wide scope such as JNDI, a ServletContext or via some other mechanism such as an IoC container.

When you are done with a CacheManager, you must stop it so that it can release its resources:

```
manager.stop();
```

This will ensure all caches within its scope are properly stopped, thread pools are shutdown. If the CacheManager was clustered it will also leave the cluster gracefully.

2.1. Configuration

{brandname} offers both declarative and programmatic configuration.

2.1.1. Configuring caches declaratively

Declarative configuration comes in a form of XML document that adheres to a provided {brandname} configuration XML [schema](#).

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

There are two main configuration abstractions in {brandname}: **global** and **cache**.

Global configuration

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

Cache configuration

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

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



Embedded and Server configuration use different schemas, but we strive to maintain them as compatible as possible so that you can easily migrate between the two.

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

infinispan.xml

```
<infinispan />
```

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

Declarative configuration is the most common approach to configuring {brandname} cache instances. In order to read XML configuration files one would typically construct an instance of

DefaultCacheManager by pointing to an XML file containing {brandname} configuration. Once the configuration file is read you can obtain reference to the default cache instance.

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

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

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

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

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

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

Cache configuration templates

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

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

```

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

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

```

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

```

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

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

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

```

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



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

Cache configuration wildcards

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

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

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



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

Declarative configuration reference

For more details on the declarative configuration schema, refer to the [configuration reference](#). If you are using XML editing tools for configuration writing you can use the provided `{brandname}` [schema](#) to assist you.

2.1.2. Configuring caches programmatically

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

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

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


```

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

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

```

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

```

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

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

```

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

```

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

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

```

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

ConfigurationBuilder Programmatic Configuration API

While the above paragraph shows how to combine declarative and programmatic configuration, starting from an XML configuration is completely optional. The `ConfigurationBuilder` fluent

interface style allows for easier to write and more readable programmatic configuration. This approach can be used for both the global and the cache level configuration. GlobalConfiguration objects are constructed using GlobalConfigurationBuilder while Configuration objects are built using ConfigurationBuilder. Let's look at some examples on configuring both global and cache level options with this API:

Configuring the transport

One of the most commonly configured global option is the transport layer, where you indicate how an {brandname} node will discover the others:

```
GlobalConfiguration globalConfig = new GlobalConfigurationBuilder().transport()
    .defaultTransport()
    .clusterName("qa-cluster")
    .addProperty("configurationFile", "jgroups-tcp.xml")
    .machineId("qa-machine").rackId("qa-rack")
    .build();
```

Using a custom JChannel

If you want to construct the JGroups [JChannel](#) by yourself, you can do so.



The JChannel must not be already connected.

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

Enabling JMX MBeans and statistics

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

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

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

Further options at the global JMX statistics level allows you to configure the cache manager name

which comes handy when you have multiple cache managers running on the same system, or how to locate the JMX MBean Server:

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

Configuring the thread pools

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

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

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

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

Or you can configure [eviction and expiration settings](#):

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

Configuring transactions and locking

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

enable JMX management and statistics gathering too:

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

Configuring cache stores

Configuring {brandname} with chained cache stores is simple too:

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

Advanced programmatic configuration

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

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

Or, add custom interceptors:

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

For information on the individual configuration options, please check the [configuration guide](#).

2.1.3. Configuration Migration Tools

The configuration format of {brandname} has changed since schema version 6.0 in order to align the embedded schema with the one used by the server. For this reason, when upgrading to schema 7.x or later, you should use the configuration converter included in the *all* distribution. Simply invoke it from the command-line passing the old configuration file as the first parameter and the name of the converted file as the second parameter.

To convert on Unix/Linux/macOS:

```
bin/config-converter.sh oldconfig.xml newconfig.xml
```

on Windows:

```
bin\config-converter.bat oldconfig.xml newconfig.xml
```



If you wish to help write conversion tools from other caching systems, please contact [infinispan-dev](#).

2.1.4. Clustered Configuration

{brandname} uses [JGroups](#) for network communications when in clustered mode. {brandname} ships with *pre-configured* JGroups stacks that make it easy for you to jump-start a clustered configuration.

Using an external JGroups file

If you are configuring your cache programmatically, all you need to do is:

```
GlobalConfiguration gc = new GlobalConfigurationBuilder()
    .transport().defaultTransport()
    .addProperty("configurationFile", "jgroups.xml")
    .build();
```

and if you happen to use an XML file to configure {brandname}, just use:

```

<infinispan>
  <jgroups>
    <stack-file name="external-file" path="jgroups.xml"/>
  </jgroups>
  <cache-container default-cache="replicatedCache">
    <transport stack="external-file" />
    <replicated-cache name="replicatedCache"/>
  </cache-container>

  ...

</infinispan>

```

In both cases above, {brandname} looks for *jgroups.xml* first in your classpath, and then for an absolute path name if not found in the classpath.

Use one of the pre-configured JGroups files

{brandname} ships with a few different JGroups files (packaged in *infinispan-core.jar*) which means they will already be on your classpath by default. All you need to do is specify the file name, e.g., instead of *jgroups.xml* above, specify */default-configs/default-jgroups-tcp.xml*.

The configurations available are:

- *default-jgroups-udp.xml* - Uses UDP as a transport, and UDP multicast for discovery. Usually suitable for larger (over 100 nodes) clusters *or* if you are using [replication](#) or [invalidation](#). Minimises opening too many sockets.
- *default-jgroups-tcp.xml* - Uses TCP as a transport and UDP multicast for discovery. Better for smaller clusters (under 100 nodes) *only if* you are using [distribution](#), as TCP is more efficient as a point-to-point protocol
- *default-jgroups-ec2.xml* - Uses TCP as a transport and [S3_PING](#) for discovery. Suitable on [Amazon EC2](#) nodes where UDP multicast isn't available.
- *default-jgroups-kubernetes.xml* - Uses TCP as a transport and [KUBE_PING](#) for discovery. Suitable on [Kubernetes](#) and [OpenShift](#) nodes where UDP multicast is not always available.

When using the XML configuration, the *tcp* and *udp* stacks are pre-declared for you, so that you don't need to do it yourself. Just reference them directly in the transport:

```

<infinispan>
  <cache-container default-cache="replicatedCache">
    <transport stack="tcp" />

    ...

  </cache-container>
</infinispan>

```

2.1.5. Inline JGroups configurations

The Infinispan XML configuration also allows you to embed JGroups configurations without requiring an external file:

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

  ...

</cache-container>
</infinispan>
```

JGroups stacks can be quite complex: most of the time, you only need to use a different discovery mechanism, add security, or make a slight tweak to a couple of parameters. For these cases, you can use stack inheritance to base your configuration off a known "good" JGroups stack. The following example creates a new `gossip-prod` based on the `prod` stack defined above, replacing the `MPING` protocol with `TCPGOSSIP`, increasing the `VERIFY_SUSPECT` timeout, removing the `FD_SOCK` protocol and adding `SYM_ENCRYPT`:

```

...

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

...

```

The `stack.combine` attribute determines the type of the operation:

- **COMBINE**: overrides an existing protocol's attributes
- **REPLACE**: replaces an existing protocol, identified by the `stack.position` attribute. If this attribute is missing, it defaults to the same protocol, in which case all non-specified attributes are reset to their defaults.
- **INSERT_AFTER**: inserts a protocol after another protocol identified by the `stack.position` attribute.
- **REMOVE**: removes the protocol.

Tuning JGroups settings

The settings above can be further tuned without editing the XML files themselves. Passing in certain system properties to your JVM at startup can affect the behaviour of some of these settings. The table below shows you which settings can be configured in this way. E.g.,

```
$ java -cp ... -Djgroups.tcp.port=1234 -Djgroups.tcp.address=10.11.12.13
```

Table 1. *default-jgroups-udp.xml*

System Property	Description	Default	Required?
jgroups.udp.mcast_add r	IP address to use for multicast (both for communications and discovery). Must be a valid Class D IP address, suitable for IP multicast.	228.6.7.8	No

jgroups.udp.mcast_port	Port to use for multicast socket	46655	No
jgroups.udp.ip_ttl	Specifies the time-to-live (TTL) for IP multicast packets. The value here refers to the number of network hops a packet is allowed to make before it is dropped	2	No

Table 2. default-jgroups-tcp.xml

System Property	Description	Default	Required?
jgroups.tcp.address	IP address to use for the TCP transport.	127.0.0.1	No
jgroups.tcp.port	Port to use for TCP socket	7800	No
jgroups.udp.mcast_address	IP address to use for multicast (for discovery). Must be a valid Class D IP address, suitable for IP multicast.	228.6.7.8	No
jgroups.udp.mcast_port	Port to use for multicast socket	46655	No
jgroups.udp.ip_ttl	Specifies the time-to-live (TTL) for IP multicast packets. The value here refers to the number of network hops a packet is allowed to make before it is dropped	2	No

Table 3. default-jgroups-ec2.xml

System Property	Description	Default	Required?
jgroups.tcp.address	IP address to use for the TCP transport.	127.0.0.1	No
jgroups.tcp.port	Port to use for TCP socket	7800	No
jgroups.s3.access_key	The Amazon S3 access key used to access an S3 bucket		No
jgroups.s3.secret_access_key	The Amazon S3 secret key used to access an S3 bucket		No

jgroups.s3.bucket	Name of the Amazon S3 bucket to use. Must be unique and must already exist		No
-------------------	--	--	----

Table 4. *default-jgroups-kubernetes.xml*

System Property	Description	Default	Required?
jgroups.tcp.address	IP address to use for the TCP transport.	eth0	No
jgroups.tcp.port	Port to use for TCP socket	7800	No

Further reading

JGroups also supports more system property overrides, details of which can be found on this page: [SystemProps](#)

In addition, the JGroups configuration files shipped with {brandname} are intended as a jumping off point to getting something up and running, and working. More often than not though, you will want to fine-tune your JGroups stack further to extract every ounce of performance from your network equipment. For this, your next stop should be the JGroups manual which has a [detailed section](#) on configuring each of the protocols you see in a JGroups configuration file.

2.2. Obtaining caches

After you configure the [CacheManager](#), you can obtain and control caches.

Invoke the `getCache()` 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.

Invoke the `createCache()` method to create caches dynamically across the entire cluster, as follows:

```
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.

Use the PERMANENT flag to ensure that caches can survive restarts, as follows:

```
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\(\)](#).

2.3. Clustering Information

The `EmbeddedCacheManager` has quite a few methods to provide information as to how the cluster is operating. The following methods only really make sense when being used in a clustered environment (that is when a Transport is configured).

2.3.1. Member Information

When you are using a cluster it is very important to be able to find information about membership in the cluster including who is the owner of the cluster.

[getMembers\(\)](#)

The `getMembers()` method returns all of the nodes in the current cluster.

[getCoordinator\(\)](#)

The `getCoordinator()` method will tell you which one of the members is the coordinator of the cluster. For most intents you shouldn't need to care who the coordinator is. You can use `isCoordinator()` method directly to see if the local node is the coordinator as well.

2.3.2. Other methods

[getTransport\(\)](#)

This method provides you access to the underlying Transport that is used to send messages to other nodes. In most cases a user wouldn't ever need to go to this level, but if you want to get Transport specific information (in this case JGroups) you can use this mechanism.

[getStats\(\)](#)

The stats provided here are coalesced from all of the active caches in this manager. These stats can be useful to see if there is something wrong going on with your cluster overall.

Chapter 3. The Cache API

3.1. The Cache interface

{brandname}'s Caches are manipulated through the [Cache](#) interface.

A Cache 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 {brandname}'s Cache should be trivial.

3.1.1. Performance Concerns of Certain Map Methods

Certain methods exposed in Map have certain performance consequences when used with {brandname}, 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.

For more performance tips, have a look at our [Performance Guide](#).

3.1.2. Mortal and Immortal Data

Further to simply storing entries, {brandname}'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* , {brandname} also supports *maxIdle* as an additional metric with which to determine expiration. Any combination of lifespans or maxIdles can be used.

3.1.3. Expiration and Mortal Data

See [Expiration](#) for more information about using mortal data with {brandname}.

3.1.4. putForExternalRead operation

{brandname}'s [Cache](#) class contains a different 'put' operation called [putForExternalRead](#) . This operation is particularly useful when {brandname} 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;
}
```

Please 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.

3.2. The AdvancedCache interface

In addition to the simple Cache interface, {brandname} offers an [AdvancedCache](#) interface, geared towards extension authors. The AdvancedCache offers the ability to inject custom interceptors, 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();
```

3.2.1. 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");
```

3.2.2. Custom Interceptors

The AdvancedCache interface also offers advanced developers a mechanism with which to attach custom interceptors. Custom interceptors allow developers to alter the behavior of the cache API methods, and the AdvancedCache interface allows developers to attach these interceptors programmatically, at run-time. See the AdvancedCache Javadocs for more details.

For more information on writing custom interceptors, see [Custom Interceptors](#).

3.3. Listeners and Notifications

{brandname} 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 [POJO](#) s 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](#).

3.3.1. 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.



Please refer to the [Javadocs on the `org.infinispan.notifications.cachelistener.annotation` package](#) for the list of cache-level notifications available in {brandname}.

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;
    }

    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.



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.



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.

{brandname} 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 {brandname} 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.

3.3.2. Cache manager-level notifications

Cache manager-level events occur on a cache manager. These too are global and cluster-wide, but involve events that affect all caches created by a single cache manager. Examples of cache manager-level events are nodes joining or leaving a cluster, or caches starting or stopping.

Please see the [Javadocs on the org.infinispan.notifications.cachemanagerlistener.annotation package](#) for a comprehensive list of all cache manager-level notifications, and their respective method-level annotations.

3.3.3. 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.

3.4. Asynchronous API

In addition to synchronous API methods like `Cache.put()` , `Cache.remove()` , etc., {brandname} 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 `Future` containing the actual result of the operation.

For example, in a cache parameterized as `Cache<String, String>`, `Cache.put(String key, String value)` returns a `String`. `Cache.putAsync(String key, String value)` would return a `Future<String>`.

3.4.1. 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<Future<?>> futures = new HashSet<Future<?>>();
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 (Future<?> f: futures) f.get();
```

3.4.2. Which processes actually happen asynchronously?

There are 4 things in {brandname} 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

As of {brandname} 4.0, 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. In future, we plan to take these offline as well. See [this developer mail list thread](#) about this topic.

3.4.3. Notifying futures

Strictly, these methods do not return JDK Futures, but rather a sub-interface known as a [NotifyingFuture](#). The main difference is that you can attach a listener to a NotifyingFuture such that you could be notified when the future completes. Here is an example of making use of a notifying future:

```
FutureListener futureListener = new FutureListener() {

    public void futureDone(Future future) {
        try {
            future.get();
        } catch (Exception e) {
            // Future did not complete successfully
            System.out.println("Help!");
        }
    }
};

cache.putAsync("key", "value").attachListener(futureListener);
```

3.4.4. Further reading

The Javadocs on the [Cache](#) interface has some examples on using the asynchronous API, as does [this article](#) by Manik Surtani introducing the API.

3.5. Invocation Flags

An important aspect of getting the most of {brandname} is the use of per-invocation flags in order to provide specific behaviour to each particular cache call. By doing this, some important optimizations can be implemented potentially saving precious time and network resources. One of the most popular usages of flags can be found right in Cache API, underneath the [putForExternalRead\(\)](#) method which is used to load an {brandname} cache with data read from an external resource. In order to make this call efficient, {brandname} basically calls a normal put operation passing the following flags: [FAIL_SILENTLY](#) , [FORCE_ASYNCHRONOUS](#) , [ZERO_LOCK_ACQUISITION_TIMEOUT](#)

What {brandname} is doing here is effectively saying that when putting data read from external read, it will use an almost-zero lock acquisition time and that if the locks cannot be acquired, it will fail silently without throwing any exception related to lock acquisition. It also specifies that regardless of the cache mode, if the cache is clustered, it will replicate asynchronously and so won't wait for responses from other nodes. The combination of all these flags make this kind of operation very efficient, and the efficiency comes from the fact this type of *putForExternalRead* calls are used with the knowledge that client can always head back to a persistent store of some sorts to retrieve the data that should be stored in memory. So, any attempt to store the data is just a best effort and if not possible, the client should try again if there's a cache miss.

3.5.1. Examples

If you want to use these or any other flags available, which by the way are described in detail the [Flag enumeration](#) , you simply need to get hold of the advanced cache and add the flags you need via the [withFlags\(\)](#) method call. For example:

```
Cache cache = ...
cache.getAdvancedCache()
    .withFlags(Flag.SKIP_CACHE_STORE, Flag.CACHE_MODE_LOCAL)
    .put("local", "only");
```

It's worth noting that these flags are only active for the duration of the cache operation. If the same flags need to be used in several invocations, even if they're in the same transaction, `withFlags()` needs to be called repeatedly. Clearly, if the cache operation is to be replicated in another node, the flags are carried over to the remote nodes as well.

Suppressing return values from a `put()` or `remove()`

Another very important use case is when you want a write operation such as `put()` to *not* return the previous value. To do that, you need to use two flags to make sure that in a distributed environment, no remote lookup is done to potentially get previous value, and if the cache is configured with a cache loader, to avoid loading the previous value from the cache store. You can see these two flags in action in the following example:

```
Cache cache = ...
cache.getAdvancedCache()
    .withFlags(Flag.SKIP_REMOTE_LOOKUP, Flag.SKIP_CACHE_LOAD)
    .put("local", "only")
```

For more information, please check the [Flag enumeration](#) javadoc.

3.6. Functional Map API

{brandname} 8 introduces a new experimental API for interacting with your data which takes advantage of the functional programming additions and improved asynchronous programming capabilities available in Java 8.

{brandname}'s [Functional Map API](#) is a distilled map-like asynchronous API which uses functions to interact with data.

3.6.1. Asynchronous and Lazy

Being an asynchronous API, all methods that return a single result, return a `CompletableFuture` which wraps the result, so you can use the resources of your system more efficiently by having the possibility to receive callbacks when the `CompletableFuture` has completed, or you can chain or compose them with other `CompletableFuture`.

For those operations that return multiple results, the API returns instances of a `Traversable` interface which offers a lazy pull-style API for working with multiple results. `Traversable`, being a lazy pull-style API, can still be asynchronous underneath since the user can decide to work on the traversable at a later stage, and the `Traversable` implementation itself can decide when to compute those results.

3.6.2. Function transparency

Since the content of the functions is transparent to {brandname}, the API has been split into 3 interfaces for read-only (`ReadOnlyMap`), read-write (`ReadWriteMap`) and write-only (`WriteOnlyMap`) operations respectively, in order to provide hints to the {brandname} internals on the type of work needed to support functions.

3.6.3. Constructing Functional Maps

To construct any of the read-only, write-only or read-write map instances, an {brandname} `AdvancedCache` is required, which is retrieved from the Cache Manager, and using the `AdvancedCache` , static method factory methods are used to create `ReadOnlyMap` , `ReadWriteMap` or `WriteOnlyMap`

```
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.functional.impl.*;

AdvancedCache<String, String> cache = ...

FunctionalMapImpl<String, String> functionalMap = FunctionalMapImpl.create(cache);
ReadOnlyMap<String, String> readOnlyMap = ReadOnlyMapImpl.create(functionalMap);
WriteOnlyMap<String, String> writeOnlyMap = WriteOnlyMapImpl.create(functionalMap);
ReadWriteMap<String, String> readWriteMap = ReadWriteMapImpl.create(functionalMap);
```



At this stage, the Functional Map API is experimental and hence the way `FunctionalMap`, `ReadOnlyMap`, `WriteOnlyMap` and `ReadWriteMap` are constructed is temporary.

3.6.4. Read-Only Map API

Read-only operations have the advantage that no locks are acquired for the duration of the operation. Here's an example on how to the equivalent operation for `Map.get(K)`:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;

ReadOnlyMap<String, String> readOnlyMap = ...
CompletableFuture<Optional<String>> readFuture = readOnlyMap.eval("key1",
ReadEntryView::find);
readFuture.thenAccept(System.out::println);
```

Read-only map also exposes operations to retrieve multiple keys in one go:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.api.functional.Traversable;

ReadOnlyMap<String, String> readOnlyMap = ...

Set<String> keys = new HashSet<>(Arrays.asList("key1", "key2"));
Traversable<String> values = readOnlyMap.evalMany(keys, ReadEntryView::get);
values.forEach(System.out::println);
```

Finally, read-only map also exposes methods to read all existing keys as well as entries, which include both key and value information.

Read-Only Entry View

The function parameters for read-only maps provide the user with a [read-only entry view](#) to interact with the data in the cache, which include these operations:

- `key()` method returns the key for which this function is being executed.
- `find()` returns a Java 8 `Optional` wrapping the value if present, otherwise it returns an empty optional. Unless the value is guaranteed to be associated with the key, it's recommended to use `find()` to verify whether there's a value associated with the key.
- `get()` returns the value associated with the key. If the key has no value associated with it, calling `get()` throws a `NoSuchElementException`. `get()` can be considered as a shortcut of `ReadEntryView.find().get()` which should be used only when the caller has guarantees that there's definitely a value associated with the key.
- `findMetaParam(Class<T> type)` allows metadata parameter information associated with the cache entry to be looked up, for example: entry lifespan, last accessed time...etc. See [Metadata Parameter Handling](#) to find out more.

3.6.5. Write-Only Map API

Write-only operations include operations that insert or update data in the cache and also removals. Crucially, a write-only operation does not attempt to read any previous value associated with the key. This is an important optimization since that means neither the cluster nor any persistence stores will be looked up to retrieve previous values. In the main {brandname} Cache, this kind of optimization was achieved using a local-only per-invocation flag, but the use case is so common that in this new functional API, this optimization is provided as a first-class citizen.

Using [write-only map API](#), an operation equivalent to `javax.cache.Cache (JCache)`'s void returning `put` can be achieved this way, followed by an attempt to read the stored value using the read-only map API:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;

WriteOnlyMap<String, String> writeOnlyMap = ...
ReadOnlyMap<String, String> readOnlyMap = ...

CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", "value1",
    (v, view) -> view.set(v));
CompletableFuture<String> readFuture = writeFuture.thenCompose(r ->
    readOnlyMap.eval("key1", ReadEntryView::get));
readFuture.thenAccept(System.out::println);
```

Multiple key/value pairs can be stored in one go using `evalMany` API:

```
WriteOnlyMap<String, String> writeOnlyMap = ...

Map<K, String> data = new HashMap<>();
data.put("key1", "value1");
data.put("key2", "value2");
CompletableFuture<Void> writerAllFuture = writeOnlyMap.evalMany(data, (v, view) ->
    view.set(v));
writerAllFuture.thenAccept(x -> "Write completed");
```

To remove all contents of the cache, there are two possibilities with different semantics. If using `evalAll` each cached entry is iterated over and the function is called with that entry's information. Using this method also results in listeners being invoked. See [functional listeners](#) for more information.

```
WriteOnlyMap<String, String> writeOnlyMap = ...

CompletableFuture<Void> removeAllFuture = writeOnlyMap.evalAll(WriteEntryView::remove
);
removeAllFuture.thenAccept(x -> "All entries removed");
```

The alternative way to remove all entries is to call `truncate` operation which clears the entire cache contents in one go without invoking any listeners and is best-effort:

```
WriteOnlyMap<String, String> writeOnlyMap = ...

CompletableFuture<Void> truncateFuture = writeOnlyMap.truncate();
truncateFuture.thenAccept(x -> "Cache contents cleared");
```

Write-Only Entry View

The function parameters for write-only maps provide the user with a [write-only entry view](#) to modify the data in the cache, which include these operations:

- `set(V, MetaParam.Writable...)` method allows for a new value to be associated with the cache entry for which this function is executed, and it optionally takes zero or more metadata parameters to be stored along with the value. See [Metadata Parameter Handling](#) for more information.
- `remove()` method removes the cache entry, including both value and metadata parameters associated with this key.

3.6.6. Read-Write Map API

The final type of operations we have are readwrite operations, and within this category CAS-like (CompareAndSwap) operations can be found. This type of operations require previous value associated with the key to be read and for locks to be acquired before executing the function. The vast majority of operations within `ConcurrentMap` and `JCache` APIs fall within this category, and they can easily be implemented using the [read-write map API](#). Moreover, with [read-write map API](#), you can make CASlike comparisons not only based on value equality but based on metadata parameter equality such as version information, and you can send back previous value or boolean instances to signal whether the CASlike comparison succeeded.

Implementing a write operation that returns the previous value associated with the cache entry is easy to achieve with the read-write map API:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;

ReadWriteMap<String, String> readWriteMap = ...

CompletableFuture<Optional<String>> readWriteFuture = readWriteMap.eval("key1",
"value1",
    (v, view) -> {
        Optional<V> prev = rw.find();
        view.set(v);
        return prev;
    });
readWriteFuture.thenAccept(System.out::println);
```

`ConcurrentMap.replace(K, V, V)` is a replace function that compares the value present in the map and if it's equals to the value passed in as first parameter, the second value is stored, returning a boolean indicating whether the replace was successfully completed. This operation can easily be implemented using the read-write map API:

```
ReadWriteMap<String, String> readWriteMap = ...

String oldValue = "old-value";
CompletableFuture<Boolean> replaceFuture = readWriteMap.eval("key1", "value1", (v,
view) -> {
    return view.find().map(prev -> {
        if (prev.equals(oldValue)) {
            rw.set(v);
            return true; // previous value present and equals to the expected one
        }
        return false; // previous value associated with key does not match
    }).orElse(false); // no value associated with this key
});
replaceFuture.thenAccept(replaced -> System.out.printf("Value was replaced? %s\n",
replaced));
```



The function in the example above captures `oldValue` which is an external value to the function which is valid use case.

Read-write map API contains `evalMany` and `evalAll` operations which behave similar to the write-only map offerings, except that they enable previous value and metadata parameters to be read.

Read-Write Entry View

The function parameters for read-write maps provide the user with the possibility to query the information associated with the key, including value and metadata parameters, and the user can also use this [read-write entry view](#) to modify the data in the cache.

The operations are exposed by read-write entry views are a union of the operations exposed by [read-only entry views](#) and [write-only entry views](#).

3.6.7. Metadata Parameter Handling

[Metadata parameters](#) provide extra information about the cache entry, such as version information, lifespan, last accessed/used time...etc. Some of these can be provided by the user, e.g. version, lifespan...etc, but some others are computed internally and can only be queried, e.g. last accessed/used time.

The functional map API provides a flexible way to store metadata parameters along with an cache entry. To be able to store a metadata parameter, it must extend `MetaParam.Writable` interface, and implement the methods to allow the internal logic to extra the data. Storing is done via the `set(V, MetaParam.Writable...)` method in the [write-only entry view](#) or [read-write entry view](#) function parameters.

Querying metadata parameters is available via the `findMetaParam(Class)` method available via [read-write entry view](#) or [read-only entry views](#) or function parameters.

Here is an example showing how to store metadata parameters and how to query them:

```

import java.time.Duration;
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.api.functional.MetaParam.*;

WriteOnlyMap<String, String> writeOnlyMap = ...
ReadOnlyMap<String, String> readOnlyMap = ...

CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", "value1",
    (v, view) -> view.set(v, new MetaLifespan(Duration.ofHours(1).toMillis())));
CompletableFuture<MetaLifespan> readFuture = writeFuture.thenCompose(r ->
    readOnlyMap.eval("key1", view -> view.findMetaParam(MetaLifespan.class).get()));
readFuture.thenAccept(System.out::println);

```

If the metadata parameter is generic, for example `MetaEntryVersion<T>`, retrieving the metadata parameter along with a specific type can be tricky if using `.class` static helper in a class because it does not return a `Class<T>` but only `Class`, and hence any generic information in the class is lost:

```

ReadOnlyMap<String, String> readOnlyMap = ...

CompletableFuture<String> readFuture = readOnlyMap.eval("key1", view -> {
    // If caller depends on the typed information, this is not an ideal way to retrieve
    it
    // If the caller does not depend on the specific type, this works just fine.
    Optional<MetaEntryVersion> version = view.findMetaParam(MetaEntryVersion.class);
    return view.get();
});

```

When generic information is important the user can define a static helper method that coerces the static class retrieval to the type requested, and then use that helper method in the call to `findMetaParam`:

```

class MetaEntryVersion<T> implements MetaParam.Writable<EntryVersion<T>> {
    ...
    public static <T> T type() { return (T) MetaEntryVersion.class; }
    ...
}

ReadOnlyMap<String, String> readOnlyMap = ...

CompletableFuture<String> readFuture = readOnlyMap.eval("key1", view -> {
    // The caller wants guarantees that the metadata parameter for version is numeric
    // e.g. to query the actual version information
    Optional<MetaEntryVersion<Long>> version = view.findMetaParam(MetaEntryVersion.
type());
    return view.get();
});

```

Finally, users are free to create new instances of metadata parameters to suit their needs. They are stored and retrieved in the very same way as done for the metadata parameters already provided by the functional map API.

3.6.8. Invocation Parameter

[Per-invocation parameters](#) are applied to regular functional map API calls to alter the behaviour of certain aspects. Adding per invocation parameters is done using the `withParams(Param<?>...)` method.

`Param.FutureMode` tweaks whether a method returning a `CompletableFuture` will span a thread to invoke the method, or instead will use the caller thread. By default, whenever a call is made to a method returning a `CompletableFuture`, a separate thread will be span to execute the method asynchronously. However, if the caller will immediately block waiting for the `CompletableFuture` to complete, spanning a different thread is wasteful, and hence `Param.FutureMode.COMPLETED` can be passed as per-invocation parameter to avoid creating that extra thread. Example:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.api.functional.Param.*;

ReadOnlyMap<String, String> readOnlyMap = ...
ReadOnlyMap<String, String> readOnlyMapCompleted = readOnlyMap.withParams(FutureMode
.COMPLETED);
Optional<String> readFuture = readOnlyMapCompleted.eval("key1", ReadEntryView::find)
.get();
```

`Param.PersistenceMode` controls whether a write operation will be propagated to a persistence store. The default behaviour is for all write-operations to be propagated to the persistence store if the cache is configured with a persistence store. By passing `PersistenceMode.SKIP` as parameter, the write operation skips the persistence store and its effects are only seen in the in-memory contents of the cache. `PersistenceMode.SKIP` can be used to implement an `Cache.evict()` method which removes data from memory but leaves the persistence store untouched:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.api.functional.Param.*;

WriteOnlyMap<String, String> writeOnlyMap = ...
WriteOnlyMap<String, String> skipPersistMap = writeOnlyMap.withParams(PersistenceMode
.SKIP);
CompletableFuture<Void> removeFuture = skipPersistMap.eval("key1", WriteEntryView:
:remove);
```

Note that there's no need for another `PersistenceMode` option to skip reading from the persistence store, because a write operation can skip reading previous value from the store by calling a write-only operation via the `WriteOnlyMap`.

Finally, new Param implementations are normally provided by the functional map API since they tweak how the internal logic works. So, for the most part of users, they should limit themselves to using the Param instances exposed by the API. The exception to this rule would be advanced users who decide to add new interceptors to the internal stack. These users have the ability to query these parameters within the interceptors.

3.6.9. Functional Listeners

The functional map offers a listener API, where clients can register for and get notified when events take place. These notifications are post-event, so that means the events are received after the event has happened.

The listeners that can be registered are split into two categories: [write listeners](#) and [read-write listeners](#).

Write Listeners

[Write listeners](#) enable user to register listeners for any cache entry write events that happen in either a read-write or write-only functional map.

Listeners for write events cannot distinguish between cache entry created and cache entry modify/update events because they don't have access to the previous value. All they know is that a new non-null entry has been written.

However, write event listeners can distinguish between entry removals and cache entry create/modify-update events because they can query what the new entry's value via [ReadEntryView.find\(\)](#) method.

Adding a write listener is done via the [WriteListeners](#) interface which is accessible via both [ReadWriteMap.listeners\(\)](#) and [WriteOnlyMap.listeners\(\)](#) method.

A write listener implementation can be defined either passing a function to [onWrite\(Consumer<ReadEntryView<K, V>>\)](#) method, or passing a [WriteListener](#) implementation to [add\(WriteListener<K, V>\)](#) method. Either way, all these methods return an [AutoCloseable](#) instance that can be used to de-register the function listener:

```

import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.api.functional.Listeners.WriteListeners.WriteListener;

WriteOnlyMap<String, String> woMap = ...

AutoCloseable writeFunctionCloseHandler = woMap.listeners().onWrite(written -> {
    // `written` is a ReadEntryView of the written entry
    System.out.printf("Written: %s%n", written.get());
});
AutoCloseable writeCloseHanlder = woMap.listeners().add(new WriteListener<String,
String>() {
    @Override
    public void onWrite(ReadEntryView<K, V> written) {
        System.out.printf("Written: %s%n", written.get());
    }
});

// Either wrap handler in a try section to have it auto close...
try(writeFunctionCloseHandler) {
    // Write entries using read-write or write-only functional map API
    ...
}
// Or close manually
writeCloseHanlder.close();

```

Read-Write Listeners

[Read-write listeners](#) enable users to register listeners for cache entry created, modified and removed events, and also register listeners for any cache entry write events.

Entry created, modified and removed events can only be fired when these originate on a read-write functional map, since this is the only one that guarantees that the previous value has been read, and hence the differentiation between create, modified and removed can be fully guaranteed.

Adding a read-write listener is done via the [ReadWriteListeners](#) interface which is accessible via [ReadWriteMap.listeners\(\)](#) method.

If interested in only one of the event types, the simplest way to add a listener is to pass a function to either [onCreate](#), [onModify](#) or [onRemove](#) methods. All these methods return an [AutoCloseable](#) instance that can be used to de-register the function listener:

```

import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;

ReadWriteMap<String, String> rwMap = ...
AutoCloseable createClose = rwMap.listeners().onCreate(created -> {
    // `created` is a ReadEntryView of the created entry
    System.out.printf("Created: %s%n", created.get());
});
AutoCloseable modifyClose = rwMap.listeners().onModify((before, after) -> {
    // `before` is a ReadEntryView of the entry before update
    // `after` is a ReadEntryView of the entry after update
    System.out.printf("Before: %s%n", before.get());
    System.out.printf("After: %s%n", after.get());
});
AutoCloseable removeClose = rwMap.listeners().onRemove(removed -> {
    // `removed` is a ReadEntryView of the removed entry
    System.out.printf("Removed: %s%n", removed.get());
});
AutoCloseable writeClose = woMap.listeners().onWrite(written -> {
    // `written` is a ReadEntryView of the written entry
    System.out.printf("Written: %s%n", written.get());
});
...
// Either wrap handler in a try section to have it auto close...
try(createClose) {
    // Create entries using read-write functional map API
    ...
}
// Or close manually
modifyClose.close();

```

If listening for two or more event types, it's better to pass in an implementation of `ReadWriteListener` interface via the `ReadWriteListeners.add()` method. `ReadWriteListener` offers the same `onCreate/onModify/onRemove` callbacks with default method implementations that are empty:

```

import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import
org.infinispan.commons.api.functional.Listeners.ReadWriteListeners.ReadWriteListener;

ReadWriteMap<String, String> rwMap = ...
AutoCloseable readWriteClose = rwMap.listeners.add(new ReadWriteListener<String,
String>() {
    @Override
    public void onCreate(ReadEntryView<String, String> created) {
        System.out.printf("Created: %s%n", created.get());
    }

    @Override
    public void onModify(ReadEntryView<String, String> before, ReadEntryView<String,
String> after) {
        System.out.printf("Before: %s%n", before.get());
        System.out.printf("After: %s%n", after.get());
    }

    @Override
    public void onRemove(ReadEntryView<String, String> removed) {
        System.out.printf("Removed: %s%n", removed.get());
    }
});
AutoCloseable writeClose = rwMap.listeners.add(new WriteListener<String, String>() {
    @Override
    public void onWrite(ReadEntryView<K, V> written) {
        System.out.printf("Written: %s%n", written.get());
    }
});

// Either wrap handler in a try section to have it auto close...
try(readWriteClose) {
    // Create/update/remove entries using read-write functional map API
    ...
}
// Or close manually
writeClose.close();

```

3.6.10. Marshalling of Functions

Running functional map in a cluster of nodes involves marshalling and replication of the operation parameters under certain circumstances.

To be more precise, when write operations are executed in a cluster, regardless of read-write or write-only operations, all the parameters to the method and the functions are replicated to other nodes.

There are multiple ways in which a function can be marshalled. The simplest way, which is also the most costly option in terms of payload size, is to mark the function as `Serializable` :

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;

WriteOnlyMap<String, String> writeOnlyMap = ...

// Force a function to be Serializable
Consumer<WriteEntryView<String>> function =
    (Consumer<WriteEntryView<String>> & Serializable) wv -> wv.set("one");

CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", function);
```

`{brandname}` provides overloads for all functional methods that make lambdas passed directly to the API serializable by default; the compiler automatically selects this overload if that's possible. Therefore you can call

```
WriteOnlyMap<String, String> writeOnlyMap = ...
CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", wv -> wv.set("one"));
```

without doing the cast described above.

A more economical way to marshall a function is to provide an `{brandname} Externalizer` for it:

```

import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.marshall.Externalizer;
import org.infinispan.commons.marshall.SerializeFunctionWith;

WriteOnlyMap<String, String> writeOnlyMap = ...

// Force a function to be Serializable
Consumer<WriteEntryView<String>> function = new SetStringConstant<>();
CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", function);

@SerializeFunctionWith(value = SetStringConstant.Externalizer0.class)
class SetStringConstant implements Consumer<WriteEntryView<String>> {
    @Override
    public void accept(WriteEntryView<String> view) {
        view.set("value1");
    }

    public static final class Externalizer0 implements Externalizer<Object> {
        public void writeObject(ObjectOutput oo, Object o) {
            // No-op
        }
        public Object readObject(ObjectInput input) {
            return new SetStringConstant<>();
        }
    }
}

```

To help users take advantage of the tiny payloads generated by `Externalizer`-based functions, the functional API comes with a helper class called `org.infinispan.commons.marshall.MarshallableFunctions` which provides marshallable functions for some of the most commonly user functions.

In fact, all the functions required to implement `ConcurrentMap` and `JCache` using the functional map API have been defined in `MarshallableFunctions`. For example, here is an implementation of `JCache`'s `boolean putIfAbsent(K, V)` using functional map API which can be run in a cluster:

```

import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.marshall.MarshallableFunctions;

ReadWriteMap<String, String> readWriteMap = ...

CompletableFuture<Boolean> future = readWriteMap.eval("key1",
    MarshallableFunctions.setValueIfAbsentReturnBoolean());
future.thenAccept(stored -> System.out.printf("Value was put? %s\n", stored));

```

3.6.11. Use Cases for Functional API

This new API is meant to complement existing Key/Value {brandname} API offerings, so you'll still be able to use `ConcurrentMap` or `JCache` standard APIs if that's what suits your use case best.

The target audience for this new API is either:

- Distributed or persistent caching/inmemorydatagrid users that want to benefit from `CompletableFuture` and/or `Traversable` for async/lazy data grid or caching data manipulation. The clear advantage here is that threads do not need to be idle waiting for remote operations to complete, but instead these can be notified when remote operations complete and then chain them with other subsequent operations.
- Users who want to go beyond the standard operations exposed by `ConcurrentMap` and `JCache`, for example, if you want to do a replace operation using metadata parameter equality instead of value equality, or if you want to retrieve metadata information from values and so on.

3.7. Encoding

3.7.1. Overview

Encoding is the data conversion operation done by {brandname} caches before storing data, and when reading back from storage.

It allows dealing with a certain data format during API calls (map, listeners, stream, etc) while the format effectively stored is different.

The data conversions are handled by instances of *org.infinispan.commons.dataconversion.Encoder* :

```

public interface Encoder {

    /**
     * Convert data in the read/write format to the storage format.
     *
     * @param content data to be converted, never null.
     * @return Object in the storage format.
     */
    Object toStorage(Object content);

    /**
     * Convert from storage format to the read/write format.
     *
     * @param content data as stored in the cache, never null.
     * @return data in the read/write format
     */
    Object fromStorage(Object content);

    /**
     * Returns the {@link MediaType} produced by this encoder or null if the storage
     format is not known.
     */
    MediaType getStorageFormat();
}

```

3.7.2. Default encoders

{brandname} automatically picks the Encoder depending on the cache configuration. The table below shows which internal Encoder is used for several configurations:

Mode	Configuration	Encoder	Description
Embedded/Server	Default	IdentityEncoder	Passthrough encoder, no conversion done
Embedded	StorageType.OFF_HEAP	GlobalMarshallerEncoder	Use the {brandname} internal marshaller to convert to byte[]. May delegate to the configured marshaller in the cache manager.
Embedded	StorageType.BINARY	BinaryEncoder	Use the {brandname} internal marshaller to convert to byte[], except for primitives and String.
Server	StorageType.OFF_HEAP	IdentityEncoder	Store byte[]s directly as received by remote clients

3.7.3. Overriding programmatically

It is possible to override programmatically the encoding used for both keys and values, by calling the *.withEncoding()* method variants from *AdvancedCache*.

Example, consider the following cache configured as OFF_HEAP:

```
// Read and write POJO, storage will be byte[] since for
// OFF_HEAP the GlobalMarshallerEncoder is used internally:
cache.put(1, new Pojo())
Pojo value = cache.get(1)

// Get the content in its stored format by overriding
// the internal encoder with a no-op encoder (IdentityEncoder)
Cache<?,?> rawContent = cache.getAdvancedCache().withValueEncoding(IdentityEncoder
.class)
byte[] marshalled = rawContent.get(1)
```

The override can be useful if any operation in the cache does not require decoding, such as counting number of entries, or calculating the size of byte[] of an OFF_HEAP cache.

3.7.4. Defining custom Encoders

A custom encoder can be registered in the *EncoderRegistry*.



Ensure that the registration is done in every node of the cluster, before starting the caches.

Consider a custom encoder used to compress/decompress with gzip:

```
public class GzipEncoder implements Encoder {

    @Override
    public Object toStorage(Object content) {
        assert content instanceof String;
        return compress(content.toString());
    }

    @Override
    public Object fromStorage(Object content) {
        assert content instanceof byte[];
        return decompress((byte[]) content);
    }

    private byte[] compress(String str) {
        try (ByteArrayOutputStream baos = new ByteArrayOutputStream();
             GZIPOutputStream gis = new GZIPOutputStream(baos)) {
            gis.write(str.getBytes("UTF-8"));
            gis.close();
        }
    }
}
```

```

        return baos.toByteArray();
    } catch (IOException e) {
        throw new RuntimeException("Unable to compress", e);
    }
}

private String decompress(byte[] compressed) {
    try (GZIPInputStream gis = new GZIPInputStream(new ByteArrayInputStream
(compressed)));
        BufferedReader bf = new BufferedReader(new InputStreamReader(gis, "UTF-8")
)) {
        StringBuilder result = new StringBuilder();
        String line;
        while ((line = bf.readLine()) != null) {
            result.append(line);
        }
        return result.toString();
    } catch (IOException e) {
        throw new RuntimeException("Unable to decompress", e);
    }
}

@Override
public MediaType getStorageFormat() {
    return MediaType.parse("application/gzip");
}

@Override
public boolean isStorageFormatFilterable() {
    return false;
}

@Override
public short id() {
    return 10000;
}
}

```

It can be registered by:

```

GlobalComponentRegistry registry = cacheManager.getGlobalComponentRegistry();
EncoderRegistry encoderRegistry = registry.getComponent(EncoderRegistry.class);
encoderRegistry.registerEncoder(new GzipEncoder());

```

And then be used to write and read data from a cache:

```

AdvancedCache<String, String> cache = ...

// Decorate cache with the newly registered encoder, without encoding keys
// (IdentityEncoder)
// but compressing values
AdvancedCache<String, String> compressingCache = (AdvancedCache<String, String>)
cache.withEncoding(IdentityEncoder.class, GzipEncoder.class);

// All values will be stored compressed...
compressingCache.put("297931749", "0412c789a37f5086f743255cfa693dd5");

// ... but API calls deals with String
String value = compressingCache.get("297931749");

// Bypassing the value encoder to obtain the value as it is stored
Object value = compressingCache.withEncoding(IdentityEncoder.class).get("297931749");

// value is a byte[] which is the compressed value

```

3.7.5. MediaType

A Cache can optionally be configured with a `org.infinispan.commons.dataconversion.MediaType` for keys and values. By describing the data format of the cache, {brandname} is able to convert data on the fly during cache operations.



The MediaType configuration is more suitable when storing binary data. When using server mode, it's common to have a MediaType configured and clients such as REST or Hot Rod reading and writing in different formats.

The data conversion between MediaType formats are handled by instances of `org.infinispan.commons.dataconversion.Transcoder`

```

public interface Transcoder {

    /**
     * Transcodes content between two different {@link MediaType}.
     *
     * @param content      Content to transcode.
     * @param contentType  The {@link MediaType} of the content.
     * @param destinationType The target {@link MediaType} to convert.
     * @return the transcoded content.
     */
    Object transcode(Object content, MediaType contentType, MediaType destinationType);

    /**
     * @return all the {@link MediaType} handled by this Transcoder.
     */
    Set<MediaType> getSupportedMediaTypes();
}

```

Configuration

Declarative:

```

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

```

Programmatic:

```

ConfigurationBuilder cfg = new ConfigurationBuilder();

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

```

Overriding the MediaType Programmatically

It's possible to decorate the Cache with a different MediaType, allowing cache operations to be executed sending and receiving different data formats.

Example:


```
DefaultCacheManager cacheManager = new DefaultCacheManager();

// The cache will store POJO for keys and values
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.encoding().key().mediaType("application/x-java-object");
cfg.encoding().value().mediaType("application/x-java-object");

cacheManager.defineConfiguration("mycache", cfg.build());

Cache<Integer, Person> cache = cacheManager.getCache("mycache");

cache.put(1, new Person("John", "Doe"));

// Wraps cache using 'application/x-java-object' for keys but JSON for values
Cache<Integer, byte[]> jsonValuesCache = (Cache<Integer, byte[]>) cache
    .getAdvancedCache().withMediaType("application/x-java-object", "application/json");

byte[] json = jsonValuesCache.get(1);
```

Will return the value in JSON format:

```
{
  "_type": "org.infinispan.sample.Person",
  "name": "John",
  "surname": "Doe"
}
```



Most Transcoders are installed when server mode is used; when using library mode, an extra dependency, *org.infinispan:infinispan-server-core* should be added to the project.

Transcoders and Encoders

Usually there will be none or only one data conversion involved in a cache operation:

- No conversion by default on caches using in embedded or server mode;
- *Encoder* based conversion for embedded caches without MediaType configured, but using OFF_HEAP or BINARY;
- *Transcoder* based conversion for caches used in server mode with multiple REST and Hot Rod clients sending and receiving data in different formats. Those caches will have MediaType configured describing the storage.

But it's possible to have both encoders and transcoders being used simultaneously for advanced use cases.

Consider an example, a cache that stores marshalled objects (with jboss marshaller) content but for security reasons a transparent encryption layer should be added in order to avoid storing "plain"

data to an external store. Clients should be able to read and write data in multiple formats.

This can be achieved by configuring the cache with the the `MediaType` that describes the storage regardless of the encoding layer:

```
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.encoding().key().mediaType("application/x-jboss-marshalling");
cfg.encoding().key().mediaType("application/x-jboss-marshalling");
```

The transparent encryption can be added by decorating the cache with a special *Encoder* that encrypts/decrypts with storing/retrieving, for example:

```
public class Scrambler implements Encoder {

    Object toStorage(Object content) {
        // Encrypt data
    }

    Object fromStorage(Object content) {
        // Decrypt data
    }

    MediaType getStorageFormat() {
        return "application/scrambled";
    }

}
```

To make sure all data written to the cache will be stored encrypted, it's necessary to decorate the cache with the Encoder above and perform all cache operations in this decorated cache:

```
Cache<?,?> secureStorageCache = cache.getAdvancedCache().withEncoding(Scrambler.class)
    .put(k,v);
```

The capability of reading data in multiple formats can be added by decorating the cache with the desired `MediaType`:

```
// Obtain a stream of values in XML format from the secure cache
secureStorageCache.getAdvancedCache().withMediaType("application/xml","application/xml")
    .values().stream();
```

Internally, {brandname} will first apply the encoder *fromStorage* operation to obtain the entries, that will be in "application/x-jboss-marshalling" format and then apply a successive conversion to "application/xml" by using the adequate Transcoder.

Chapter 4. Eviction and Data Container

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

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



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



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

4.1. Enabling Eviction

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

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

4.1.1. Eviction strategy

Strategies control how the eviction is handled.

The possible choices are

NONE

Eviction is not enabled and it is assumed that the user will not invoke `evict` directly on the cache. If passivation is enabled this will cause a warning message to be emitted. This is the default strategy.

MANUAL

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

REMOVE

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

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

EXCEPTION

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

4.1.2. Eviction types

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

COUNT

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

MEMORY

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

4.1.3. Storage type

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

OBJECT

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

BINARY

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

OFF-HEAP

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



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

4.1.4. More defaults

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

In case there is an memory element, this table describes the behaviour of eviction based on information provided in the xml configuration ("-" in Supplied size or Supplied strategy column

means that the attribute wasn't supplied)

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

4.2. Expiration

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



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

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

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

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



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

4.2.1. Difference between Eviction and Expiration

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

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

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

lifespan

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

maximum idle time

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

4.3. Expiration details

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

4.3.1. Maximum Idle Expiration

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

Local Max Idle

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

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

Clustered Max Idle

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

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

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

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



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

4.3.2. Configuration

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

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

Programmatically, the same would be defined using:

```
Configuration c = new ConfigurationBuilder()
    .memory().size(2000)
    .expiration().wakeUpInterval(50001).lifespan(10001).maxIdle(5001)
    .build();
```

4.3.3. Memory Based Eviction Configuration

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

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

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

4.3.4. Default values

Eviction is disabled by default. Default values are used:

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

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

4.3.5. Using expiration

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

For example, assume the following configuration:

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



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

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

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

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

4.4. Expiration designs

Central to expiration is an `ExpirationManager`.

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

The expiration manager processes expirations in the following manner:

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

Chapter 5. Persistence

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

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

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

These SPIs allow for the following features:

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

5.1. Configuration

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



Implementing both a `CacheWriter` and `CacheLoader`

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

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

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

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

    </store>
  </persistence>
</local-cache>
```

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

connection-attempts

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

connection-interval

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

availability-interval

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

passivation

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

This property has a significant impact on {brandname} interactions with the loaders. See [Cache Passivation](#) for more information.

class

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

fetch-state

Fetches the persistent state of a cache when joining a cluster. The default value is `false` (boolean).

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

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

preload

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

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

Pre-loading data is done locally, so any data loaded is stored locally in the node only. The pre-loaded data is not replicated or distributed. Likewise, {brandname} pre-loads data only up to the maximum configured number of entries in [eviction](#).

shared

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

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

segmented

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

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

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

read-only

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

purge

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

max-batch-size

Sets the maximum size of a batch to insert or delete from the cache store. The default value is #100.

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

write-behind

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

`singleton`

Enables one node in the cluster, the coordinator, to store modifications. The default value is `false` (boolean).

Whenever data enters a node, it is replicated or distributed to keep the in-memory state of the caches synchronized. The coordinator is responsible for pushing that state to disk.

If `true` you must set this property on all nodes in the cluster. Only the coordinator of the cluster persists data. However, all nodes must have this property enabled to prevent other nodes from persisting data.

You cannot enable the `singleton` property if the cache store is shared.



You can define additional attributes in the `properties` section to configure specific aspects of each cache loader, such as the `myProp` attribute in the previous example.

Other cache loaders with more complex configurations also include additional properties. See the following JDBC cache store configuration for examples.

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

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

The same configuration can be achieved programmatically:

```

ConfigurationBuilder builder = new ConfigurationBuilder();
builder.persistence()
    .passivation(false)
    .addSingleFileStore()
    .preload(true)
    .shared(false)
    .fetchPersistentState(true)
    .ignoreModifications(false)
    .purgeOnStartup(false)
    .location(System.getProperty("java.io.tmpdir"))
    .async()
    .enabled(true)
    .threadPoolSize(5)
    .singleton()
    .enabled(true)
    .pushStateWhenCoordinator(true)
    .pushStateTimeout(20000);

```

5.2. Cache Passivation

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

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

5.2.1. Limitations

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

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

5.2.2. Cache Loader Behavior with Passivation Disabled vs Enabled

When passivation is disabled, whenever an element is modified, added or removed, then that

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

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

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

5.3. Cache Loaders and transactional caches

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

5.4. Write-Through And Write-Behind Caching

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

- Write-Through (Synchronous)

- Write-Behind (Asynchronous)

5.4.1. Write-Through (Synchronous)

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

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

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

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

5.4.2. Write-Behind (Asynchronous)

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

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

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

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

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


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

5.4.3. Segmented Stores

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

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

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



Loss of data can occur if the number of segments in a cache store are not changed gracefully. For this reason, if you change the `numSegments` setting in the store configuration, you must migrate the existing store to use the new configuration.

The recommended method to migrate the cache store configuration is to perform a rolling upgrade. The store migrator supports migrating a non-segmented cache store to a segmented cache store only. The store migrator does not currently support migrating from a segmented cache store.



Not all cache stores support segmentation. See the appropriate section for each store to determine if it supports segmentation.

If you plan to convert or write a new store to support segmentation, see the following SPI section that provides more details.

5.5. Filesystem based cache stores

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



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

5.5.1. Single File Store

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

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

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

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

Segmentation support

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

Configuration

The following examples show single file cache store configuration:

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

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

5.5.2. Soft-Index File Store

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

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

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

Segmentation support

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

Configuration

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

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

Programmatic configuration would look as follows:

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

Current limitations

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

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

5.6. JDBC String based Cache Store

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

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



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



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

5.6.1. Connection management (pooling)

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

{brandname} ships with three `ConnectionFactory` implementations:

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

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

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

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

- `ManagedConnectionFactory` is a connection factory that can be used within managed environments, such as application servers. It knows how to look into the JNDI tree at a certain location (configurable) and delegate connection management to the `DataSource`. Refer to javadoc [javadoc](#) for details on how this can be configured.
- `SimpleConnectionFactory` is a factory implementation that will create database connection on a per invocation basis. Not recommended in production.

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

5.6.2. Sample configurations

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

```

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

```

```

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

```

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

```

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

```

```

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

```



Apache Derby users

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

5.7. Remote store

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

5.7.1. Segmentation support

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

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



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

5.7.2. Sample Usage

```

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

```

```

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

```

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

5.8. Cluster cache loader

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

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

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



The cluster cache loader does not support segmentation.


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

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

For a list of ClusterCacheLoader configuration refer to the [javadoc](#).



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

5.9. Command-Line Interface cache loader

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

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



Details on the format of the URL and how to make sure a node can receive invocations via the CLI can be found in the [Command-Line Interface chapter](#).



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

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

```
ConfigurationBuilder b = new ConfigurationBuilder();
b.persistence()
  .addStore(CLInterfaceLoaderConfigurationBuilder.class)
  .connectionString("jmx://1.2.3.4:4444/MyCacheManager/myCache");
```

5.10. RocksDB Cache Store

{brandname} supports using RocksDB as a cache store.

5.10.1. Introduction

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

5.10.2. Segmentation support

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



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

Sample Usage

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

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

5.10.3. Configuration

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

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

Sample Programatic Configuration

```

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

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

```

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

Sample XML Configuration

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

5.10.4. Additional References

Refer to the [test case](#) for code samples in action.

Refer to [test configurations](#) for configuration samples.

5.11. JPA Cache Store

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

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

Things to note

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



The JPA cache store does not support segmentation.

5.11.1. Sample Usage

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

persistence.xml

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

User entity class

```

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

    ...
}

```

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

```

EmbeddedCacheManager cacheManager = ...;

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

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

```

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

```

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

```

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

```

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

```

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

```

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

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

```

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

5.11.2. Configuration

Sample Programatic Configuration

```

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

```

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

Sample XML Configuration

```

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

```

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

5.11.3. Additional References

Refer to the [test case](#) for code samples in action.

Refer to [test configurations](#) for configuration samples.

5.12. Custom Cache Stores

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

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

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

4. Add your custom store to your cache's configuration:

1. Add your custom store to the ConfigurationBuilder, for example:

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

2. Define your custom store via xml:

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

5.12.1. HotRod Deployment

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

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

5.13. Store Migrator

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

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

5.13.1. Migrating Cache Stores

To perform a migration with `StoreMigrator`,

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

You can find an example properties file that contains all applicable configuration options in [migrator.properties](#).

3. Specify `.properties` file as an argument for `StoreMigrator`.
4. Run `mvn exec:java` to execute the migrator.

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

```

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

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

  <dependencies>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-tools</artifactId>
      <version>${version.infinispan}</version>
    </dependency>

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

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

```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

5.13.2. Store Migrator Properties

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

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

Common Properties

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

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

JDBC Properties

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

Property	Description	Example value	Required
marshaller.type	<p>The marshaller to use for the store. Possible values are:</p> <ul style="list-style-type: none"> - LEGACY {brandname} 8.2.x marshaller. Valid for source stores only. - CURRENT {brandname} 9.x marshaller. - CUSTOM Custom marshaller. 	CURRENT	TRUE
marshaller.class	The class of the marshaller if type=CUSTOM	org.example.CustomMarshaller	
marshaller.externalizers	A comma-separated list of custom AdvancedExternalizer implementations to load [id]:<Externalizer class>	25:Externalizer1,org.example.Externalizer2	
connection_pool.connection_url	The JDBC connection url	jdbc:postgresql:postgres	TRUE
connection_pool.driver_class	The class of the JDBC driver	org.postgresql.Driver	TRUE
connection_pool.username	Database username		TRUE
connection_pool.password	Database password		TRUE
db.major_version	Database major version	9	
db.minor_version	Database minor version	5	
db.disable_upsert	Disable db upsert	false	
db.disable_indexing	Prevent table index being created	false	
table.<binary string>.table_name_prefix	Additional prefix for table name	tablePrefix	
table.<binary string>.<id data timestamp>.name	Name of the column	id_column	TRUE
table.<binary string>.<id data timestamp>.type	Type of the column	VARCHAR	TRUE

Property	Description	Example value	Required
key_to_string_mapper	TwoWayKey2StringMapper Class	<code>org.infinispan.persistence.keymappers.DefaultTwoWayKey2StringMapper</code>	

LevelDB/RocksDB Properties

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

SingleFileStore Properties

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

SoftIndexFileStore Properties

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

Following sections describe the SPI and also discuss the SPI implementations that {brandname} ships out of the box.

5.14. SPI

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

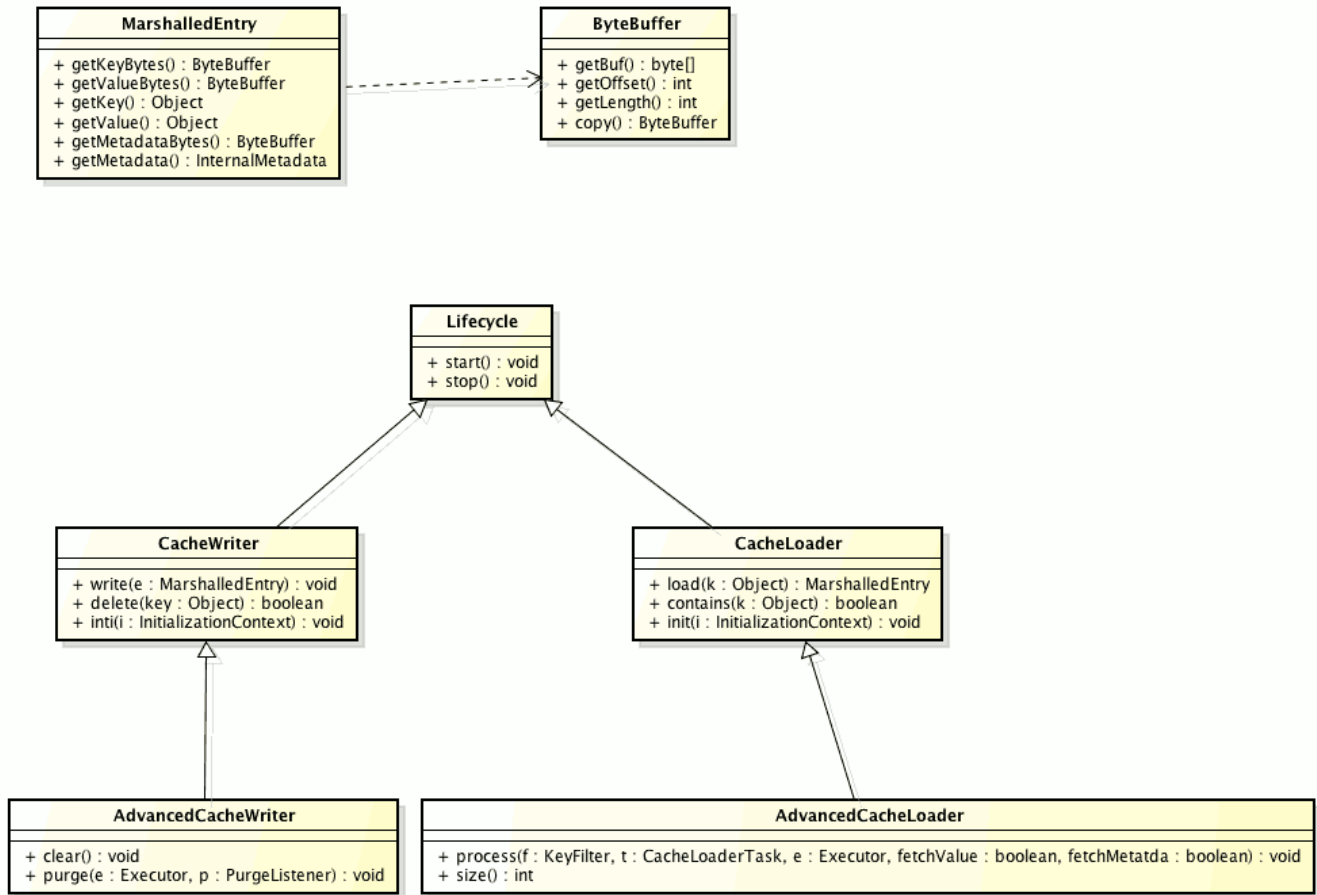


Figure 1. Persistence SPI

Some notes about the classes:

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

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

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

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

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

If you're looking at migrating your existing store to the new API or to write a new store implementation, the [SingleFileStore](#) might be a good starting point/example.

5.15. More implementations

Many more cache loader and cache store implementations exist. Visit [this website](#) for more details.

Chapter 6. Clustering

A cache manager can be configured to be either local (standalone) or clustered. When clustered, manager instances use JGroups' discovery protocols to automatically discover neighboring instances on the same local network and form a cluster.

Creating a local-only cache manager is trivial: just use the no-argument `DefaultCacheManager` constructor, or supply the following XML configuration file.

```
<infinispan/>
```

To start a clustered cache manager, you need to create a clustered configuration.

```
GlobalConfigurationBuilder gcb = GlobalConfigurationBuilder.defaultClusteredBuilder();
DefaultCacheManager manager = new DefaultCacheManager(gcb.build());
```

```
<infinispan>
  <cache-container>
    <transport/>
  </cache-container>
</infinispan>
```

Individual caches can then be configured in different modes:

- **Local:** changes and reads are never replicated. This is the only mode available in non-clustered cache managers.
- **Invalidation:** changes are not replicated, instead the key is invalidated on all nodes; reads are local.
- **Replicated:** changes are replicated to all nodes, reads are always local.
- **Distributed:** changes are replicated to a fixed number of nodes, reads request the value from at least one of the owner nodes.

6.1. Which cache mode should I use?

Which cache you should use depends on the qualities/guarantees you need for your data. The following table summarizes the most important ones:

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

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

6.2. Local Mode

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

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

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

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

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

6.2.1. Simple Cache

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

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

- transactions and invocation batching

- persistence (cache stores and loaders)
- custom interceptors (there's no interceptor stack!)
- indexing
- compatibility (embedded/server mode)
- store as binary (which is hardly useful for local caches)

From the API perspective these features throw an exception:

- adding custom interceptors
- Distributed Executors Framework

So, what's left?

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

Declarative configuration

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

Programmatic configuration

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

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

6.3. Invalidation Mode

In invalidation, the caches on different nodes do not actually share any data. Instead, when a key is written to, the cache only aims to remove data that may be stale from other nodes. This cache mode only makes sense if you have another, permanent store for your data such as a database and are

only using {brandname} as an optimization in a read-heavy system, to prevent hitting the database for every read. If a cache is configured for invalidation, every time data is changed in a cache, other caches in the cluster receive a message informing them that their data is now stale and should be removed from memory and from any local store.

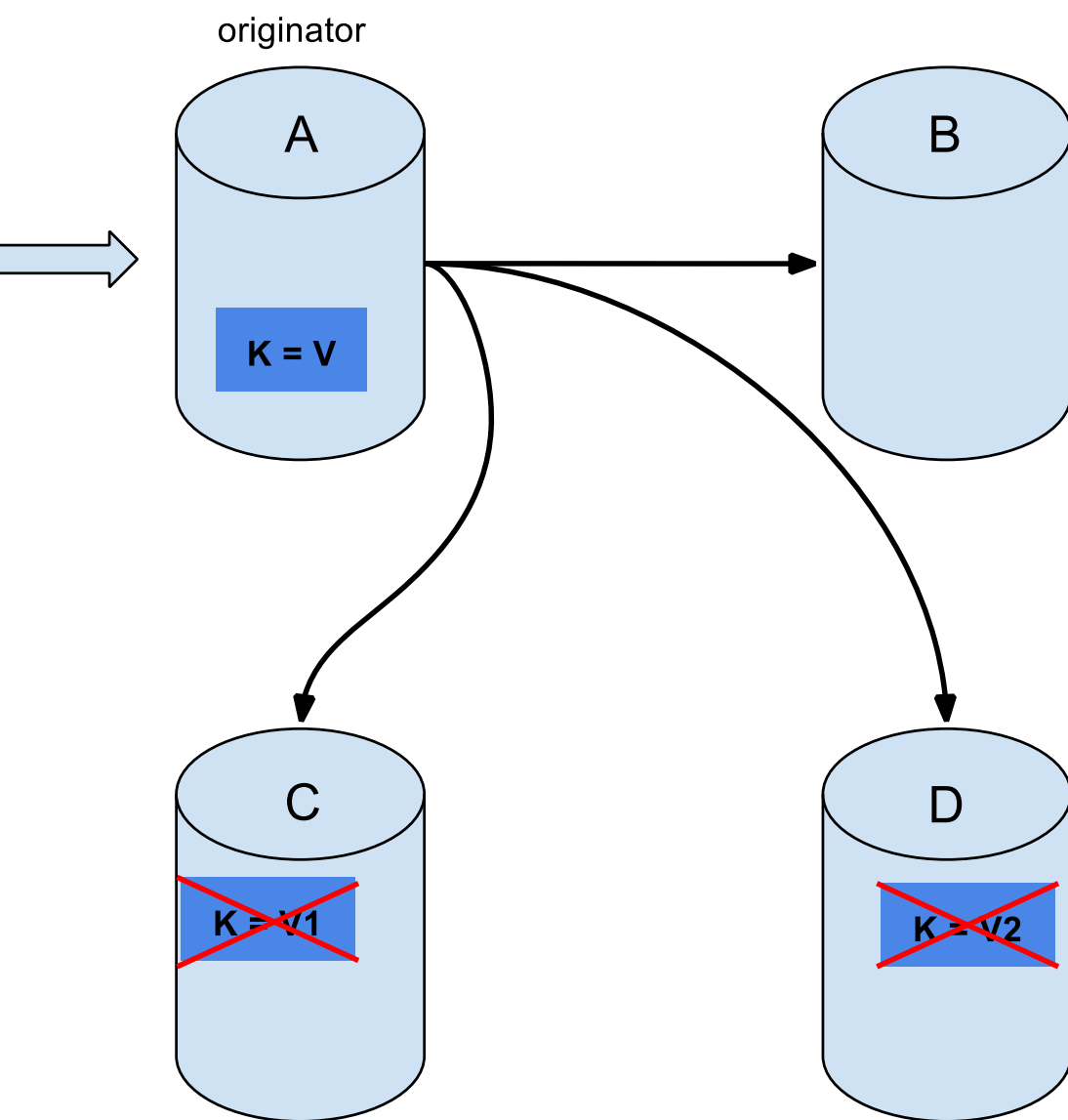


Figure 2. Invalidation mode

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

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



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

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

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

Transactions can be used to batch the invalidation messages. They won't behave like regular transactions though, as locks are only acquired on the local node, and entries can be invalidated by other transactions at any time.

6.4. Replicated Mode

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

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

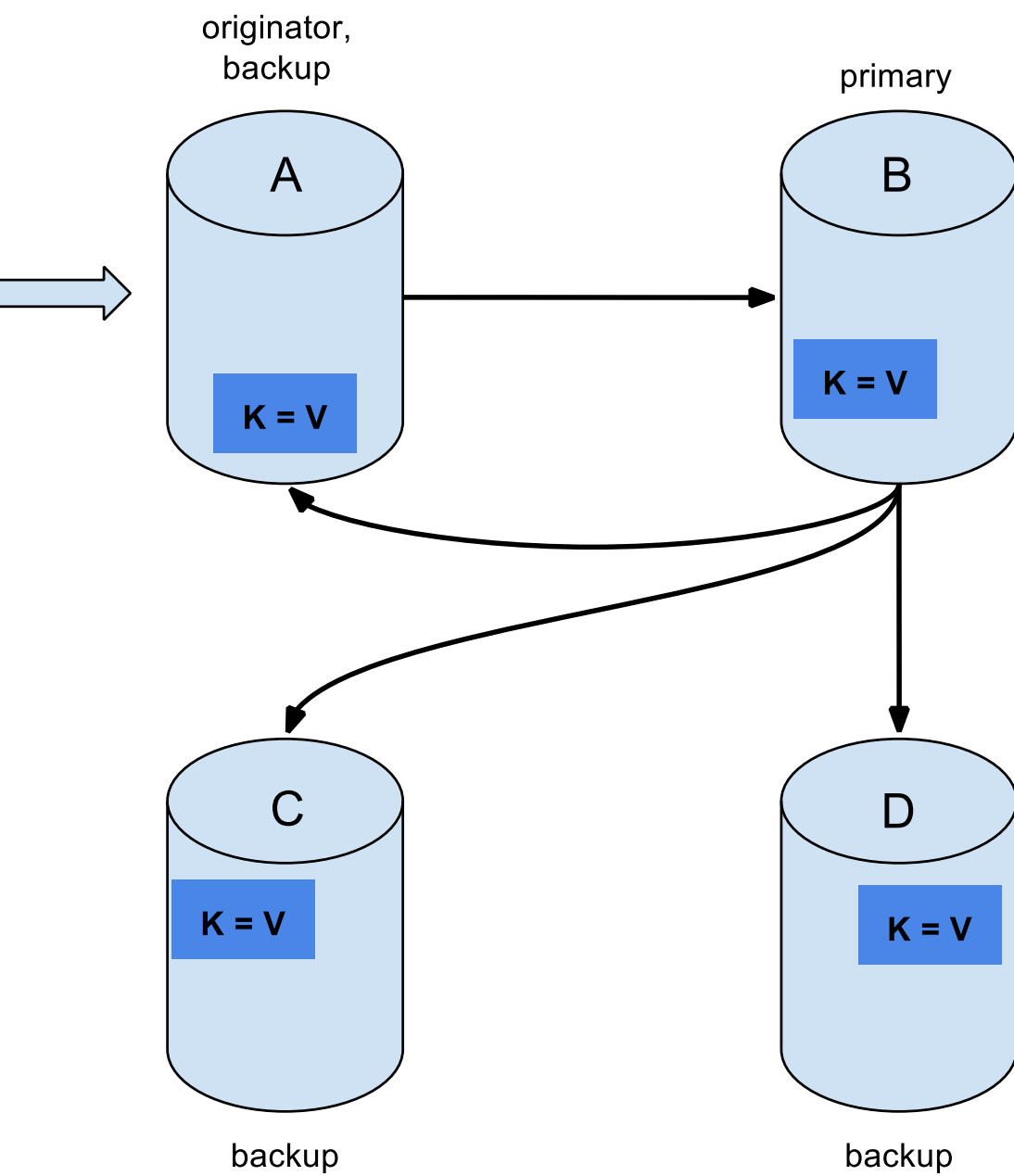


Figure 3. Replicated mode

Replicated mode can be synchronous or asynchronous.

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

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

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

6.5. Distribution Mode

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

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

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

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

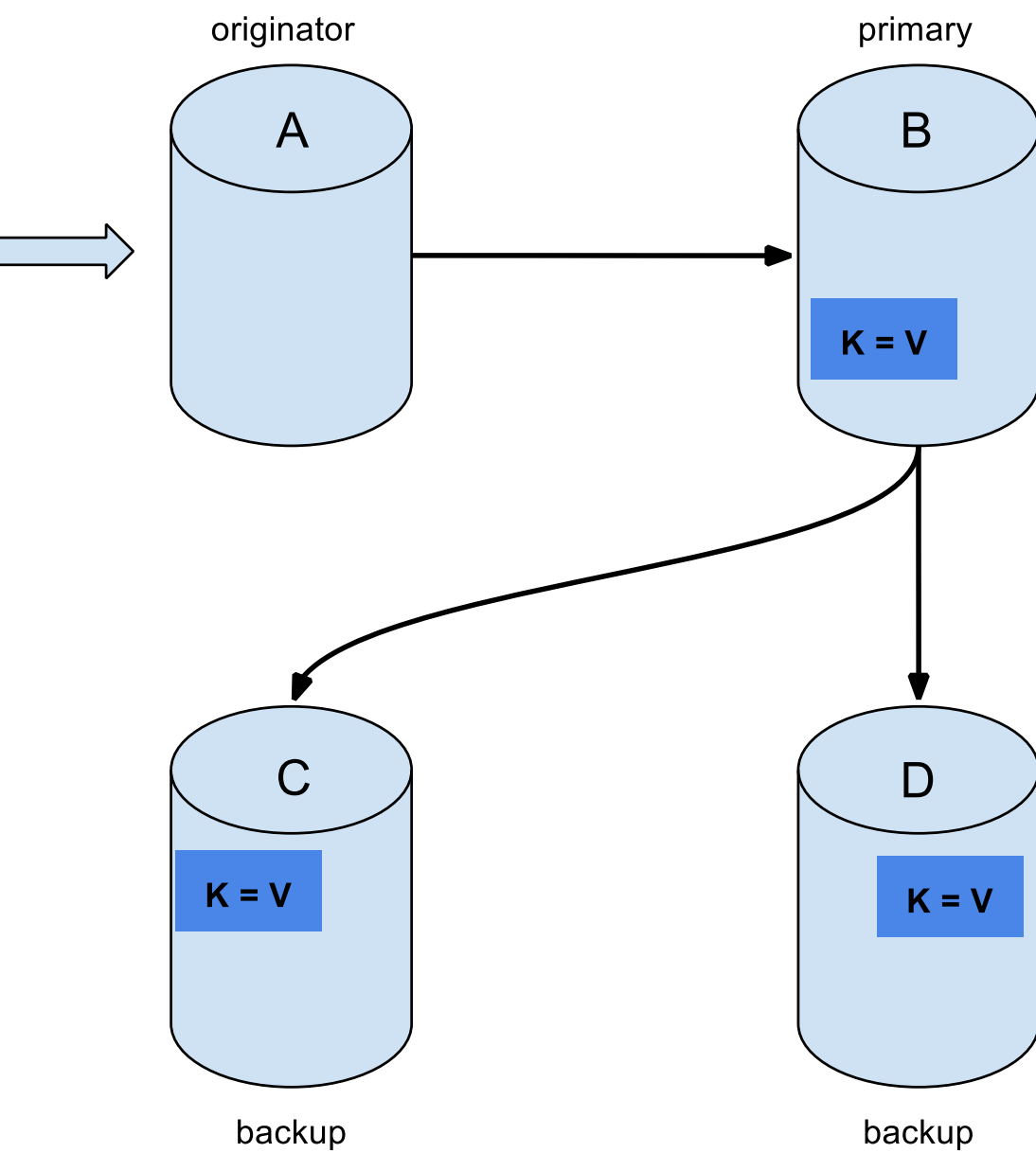


Figure 4. Distributed mode

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

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



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

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

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

6.5.1. Read consistency

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

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

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

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

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

6.5.2. Key Ownership

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

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

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

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

You can customize the key-to-segment mapping by configuring a [KeyPartitioner](#) or by using the [Grouping API](#).

However, {brandname} provides the following implementations:

SyncConsistentHashFactory

Uses an algorithm based on [consistent hashing](#). Selected by default when server hinting is disabled.

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

TopologyAwareSyncConsistentHashFactory

Similar to `SyncConsistentHashFactory`, but adapted for [Server Hinting](#). Selected by default when server hinting is enabled.

DefaultConsistentHashFactory

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

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

TopologyAwareConsistentHashFactory

Similar to `DefaultConsistentHashFactory`, but adapted for [Server Hinting](#).

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

ReplicatedConsistentHashFactory

Used internally to implement replicated caches. You should never explicitly select this algorithm

in a distributed cache.

Capacity Factors

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

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

Zero Capacity Node

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

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

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

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

Hashing Configuration

This is how you configure hashing declaratively, via XML:

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

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

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

6.5.3. Initial cluster size

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

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

The programmatic Java configuration:

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

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

6.5.4. L1 Caching

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

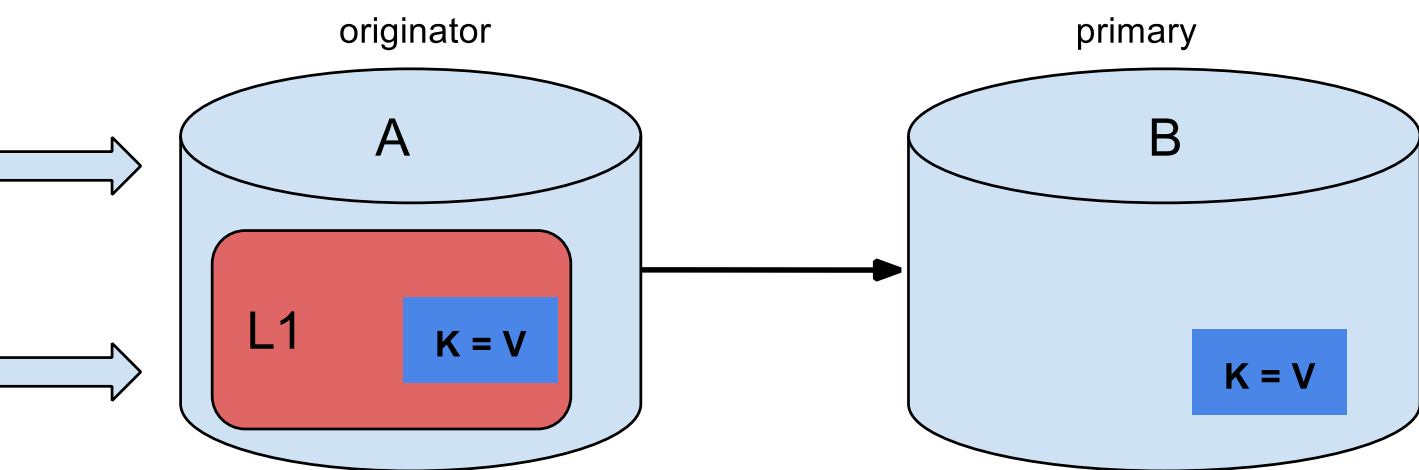


Figure 5. L1 caching

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

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

6.5.5. Server Hinting

The following topology hints can be specified:

Machine

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

Rack

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

Site

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

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

Configuration

The hints are configured at transport level:

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

6.5.6. 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, we can generate a sequence of (pseudo-)random keys, see what their primary owner is, and hand them out to the application when it needs a key mapping to a particular node.

API

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

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

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

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

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

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

Lifecycle

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

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

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

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

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

Topology changes

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

As such, applications should treat **KeyAffinityService** purely as an optimization, and they should

not rely on the location of a generated key for correctness.

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

6.5.7. The Grouping API

Complementary to [Key affinity service](#) and similar to [AtomicMap](#), the grouping API allows you to co-locate a group of entries on the same nodes, but without being able to select the actual nodes.

How does it work?

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

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

How do I use the grouping API?

First, you must enable groups. If you are configuring `{brandname}` programmatically, then call:

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

Or, if you are using XML:

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

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


```

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

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

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

```



The group method must return a **String**

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

```

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

    Class<T> getKeyType();
}

```

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

```

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 {brandname} 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.acme.KXGrouper" />
  </groups>
</distributed-cache>

```

Advanced Interface

AdvancedCache has two group-specific methods:

getGroup(groupName)

Retrieves all keys in the cache that belong to a group.

removeGroup(groupName)

Removes all the keys in the cache that belong to a group.

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

6.6. Scattered Mode

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

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

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

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

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

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

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

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

And this is how you can configure it programmatically:

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

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

6.7. Asynchronous Options

6.7.1. Asynchronous Communications

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

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

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

6.7.2. Asynchronous API

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

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



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

6.7.3. Return Values

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 {brandname} 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. For more information, see [Write Skews](#).

6.8. Partition handling

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

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

Split brain

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

`numOwners` nodes crash in sequence

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

The partition handling functionality discussed in this section allows the user to configure what operations can be performed on a cache in the event of a split brain occurring. {brandname}

provides multiple partition handling strategies, which in terms of Brewer’s [CAP theorem](#) determine whether availability or consistency is sacrificed in the presence of partition(s). Below is a list of the provided strategies:

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

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

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

6.8.1. Split brain

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

Split Strategies

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

ALLOW_READ_WRITES

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

DENY_READ_WRITES

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

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

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

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

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

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

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

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

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

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

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

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

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

ALLOW_READS

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

Current limitations

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

6.8.2. Successive nodes stopped

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

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

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

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

can force the availability mode back to AVAILABLE via JMX.

6.8.3. Conflict Manager

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

Detecting Conflicts

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

Merge Policies

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

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



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

Currently {brandname} provides the following implementations of `EntryMergePolicy`:

Policy	Description
MergePolicy.NONE (default)	No attempt is made to resolve conflicts. Entries hosted on the minority partition are removed and the nodes in this partition do not hold any data until the rebalance starts. Note, this behaviour is equivalent to prior Infinispan versions which did not support conflict resolution. Note, in this case all changes made to entries hosted on the minority partition are lost, but once the rebalance has finished all entries will be consistent.
MergePolicy.PREFERRED_ALWAYS	Always utilise the "preferredEntry". MergePolicy.NONE is almost equivalent to PREFERRED_ALWAYS, albeit without the performance impact of performing conflict resolution, therefore MergePolicy.NONE should be chosen unless the following scenario is a concern. When utilising the DENY_READ_WRITES or DENY_READ strategy, it is possible for a write operation to only partially complete when the partitions enter DEGRADED mode, resulting in replicas containing inconsistent values. MergePolicy.PREFERRED_ALWAYS will detect said inconsistency and resolve it, whereas with MergePolicy.NONE the CacheEntry replicas will remain inconsistent after the cluster has rebalanced.
MergePolicy.PREFERRED_NON_NULL	Utilise the "preferredEntry" if it is non-null, otherwise utilise the first entry from "otherEntries".
MergePolicy.REMOVE_ALL	Always remove a key from the cache when a conflict is detected.
Fully qualified class name	The custom implementation for merge will be used Custom merge policy

6.8.4. Usage

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

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

```

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

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

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

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

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

```



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

6.8.5. Configuring partition handling

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

```

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

```

The same can be achieved programmatically:

```

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

```

Implement a custom merge policy

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

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

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

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

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

        return the_solved_CacheEntry;
    }
}
```

Deploy custom merge policies to a Infinispan server instance

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

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

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

6.8.6. Monitoring and administration

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

The availability mode is also accessible via the [AdvancedCache](#) interface:

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

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

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

Chapter 7. Marshalling

Marshalling is the process of converting Java POJOs into something that can be written in a format that can be transferred over the wire. Unmarshalling is the reverse process whereby data read from a wire format is transformed back into Java POJOs. {brandname} uses marshalling/unmarshalling in order to:

- Transform data so that it can be send over to other {brandname} nodes in a cluster.
- Transform data so that it can be stored in underlying cache stores.
- Store data in {brandname} in a wire format to provide lazy deserialization capabilities.

7.1. The Role Of JBoss Marshalling

Since performance is a very important factor in this process, {brandname} uses JBoss Marshalling framework instead of standard Java Serialization in order to marshall/unmarshall Java POJOs. Amongst other things, this framework enables {brandname} to provide highly efficient ways to marshall internal {brandname} Java POJOs that are constantly used. Apart from providing more efficient ways to marshall Java POJOs, including internal Java classes, JBoss Marshalling uses highly performant `java.io.ObjectOutput` and `java.io.ObjectInput` implementations compared to standard `java.io.ObjectOutputStream` and `java.io.ObjectInputStream`.

7.2. Support For Non-Serializable Objects

From a users perspective, a very common concern is whether {brandname} supports storing non-Serializable objects. In 4.0, an {brandname} cache instance can only store non-Serializable key or value objects if, and only if:

- cache is configured to be a local cache *and...*
- cache is not configured with lazy serialization *and...*
- cache is not configured with any write-behind cache store

If either of these options is true, key/value pairs in the cache will need to be marshalled and currently they require to either to extend `java.io.Serializable` or `java.io.Externalizable`.



Since {brandname} 5.0, marshalling non-Serializable key/value objects are supported as long as users can to provide meaningful Externalizer implementations for these non-Seralizable objects.

If you're unable to retrofit Serializable or Externalizable into the classes whose instances are stored in {brandname}, you could alternatively use something like `XStream` to convert your Non-Serializable objects into a String that can be stored into {brandname}. The one caveat about using XStream is that it slows down the process of storing key/value objects due to the XML transformation that it needs to do.

7.2.1. Store As Binary

Store as binary enables data to be stored in its serialized form. This can be useful to achieve lazy deserialization, which is the mechanism by which {brandname} by which serialization and deserialization of objects is deferred till the point in time in which they are used and needed. This typically means that any deserialization happens using the thread context class loader of the invocation that requires deserialization, and is an effective mechanism to provide classloader isolation. By default lazy deserialization is disabled but if you want to enable it, you can do it like this:

- Via XML at the Cache level, either under `<*-cache />` element:

```
<memory>
  <binary />
</memory>
```

- Programmatically:

```
ConfigurationBuilder builder = ...
builder.memory().storageType(StorageType.BINARY);
```

Equality Considerations

When using lazy deserialization/storing as binary, keys and values are wrapped as [WrappedBytes](#). It is this wrapper class that transparently takes care of serialization and deserialization on demand, and internally may have a reference to the object itself being wrapped, or the serialized, byte array representation of this object.

This has a particular effect on the behavior of equality. The `equals()` method of this class will either compare binary representations (byte arrays) or delegate to the wrapped object instance's `equals()` method, depending on whether both instances being compared are in serialized or deserialized form at the time of comparison. If one of the instances being compared is in one form and the other in another form, then one instance is either serialized or deserialized.

This will affect the way keys stored in the cache will work, when `storeAsBinary` is used, since comparisons happen on the key which will be wrapped by a `MarshaledValue`. Implementers of `equals()` methods on their keys need to be aware of the behavior of equality comparison, when a key is wrapped as a `MarshaledValue`, as detailed above.

Store-by-value via defensive copying

The configuration `storeAsBinary` offers the possibility to enable defensive copying, which allows for store-by-value like behaviour.

{brandname} marshalls objects the moment they're stored, hence changes made to object references are not stored in the cache, not even for local caches. This provides store-by-value like behaviour. Enabling `storeAsBinary` can be achieved:

- Via XML at the Cache level, either under `<*-cache />` or `<default />` elements:

```
<store-as-binary keys="true" values="true"/>
```

- Programmatically:

```
ConfigurationBuilder builder = ...  
builder.storeAsBinary().enable().storeKeysAsBinary(true).storeValuesAsBinary(true);
```

7.3. Advanced Configuration

Internally, {brandname} uses an implementation of [this Marshaller interface](#) in order to marshall/unmarshall Java objects so that they're sent over nodes in the grid, or so that they're stored in a cache store, or even so to transform them into byte arrays for lazy deserialization.

By default, {brandname} uses the [GlobalMarshaller](#). Optionally, {brandname} users can provide their own marshaller, for example:

- Via XML at the CacheManager level, under `<cache-manager />` element:

```
<serialization marshaller="com.acme.MyMarshaller"/>
```

- Programmatically:

```
GlobalConfigurationBuilder builder = ...  
builder.serialization().marshaller(myMarshaller); // needs an instance of the  
marshaller
```

7.3.1. Troubleshooting

Sometimes it might happen that the {brandname} marshalling layer, and in particular JBoss Marshalling, might have issues marshalling/unmarshalling some user object. In {brandname} 4.0, marshalling exceptions will contain further information on the objects that were being marshalled. Example:


```

java.io.NotSerializableException: java.lang.Object
at org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.java:857)
at org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.java:407)
at
org.infinispan.marshall.exts.ReplicableCommandExternalizer.writeObject(ReplicableCommandExternalizer.java:54)
at
org.infinispan.marshall.jboss.ConstantObjectTable$ExternalizerAdapter.writeObject(ConstantObjectTable.java:267)
at org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.java:143)
at org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.java:407)
at
org.infinispan.marshall.jboss.JBossMarshaller.objectToObjectStream(JBossMarshaller.java:167)
at
org.infinispan.marshall.VersionAwareMarshaller.objectToBuffer(VersionAwareMarshaller.java:92)
at
org.infinispan.marshall.VersionAwareMarshaller.objectToByteBuffer(VersionAwareMarshaller.java:170)
at
org.infinispan.marshall.DefaultMarshallerTest.testNestedNonSerializable(VersionAwareMarshallerTest.java:415)
Caused by: an exception which occurred:
in object java.lang.Object@b40ec4
in object org.infinispan.commands.write.PutKeyValueCommand@df661da7
... Removed 22 stack frames

```

The way the "in object" messages are read is the same in which stacktraces are read. The highest "in object" being the most inner one and the lowest "in object" message being the most outer one. So, the above example indicates that a `java.lang.Object` instance contained in an instance of `org.infinispan.commands.write.PutKeyValueCommand` could not be serialized because `java.lang.Object@b40ec4` is not serializable.

This is not all though! If you enable DEBUG or TRACE logging levels, marshalling exceptions will contain show the `toString()` representations of objects in the stacktrace. For example:

```

java.io.NotSerializableException: java.lang.Object
...
Caused by: an exception which occurred:
in object java.lang.Object@b40ec4
-> toString = java.lang.Object@b40ec4
in object org.infinispan.commands.write.PutKeyValueCommand@df661da7
-> toString = PutKeyValueCommand{key=k, value=java.lang.Object@b40ec4,
putIfAbsent=false, lifespanMillis=0, maxIdleTimeMillis=0}

```

With regards to unmarshalling exceptions, showing such level of information it's a lot more complicated but where possible. {brandname} will provide class type information. For example:

```
java.io.IOException: Injected failure!
at
org.infinispan.marshall.DefaultMarshallerTest$1.readExternal(VersionAwareMarshallerTest.java:426)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadNewObject(RiverUnmarshaller.java:172)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshaller.java:273)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshaller.java:210)
at org.jboss.marshalling.AbstractUnmarshaller.readObject(AbstractUnmarshaller.java:85)
at
org.infinispan.marshall.jboss.JBossMarshaller.objectFromObjectStream(JBossMarshaller.java:210)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(VersionAwareMarshaller.java:104)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(VersionAwareMarshaller.java:177)
at
org.infinispan.marshall.DefaultMarshallerTest.testErrorUnmarshalling(VersionAwareMarshallerTest.java:431)
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.DefaultMarshallerTest$1
```

In this example, an `IOException` was thrown when trying to unmarshall a instance of the inner class `org.infinispan.marshall.DefaultMarshallerTest$1`. In similar fashion to marshalling exceptions, when `DEBUG` or `TRACE` logging levels are enabled, classloader information of the class type is provided. For example:

```

java.io.IOException: Injected failure!
...
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.DefaultMarshallerTest$1
-> classloader hierarchy:
-> type classloader = sun.misc.Launcher$AppClassLoader@198dfaf
->...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/eclipse
-testng.jar
->...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/lib/testng
-jdk15.jar
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/test-classes/
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/classes/
->...file:/home/galder/.m2/repository/org/testng/testng/5.9/testng-5.9-jdk15.jar
->...file:/home/galder/.m2/repository/net/jcip/jcip-annotations/1.0/jcip-annotations
-1.0.jar
->...file:/home/galder/.m2/repository/org/easymock/easymockclassexension/2.4/easymock
classexension-2.4.jar
->...file:/home/galder/.m2/repository/org/easymock/easymock/2.4/easymock-2.4.jar
->...file:/home/galder/.m2/repository/cglib/cglib-nodep/2.1_3/cglib-nodep-2.1_3.jar
->...file:/home/galder/.m2/repository/javax/xml/bind/jaxb-api/2.1/jaxb-api-2.1.jar
->...file:/home/galder/.m2/repository/javax/xml/stream/stax-api/1.0-2/stax-api-1.0
-2.jar
->...file:/home/galder/.m2/repository/javax/activation/activation/1.1/activation
-1.1.jar
->...file:/home/galder/.m2/repository/jgroups/jgroups/2.8.0.CR1/jgroups-2.8.0.CR1.jar
->...file:/home/galder/.m2/repository/org/jboss/javaee/jboss-transaction
-api/1.0.1.GA/jboss-transaction-api-1.0.1.GA.jar
->...file:/home/galder/.m2/repository/org/jboss/marshalling/river/1.2.0.CR4
-SNAPSHOT/river-1.2.0.CR4-SNAPSHOT.jar
->...file:/home/galder/.m2/repository/org/jboss/marshalling/marshalling-api/1.2.0.CR4
-SNAPSHOT/marshalling-api-1.2.0.CR4-SNAPSHOT.jar
->...file:/home/galder/.m2/repository/org/jboss/jboss-common-core/2.2.14.GA/jboss
-common-core-2.2.14.GA.jar
->...file:/home/galder/.m2/repository/org/jboss/logging/jboss-logging
-spi/2.0.5.GA/jboss-logging-spi-2.0.5.GA.jar
->...file:/home/galder/.m2/repository/log4j/log4j/1.2.14/log4j-1.2.14.jar
->...file:/home/galder/.m2/repository/com/thoughtworks/xstream/xstream/1.2/xstream
-1.2.jar
->...file:/home/galder/.m2/repository/xpp3/xpp3_min/1.1.3.4.0/xpp3_min-1.1.3.4.0.jar
->...file:/home/galder/.m2/repository/com/sun/xml/bind/jaxb-impl/2.1.3/jaxb-impl
-2.1.3.jar
-> parent classloader = sun.misc.Launcher$ExtClassLoader@1858610
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/localedata.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunpkcs11.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunjce_provider.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/dnsns.jar
... Removed 22 stack frames
</code>

```

Finding the root cause of marshalling/unmarshalling exceptions can sometimes be really daunting

but we hope that the above improvements would help get to the bottom of those in a more quicker and efficient manner.

7.4. User Defined Externalizers

One of the key aspects of {brandname} is that it often needs to marshall/unmarshall objects in order to provide some of its functionality. For example, if it needs to store objects in a write-through or write-behind cache store, the stored objects need marshallng. If a cluster of {brandname} nodes is formed, objects shipped around need marshallng. Even if you enable lazy deserialization, objects need to be marshalled so that they can be lazily unmarshalled with the correct classloader.

Using standard JDK serialization is slow and produces payloads that are too big and can affect bandwidth usage. On top of that, JDK serialization does not work well with objects that are supposed to be immutable. In order to avoid these issues, {brandname} uses [JBoss Marshalling](#) for marshallng/unmarshalling objects. JBoss Marshalling is fast, produces very space efficient payloads, and on top of that during unmarshalling, it enables users to have full control over how to construct objects, hence allowing objects to carry on being immutable.

Starting with 5.0, users of {brandname} can now benefit from this marshallng framework as well, and they can provide their own externalizer implementations, but before finding out how to provide externalizers, let's look at the benefits they bring.

7.4.1. Benefits of Externalizers

The JDK provides a simple way to serialize objects which, in its simplest form, is just a matter of extending [java.io.Serializable](#) , but as it's well known, this is known to be slow and it generates payloads that are far too big. An alternative way to do serialization, still relying on JDK serialization, is for your objects to extend [java.io.Externalizable](#) . This allows for users to provide their own ways to marshall/unmarshall classes, but has some serious issues because, on top of relying on slow JDK serialization, it forces the class that you want to serialize to extend this interface, which has two side effects: The first is that you're forced to modify the source code of the class that you want to marshall/unmarshall which you might not be able to do because you either, don't own the source, or you don't even have it. Secondly, since Externalizable implementations do not control object creation, you're forced to add set methods in order to restore the state, hence potentially forcing your immutable objects to become mutable.

Instead of relying on JDK serialization, {brandname} uses JBoss Marshalling to serialize objects and requires any classes to be serialized to be associated with an [Externalizer](#) interface implementation that knows how to transform an object of a particular class into a serialized form and how to read an object of that class from a given input. {brandname} does not force the objects to be serialized to implement Externalizer. In fact, it is recommended that a separate class is used to implement the Externalizer interface because, contrary to JDK serialization, Externalizer implementations control how objects of a particular class are created when trying to read an object from a stream. This means that readObject() implementations are responsible of creating object instances of the target class, hence giving users a lot of flexibility on how to create these instances (whether direct instantiation, via factory or reflection), and more importantly, allows target classes to carry on being immutable. This type of externalizer architecture promotes good OOP designs principles, such as the principle of [single responsibility](#) .

It's quite common, and in general recommended, that Externalizer implementations are stored as inner static public classes within classes that they externalize. The advantages of doing this is that related code stays together, making it easier to maintain. In {brandname}, there are two ways in which {brandname} can be plugged with user defined externalizers:

7.4.2. User Friendly Externalizers

In the simplest possible form, users just need to provide an [Externalizer](#) implementation for the type that they want to marshall/unmarshall, and then annotate the marshalled type class with {@link SerializeWith} annotation indicating the externalizer class to use. For example:

```
import org.infinispan.commons.marshall.Externalizer;
import org.infinispan.commons.marshall.SerializeWith;

@SerializeWith(Person.PersonExternalizer.class)
public class Person {

    final String name;
    final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public static class PersonExternalizer implements Externalizer<Person> {
        @Override
        public void writeObject(ObjectOutput output, Person person)
            throws IOException {
            output.writeObject(person.name);
            output.writeInt(person.age);
        }

        @Override
        public Person readObject(ObjectInput input)
            throws IOException, ClassNotFoundException {
            return new Person((String) input.readObject(), input.readInt());
        }
    }
}
```

At runtime JBoss Marshalling will inspect the object and discover that it is marshallable (thanks to the annotation) and so marshall it using the externalizer class passed. To make externalizer implementations easier to code and more typesafe, make sure you define type `<T>` as the type of object that's being marshalled/unmarshalled.

Even though this way of defining externalizers is very user friendly, it has some disadvantages:

- Due to several constraints of the model, such as support for different versions of the same class

or the need to marshall the Externalizer class, the payload sizes generated via this method are not the most efficient.

- This model requires that the marshalled class be annotated with [link:https://docs.jboss.org/infinispan/10.0/apidocs/org/infinispan/commons/marshall/SerializeWith.html](https://docs.jboss.org/infinispan/10.0/apidocs/org/infinispan/commons/marshall/SerializeWith.html) but a user might need to provide an Externalizer for a class for which source code is not available, or for any other constraints, it cannot be modified.
- The use of annotations by this model might be limiting for framework developers or service providers that try to abstract lower level details, such as the marshalling layer, away from the user.

If you're affected by any of these disadvantages, an alternative method to provide externalizers is available via more advanced externalizers:

7.4.3. Advanced Externalizers

[AdvancedExternalizer](#) provides an alternative way to provide externalizers for marshalling/unmarshalling user defined classes that overcome the deficiencies of the more user-friendly externalizer definition model explained in Externalizer. For example:

```

import org.infinispan.marshall.AdvancedExternalizer;

public class Person {

    final String name;
    final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public static class PersonExternalizer implements AdvancedExternalizer<Person> {
        @Override
        public void writeObject(ObjectOutput output, Person person)
            throws IOException {
            output.writeObject(person.name);
            output.writeInt(person.age);
        }

        @Override
        public Person readObject(ObjectInput input)
            throws IOException, ClassNotFoundException {
            return new Person((String) input.readObject(), input.readInt());
        }

        @Override
        public Set<Class<? extends Person>> getTypeClasses() {
            return Util.<Class<? extends Person>>asSet(Person.class);
        }

        @Override
        public Integer getId() {
            return 2345;
        }
    }
}

```

The first noticeable difference is that this method does not require user classes to be annotated in anyway, so it can be used with classes for which source code is not available or that cannot be modified. The bound between the externalizer and the classes that are marshalled/unmarshalled is set by providing an implementation for `getTypeClasses()` which should return the list of classes that this externalizer can marshal:

Linking Externalizers with Marshaller Classes

Once the Externalizer's `readObject()` and `writeObject()` methods have been implemented, it's time to link them up together with the type classes that they externalize. To do so, the Externalizer implementation must provide a `getTypeClasses()` implementation. For example:

```
import org.infinispan.commons.util.Util;
...
@Override
public Set<Class<? extends ReplicableCommand>> getTypeClasses() {
    return Util.asSet(LockControlCommand.class, RehashControlCommand.class,
        StateTransferControlCommand.class, GetKeyValueCommand.class,
        ClusteredGetCommand.class,
        SingleRpcCommand.class, CommitCommand.class,
        PrepareCommand.class, RollbackCommand.class,
        ClearCommand.class, EvictCommand.class,
        InvalidateCommand.class, InvalidateL1Command.class,
        PutKeyValueCommand.class, PutMapCommand.class,
        RemoveCommand.class, ReplaceCommand.class);
}
```

In the code above, `ReplicableCommandExternalizer` indicates that it can externalize several type of commands. In fact, it marshalls all commands that extend `ReplicableCommand` interface, but currently the framework only supports class equality comparison and so, it's not possible to indicate that the classes to be marshalled are all children of a particular class/interface.

However there might be sometimes when the classes to be externalized are private and hence it's not possible to reference the actual class instance. In this situation, users can attempt to look up the class with the given fully qualified class name and pass that back. For example:

```
@Override
public Set<Class<? extends List>> getTypeClasses() {
    return Util.<Class<? extends List>>asSet(
        Util.loadClass("java.util.Collections$SingletonList"));
}
```

Externalizer Identifier

Secondly, in order to save the maximum amount of space possible in the payloads generated, advanced externalizers require externalizer implementations to provide a positive identifier via `getId()` implementations or via XML/programmatic configuration that identifies the externalizer when unmarshalling a payload. In order for this to work however, advanced externalizers require externalizers to be registered on cache manager creation time via XML or programmatic configuration which will be explained in next section. On the contrary, externalizers based on `Externalizer` and `SerializeWith` require no pre-registration whatsoever. Internally, `{brandname}` uses this advanced externalizer mechanism in order to marshal/unmarshal internal classes.

So, `getId()` should return a positive integer that allows the externalizer to be identified at read time to figure out which `Externalizer` should read the contents of the incoming buffer, or it can return null. If `getId()` returns null, it is indicating that the id of this advanced externalizer will be defined via XML/programmatic configuration, which will be explained in next section.

Regardless of the source of the id, using a positive integer allows for very efficient variable length encoding of numbers, and it's much more efficient than shipping externalizer

implementation class information or class name around. {brandname} users can use any positive integer as long as it does not clash with any other identifier in the system. It's important to understand that a user defined externalizer can even use the same numbers as the externalizers in the {brandname} Core project because the internal {brandname} Core externalizers are special and they use a different number space to the user defined externalizers. On the contrary, users should avoid using numbers that are within the pre-assigned identifier ranges which can be found at the end of this article. {brandname} checks for id duplicates on startup, and if any are found, startup is halted with an error.

When it comes to maintaining which ids are in use, it's highly recommended that this is done in a centralized way. For example, getId() implementations could reference a set of statically defined identifiers in a separate class or interface. Such class/interface would give a global view of the identifiers in use and so can make it easier to assign new ids.

Registering Advanced Externalizers

The following example shows the type of configuration required to register an advanced externalizer implementation for Person object shown earlier stored as a static inner class within it:

infinispan.xml

```
<infinispan>
  <cache-container>
    <serialization>
      <advanced-externalizer class="Person$PersonExternalizer"/>
    </serialization>
  </cache-container>
  ...
</infinispan>
```

Programmatically:

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
    .addAdvancedExternalizer(new Person.PersonExternalizer());
```

As mentioned earlier, when listing these externalizer implementations, users can optionally provide the identifier of the externalizer via XML or programmatically instead of via getId() implementation. Again, this offers a centralized way to maintain the identifiers but it's important that the rules are clear: An AdvancedExternalizer implementation, either via XML/programmatic configuration or via annotation, needs to be associated with an identifier. If it isn't, {brandname} will throw an error and abort startup. If a particular AdvancedExternalizer implementation defines an id both via XML/programmatic configuration and annotation, the value defined via XML/programmatically is the one that will be used. Here's an example of an externalizer whose id is defined at registration time:

infinispan.xml

```
<infinispan>
  <cache-container>
    <serialization>
      <advanced-externalizer id="123"
                             class="Person$PersonExternalizer"/>
    </serialization>
  </cache-container>
  ...
</infinispan>
```

Programmatically:

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
  .addAdvancedExternalizer(123, new Person.PersonExternalizer());
```

Finally, a couple of notes about the programmatic configuration. `GlobalConfiguration.addExternalizer()` takes varargs, so it means that it is possible to register multiple externalizers in just one go, assuming that their ids have already been defined via `@Marshall`s annotation. For example:

```
builder.serialization()
  .addAdvancedExternalizer(new Person.PersonExternalizer(),
                           new Address.AddressExternalizer());
```

Preassigned Externalizer Id Ranges

This is the list of Externalizer identifiers that are used by {brandname} based modules or frameworks. {brandname} users should avoid using ids within these ranges.

{brandname} Tree Module:	1000 - 1099
{brandname} Server Modules:	1100 - 1199
Hibernate {brandname} Second Level Cache:	1200 - 1299
{brandname} Lucene Directory:	1300 - 1399
Hibernate OGM:	1400 - 1499
Hibernate Search:	1500 - 1599
{brandname} Query Module:	1600 - 1699
{brandname} Remote Query Module:	1700 - 1799
{brandname} Scripting Module:	1800 - 1849
{brandname} Server Event Logger Module:	1850 - 1899
{brandname} Remote Store:	1900 - 1999

{brandname} Counters:	2000 - 2049
{brandname} Multimap:	2050 - 2099
{brandname} Locks:	2100 - 2149

Chapter 8. Transactions

{brandname} 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 {brandname} 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](#).

{brandname} 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 {brandname} 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 {brandname} 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](#).

WARN: [DummyTransactionManagerLookup](#) has been deprecated in 9.0 and it will be removed in the future. Use [EmbeddedTransactionManagerLookup](#) instead.

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();
```

8.1. 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"
  write-skew="false"/>
<transaction
  locking="OPTIMISTIC"
  auto-commit="true"
  complete-timeout="60000"
  mode="NONE"
  notifications="true"
  protocol="DEFAULT"
  reaper-interval="30000"
  recovery-cache="__recoveryInfoCacheName__"
  stop-timeout="30000"
  transaction-manager-lookup=
"org.infinispan.transaction.lookup.GenericTransactionManagerLookup"/>
<versioning
  scheme="NONE"/>
```

or programmatically:

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.locking()
    .isolationLevel(IsolationLevel.READ_COMMITTED)
    .writeSkewCheck(false);
builder.transaction()
    .lockingMode(LockingMode.OPTIMISTIC)
    .autoCommit(true)
    .completedTxTimeout(60000)
    .transactionMode(TransactionMode.NON_TRANSACTIONAL)
    .useSynchronization(false)
    .notifications(true)
    .transactionProtocol(TransactionProtocol.DEFAULT)
    .reaperWakeUpInterval(30000)
    .cacheStopTimeout(30000)
    .transactionManagerLookup(new GenericTransactionManagerLookup())
    .recovery()
    .enabled(false)
    .recoveryInfoCacheName("__recoveryInfoCacheName__");
builder.versioning()
    .enabled(false)
    .scheme(VersioningScheme.NONE);
```

- **isolation** - configures the isolation level. Check section [Isolation Levels](#) for more details. Default

is `REPEATABLE_READ`.

- `write-skew` - enables write skew checks (deprecated). {brandname} automatically sets this attribute in Library Mode. Default is `false` for `READ_COMMITTED`. Default is `true` for `REPEATABLE_READ`. See [Write Skews](#) for more details.
- `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`.
- `protocol` - configures the protocol uses. Default is `DEFAULT`. Values available are:
 - `DEFAULT` - uses the traditional Two-Phase-Commit protocol. It is described below.
 - `TOTAL_ORDER` - uses total order ensured by the `Transport` to commit transactions. Check section [Total Order based commit protocol](#) for details.
- `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`.
- Versioning `scheme` - configure the version scheme to use when write skew is enabled with optimistic or total order transactions. Check section [Write Skews](#) for more details. Default is `NONE`.

For more details on how Two-Phase-Commit (2PC) is implemented in {brandname} and how locks are being acquired see the section below. More details about the configuration settings are

available in [Configuration reference](#).

8.2. Isolation levels

{brandname} 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 {brandname}. 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()
```

8.3. Transaction locking

8.3.1. 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 succeeds, 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).



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

8.3.2. 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.
```



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

8.3.3. 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.

8.4. 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.

In Library Mode, {brandname} automatically performs write skew checks to ensure data consistency for `REPEATABLE_READ` isolation levels in optimistic transactions. This allows {brandname} to detect and roll back one of the transactions.



The `write-skew` attribute is deprecated for Library Mode. In Remote Client/Server Mode, this attribute is not a valid declaration.

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.

In clustered environments, you should configure data versioning to ensure reliable write skew checks. {brandname} provides an implementation of the `EntryVersion` interface called `SIMPLE` versioning, which is backed by a long that is incremented each time the entry is updated.

```
<versioning scheme="SIMPLE|NONE" />
```

Or

```
new ConfigurationBuilder().versioning().scheme(SIMPLE);
```

8.4.1. 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`.



- 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
```

8.5. 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.

8.6. Enlisting Synchronizations

By default {brandname} registers itself as a first class participant in distributed transactions through [XAResource](#). There are situations where {brandname} 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 {brandname} is used as a 2nd level cache in Hibernate.

{brandname} 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 {brandname} 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 {brandname} as a [Synchronisation](#) makes the [TransactionManager](#) skip writing the log to the disk (performance improvement).

8.7. 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.



Generally speaking, one should use batching API whenever the only participant in the transaction is an {brandname} 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 {brandname}, then batching can be used. If one account is in a database and the other is {brandname}, then distributed transactions are required.



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

8.7.1. 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
```

8.7.2. 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.

8.8. 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.

8.8.1. 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 {brandname}. When `TransactionManager.commit()` is invoked, both resources prepare successfully (1st phase). During the commit (2nd) phase, the database successfully applies the changes whilst {brandname} 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 {brandname} (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 {brandname} ends up in a consistent state.

8.8.2. How does it work

Recovery is coordinated by the transaction manager. The transaction manager works with {brandname} 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 {brandname} cluster and replay the commit of transactions or force the rollback. {brandname} provides JMX tooling for this - this is explained extensively in the [Transaction recovery and reconciliation](#) section.

8.8.3. Configuring recovery

Recovery is *not* enabled by default in {brandname}. If disabled, the `TransactionManager` won't be able to work with {brandname} 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.



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

Enable JMX support

In order to be able to use JMX for managing recovery JMX support must be explicitly enabled. More about enabling JMX in the [Management](#) chapter.

8.8.4. Recovery cache

In order to track in-doubt transactions and be able to reply them, {brandname} 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 `{brandname}` will configure a local cache for you.

It is possible (though not mandated) to share same recovery cache between all the `{brandname}` 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.

8.8.5. 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 `{brandname}` `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.

8.8.6. 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 `{brandname}` server through JMX, and lists the in doubt transactions. The image below demonstrates JConsole connecting to an `{brandname}` node that has an in doubt transaction.

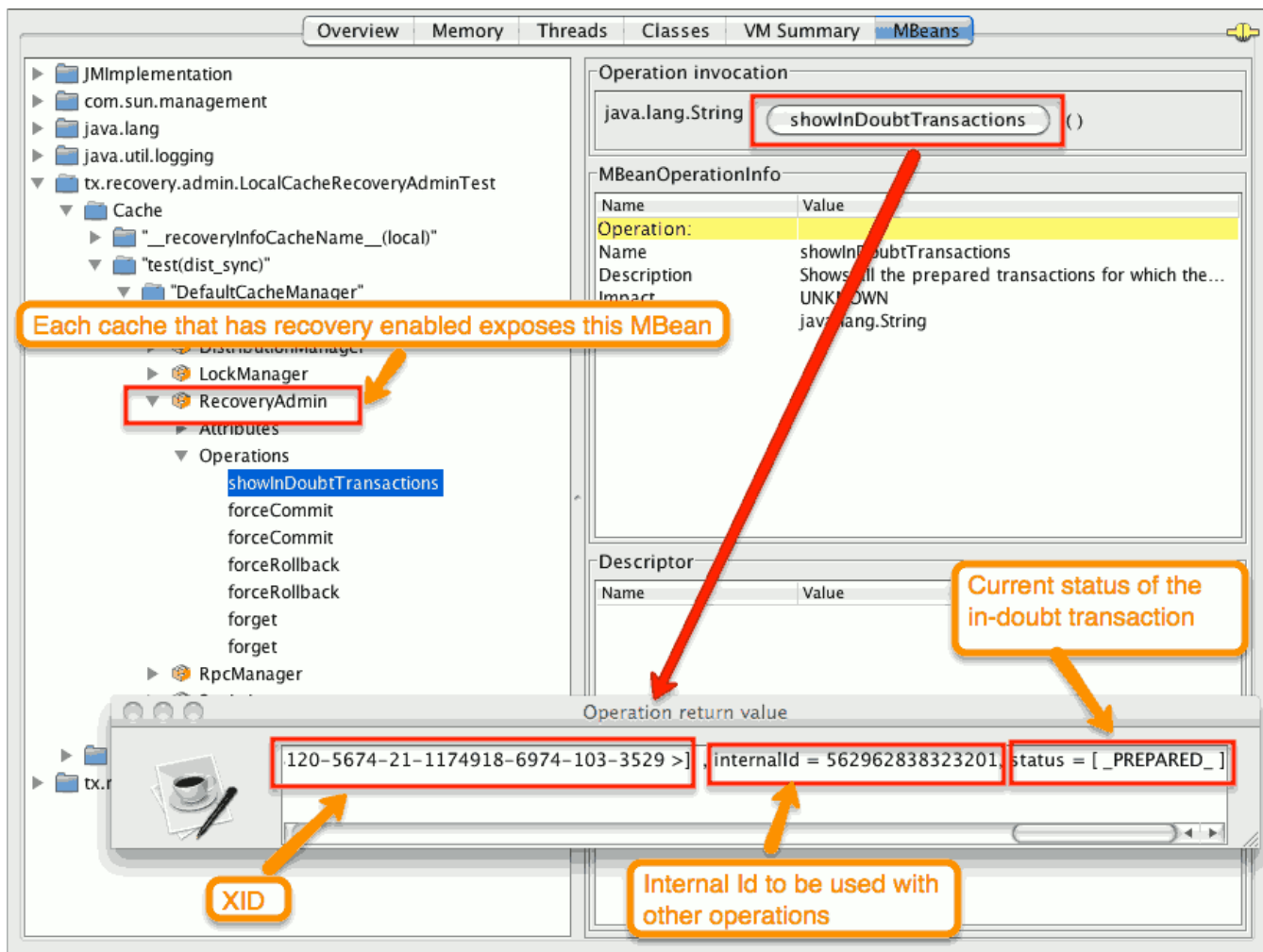


Figure 6. Show in-doubt transactions

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 {brandname} 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 {brandname}'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.

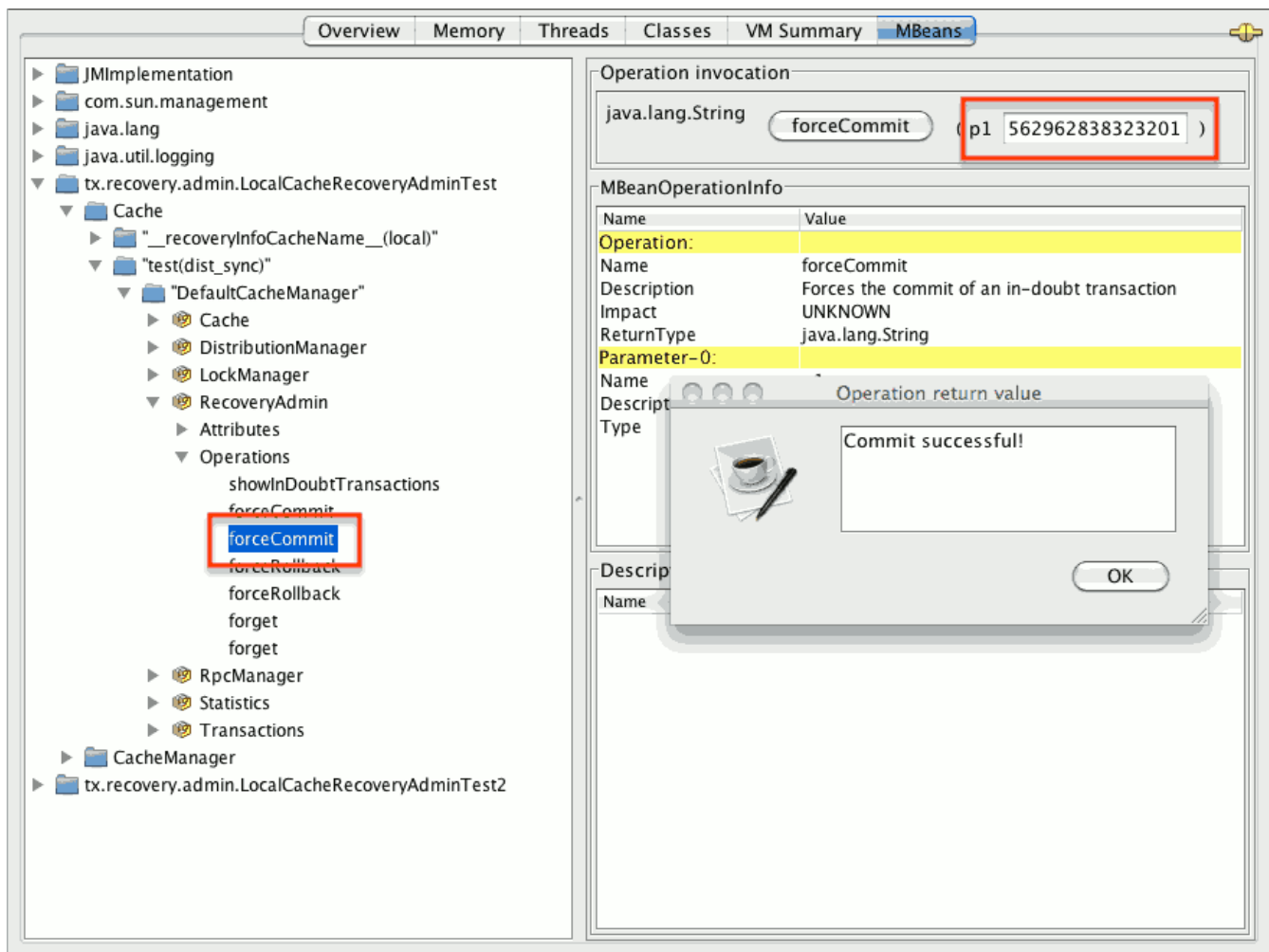


Figure 7. Force commit



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

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.

8.8.7. Want to know more?

The [recovery design document](#) describes in more detail the insides of transaction recovery implementation.

8.9. Total Order based commit protocol

The Total Order based protocol is a multi-master scheme (in this context, multi-master scheme means that all nodes can update all the data) as the (optimistic/pessimist) locking mode implemented in {brandname}. This commit protocol relies on the concept of totally ordered

delivery of messages which, informally, implies that each node which delivers a set of messages, delivers them in the same order.

This protocol comes with this advantages.

1. transactions can be committed in one phase, as they are delivered in the same order by the nodes that receive them.
2. it mitigates distributed deadlocks.

The weaknesses of this approach are the fact that its implementation relies on a single thread per node which delivers the transaction and its modification, and the slightly cost of total ordering the messages in **Transport**.

Thus, this protocol delivers best performance in scenarios of *high contention* , in which it can benefit from the single-phase commit and the deliver thread is not the bottleneck.

Currently, the Total Order based protocol is available only in *transactional* caches for *replicated* and *distributed* modes.

8.9.1. Overview

The Total Order based commit protocol only affects how transactions are committed by {brandname} and the isolation level and write skew affects it behaviour.

When write skew is disabled, the transaction can be committed/rolled back in single phase. The data consistency is guaranteed by the **Transport** that ensures that all owners of a key will deliver the same transactions set by the same order.

On other hand, when write skew is enabled, the protocol adapts and uses one phase commit when it is safe. In **XaResource** enlistment, we can use one phase if the **TransactionManager** request a commit in one phase (last resource commit optimization) and the {brandname} cache is configured in replicated mode. This optimization is not safe in distributed mode because each node performs the write skew check validation in different keys subset. When in **Synchronization** enlistment, the **TransactionManager** does not provide any information if {brandname} is the only resource enlisted (last resource commit optimization), so it is not possible to commit in a single phase.

Commit in one phase

When the transaction ends, {brandname} sends the transaction (and its modification) in total order. This ensures all the transactions are deliver in the same order in all the involved {brandname} nodes. As a result, when a transaction is delivered, it performs a deterministic write skew check over the same state (if enabled), leading to the same outcome (transaction commit or rollback).

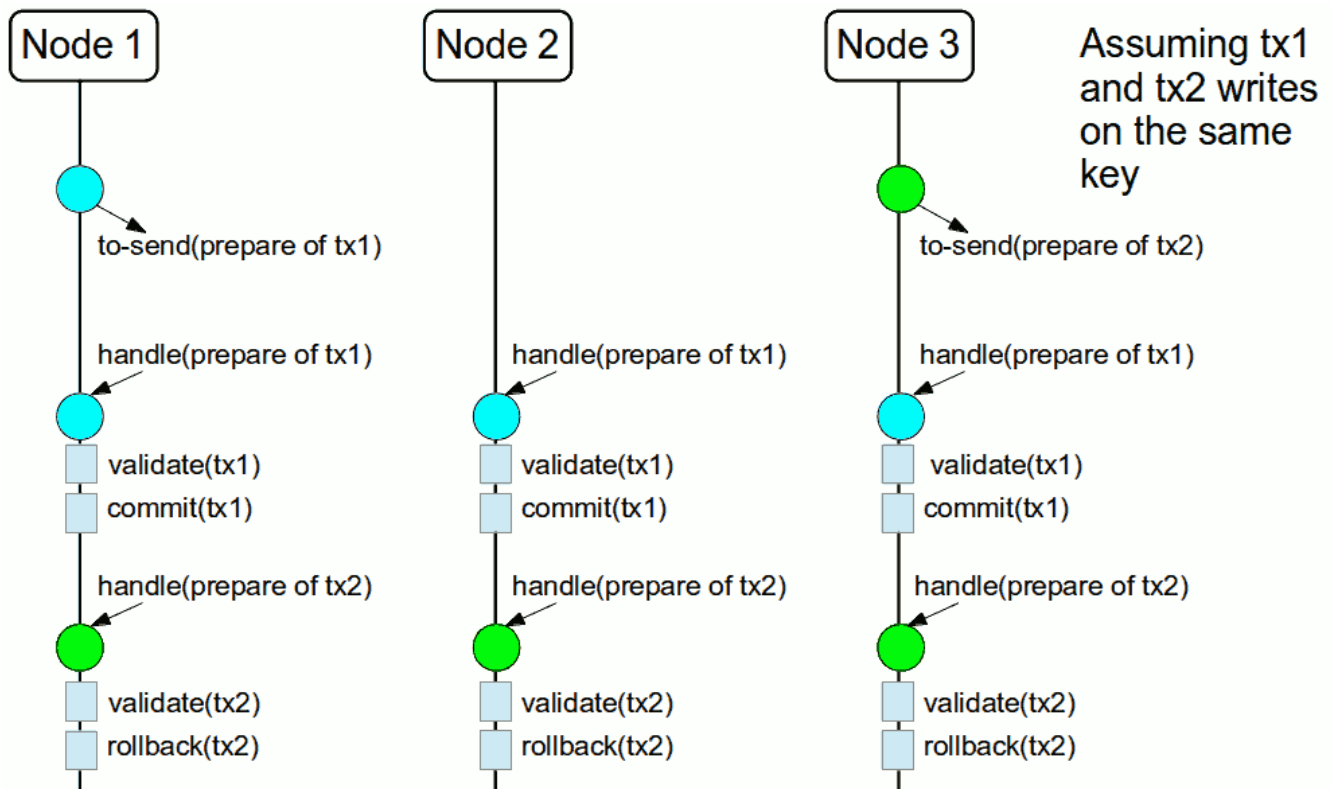


Figure 8. 1-phase commit

The figure above demonstrates a high level example with 3 nodes. Node1 and Node3 are running one transaction each and lets assume that both transaction writes on the same key. To make it more interesting, lets assume that both nodes tries to commit at the same time, represented by the first colored circle in the figure. The blue circle represents the transaction tx1 and the green the transaction tx2 . Both nodes do a remote invocation in total order (to-send) with the transaction's modifications. At this moment, all the nodes will agree in the same deliver order, for example, tx1 followed by tx2 . Then, each node delivers tx1 , perform the validation and commits the modifications. The same steps are performed for tx2 but, in this case, the validation will fail and the transaction is rollback in all the involved nodes.

Commit in two phases

In the first phase, it sends the modification in total order and the write skew check is performed. The result of the write skew check is sent back to the originator. As soon as it has the confirmation that all keys are successfully validated, it give a positive response to the TransactionManager. On other hand, if it receives a negative reply, it returns a negative response to the TransactionManager. Finally, the transaction is committed or aborted in the second phase depending of the TransactionManager request.

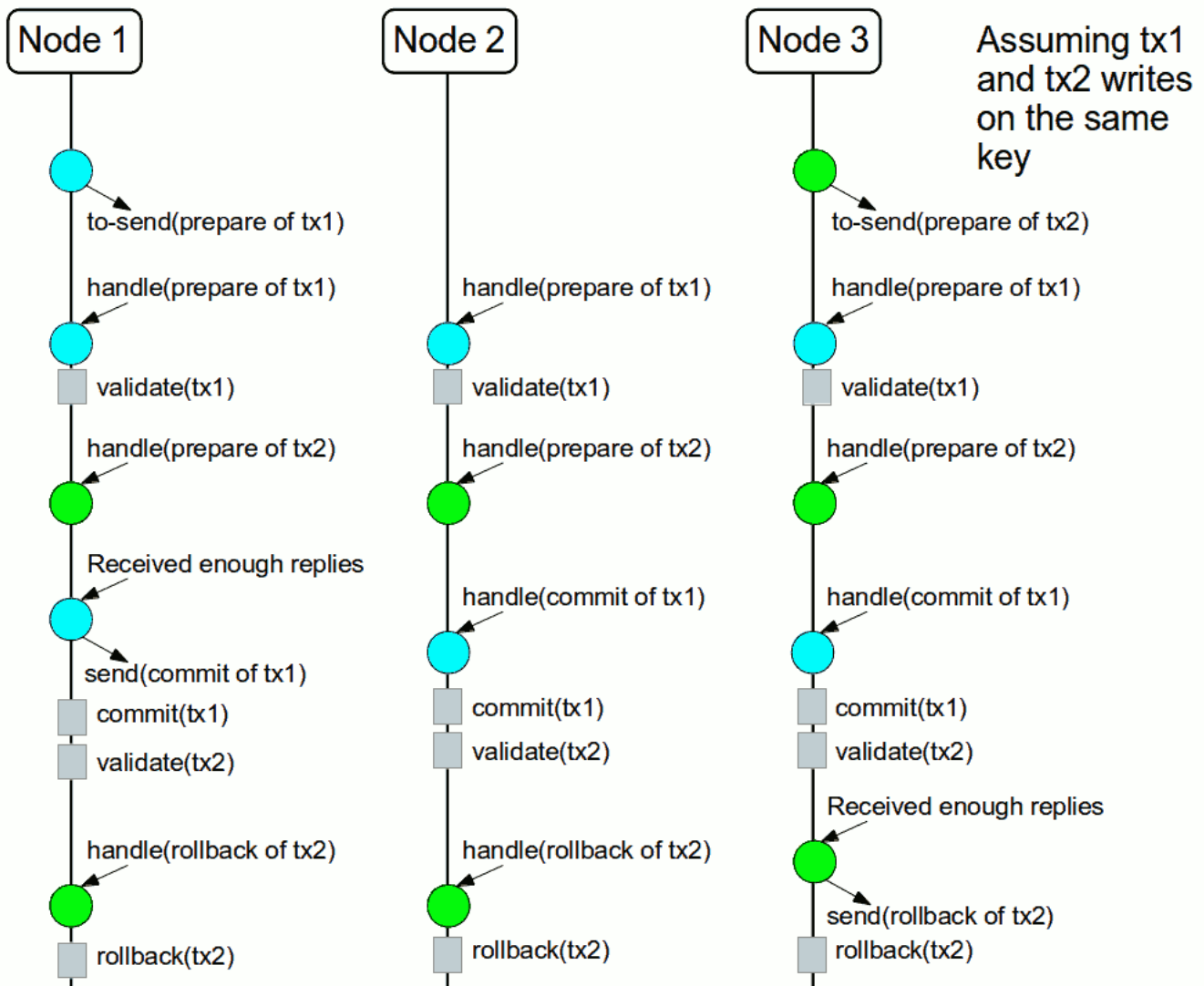


Figure 9. 2-phase commit

The figure above shows the scenario described in the first figure but now committing the transactions using two phases. When *tx1* is deliver, it performs the validation and it replies to the **TransactionManager**. Next, lets assume that *tx2* is deliver before the **TransactionManager** request the second phase for *tx1*. In this case, *tx2* will be enqueued and it will be validated only when *tx1* is completed. Eventually, the **TransactionManager** for *tx1* will request the second phase (the commit) and all the nodes are free to perform the validation of *tx2*.

Transaction Recovery

Transaction recovery is currently not available for Total Order based commit protocol.

State Transfer

For simplicity reasons, the total order based commit protocol uses a blocking version of the current state transfer. The main differences are:

1. enqueue the transaction deliver while the state transfer is in progress;
2. the state transfer control messages (**CacheTopologyControlCommand**) are sent in total order.

This way, it provides a synchronization between the state transfer and the transactions deliver that is the same all the nodes. Although, the transactions caught in the middle of state transfer (i.e. sent

before the state transfer start and deliver after it) needs to be re-sent to find a new total order involving the new joiners.

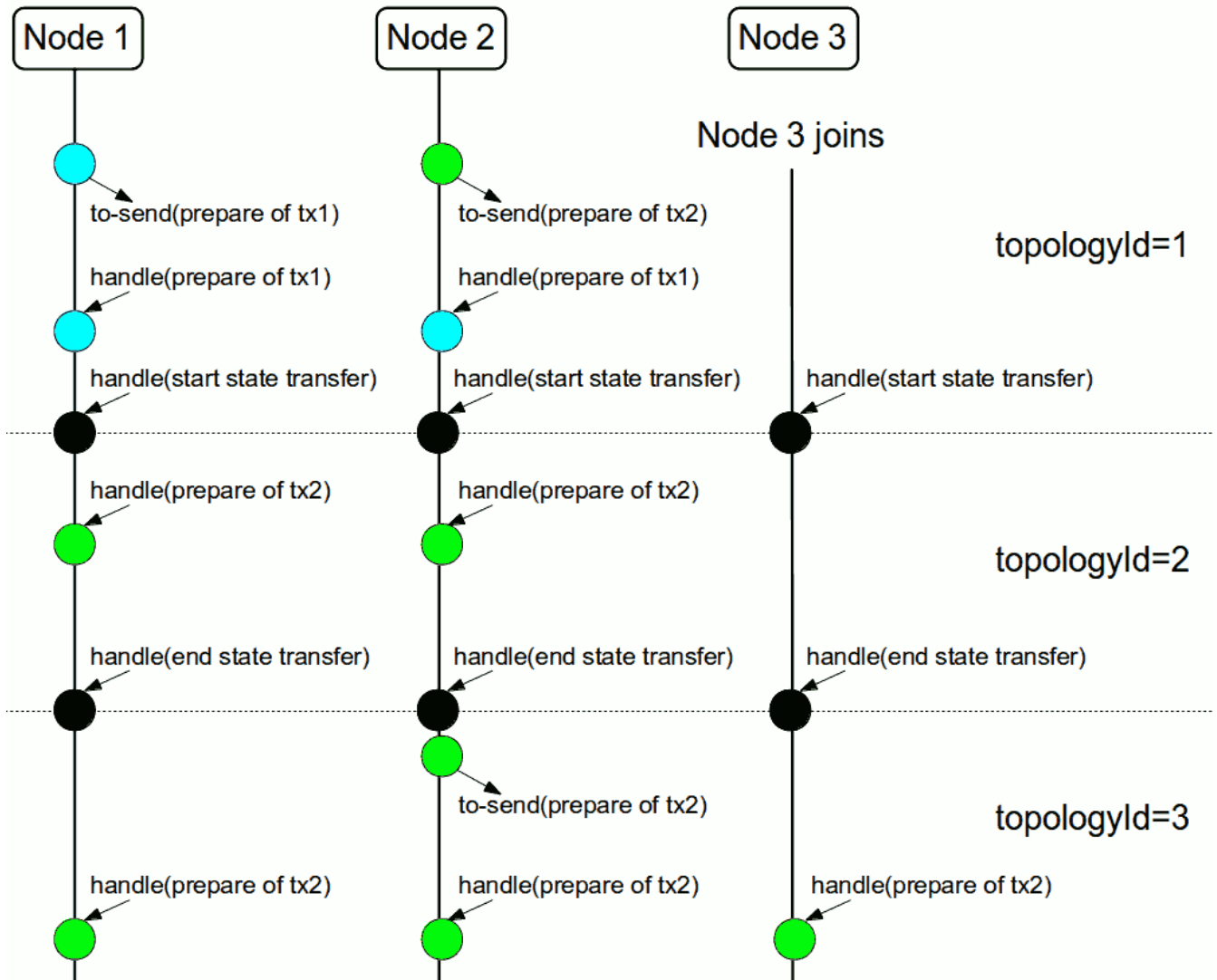


Figure 10. Node joining during transaction

The figure above describes a node joining. In the scenario, the tx2 is sent in *topologyId=1* but when it is received, it is in *topologyId=2*. So, the transaction is re-sent involving the new nodes.

8.9.2. Configuration

To use total order in your cache, you need to add the TOA protocol in your `jgroups.xml` configuration file.

jgroups.xml

```
<tom.TOA />
```



Check the [JGroups Manual](#) for more details.



If you are interested in detail how JGroups guarantees total order, check the link: <http://jgroups.org/manual/index.html#TOA> [TOA manual].

Also, you need to set the `protocol=TOTAL_ORDER` in the `<transaction>` element, as shown in [Transaction configuration](#).

8.9.3. When to use it?

Total order shows benefits when used in write intensive and high contented workloads. It avoids contention in the lock keys.

Chapter 9. Locking and Concurrency

{brandname} makes use of multi-versioned concurrency control ([MVCC](#)) - 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, including:

- 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

9.1. Locking implementation details

{brandname}'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, {brandname}'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.

9.1.1. How does it work in clustered caches?

In clustered caches, each key has a node responsible to lock the key. This node is called primary owner.

Non Transactional caches

1. The write operation is sent to the primary owner of the key.
2. The primary owner tries to lock the key.
 - a. If it succeeds, it forwards the operation to the other owners;
 - b. Otherwise, an exception is thrown.



If the operation is conditional and it fails on the primary owner, it is not forwarded to the other owners.



If the operation is executed locally in the primary owner, the first step is skipped.

9.1.2. Transactional caches

The transactional cache supports optimistic and pessimistic locking mode. Check section [Transaction Locking](#) for more information about it.

9.1.3. Isolation levels

Isolation level affects what transactions can read when running concurrently with other transaction. Check section [Isolation Levels](#) for more details about it.

9.1.4. 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.

9.1.5. 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);
```

9.1.6. 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 **DataContainers**. 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 {brandname}.

Configuration example:

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

Or

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

9.1.7. 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);
```

9.1.8. 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.

9.2. Data Versioning

{brandname} supports two forms of data versioning: simple and external. The simple versioning is used in transactional caches for write skew check. See [Write Skews](#).

The external versioning is used to encapsulate an external source of data versioning within {brandname}, such as when using {brandname} 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.



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

Chapter 10. Executing code in the Grid

The main benefit of a Cache is the ability to very quickly lookup a value by its key, even across machines. In fact this use alone is probably the reason many users use {brandname}. However {brandname} can provide many more benefits that aren't immediately apparent. Since {brandname} is usually used in a cluster of machines we also have features available that can help utilize the entire cluster for performing the user's desired workload.



This section covers only executing code in the grid using an embedded cache, if you are using a remote cache you should check out [Executing code in the Remote Grid](#).

10.1. Cluster Executor

Since you have a group of machines, it makes sense to leverage their combined computing power for executing code on all of them. The cache manager comes with a nice utility that allows you to execute arbitrary code in the cluster. Note this feature requires no Cache to be used. This [Cluster Executor](#) can be retrieved by calling `executor()` on the `EmbeddedCacheManager`. This executor is retrievable in both clustered and non clustered configurations.



The `ClusterExecutor` is specifically designed for executing code where the code is not reliant upon the data in a cache and is used instead as a way to help users to execute code easily in the cluster.

This manager was built specifically using Java 8 and such has functional APIs in mind, thus all methods take a functional interface as an argument. Also since these arguments will be sent to other nodes they need to be serializable. We even used a nice trick to ensure our lambdas are immediately Serializable. That is by having the arguments implement both Serializable and the real argument type (ie. Runnable or Function). The JRE will pick the most specific class when determining which method to invoke, so in that case your lambdas will always be serializable. It is also possible to use an Externalizer to possibly reduce message size further.

The manager by default will submit a given command to all nodes in the cluster including the node where it was submitted from. You can control on which nodes the task is executed on by using the `filterTargets` methods as is explained in the section.

10.1.1. Filtering execution nodes

It is possible to limit on which nodes the command will be ran. For example you may want to only run a computation on machines in the same rack. Or you may want to perform an operation once in the local site and again on a different site. A cluster executor can limit what nodes it sends requests to at the scope of same or different machine, rack or site level.

SameRack.java

```
EmbeddedCacheManager manager = ...;
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_RACK).submit(...)
```

To use this topology base filtering you must enable topology aware consistent hashing through [Server Hinting](#).

You can also filter using a predicate based on the **Address** of the node. This can also be optionally combined with topology based filtering in the previous code snippet.

We also allow the target node to be chosen by any means using a **Predicate** that will filter out which nodes can be considered for execution. Note this can also be combined with Topology filtering at the same time to allow even more fine control of where you code is executed within the cluster.

Predicate.java

```
EmbeddedCacheManager manager = ...;
// Just filter
manager.executor().filterTargets(a -> a.equals(..)).submit(...)
// Filter only those in the desired topology
manager.executor().filterTargets(ClusterExecutionPolicy.SAME_SITE, a -> a.equals(
..)).submit(...)
```

10.1.2. Timeout

Cluster Executor allows for a timeout to be set per invocation. This defaults to the distributed sync timeout as configured on the Transport Configuration. This timeout works in both a clustered and non clustered cache manager. The executor may or may not interrupt the threads executing a task when the timeout expires. However when the timeout occurs any **Consumer** or **Future** will be completed passing back a **TimeoutException**. This value can be overridden by invoking the [timeout](#) method and supplying the desired duration.

10.1.3. Single Node Submission

Cluster Executor can also run in single node submission mode instead of submitting the command to all nodes it will instead pick one of the nodes that would have normally received the command and instead submit it to only one. Each submission will possibly use a different node to execute the task on. This can be very useful to use the ClusterExecutor as a **java.util.concurrent.Executor** which you may have noticed that ClusterExecutor implements.

SingleNode.java

```
EmbeddedCacheManager manager = ...;
manager.executor().singleNodeSubmission().submit(...)
```

Failover

When running in single node submission it may be desirable to also allow the Cluster Executor handle cases where an exception occurred during the processing of a given command by retrying the command again. When this occurs the Cluster Executor will choose a single node again to resubmit the command to up to the desired number of failover attempts. Note the chosen node could be any node that passes the topology or predicate check. Failover is enabled by invoking the overridden [singleNodeSubmission](#) method. The given command will be resubmitted again to a single node until either the command completes without exception or the total submission amount is equal to the provided failover count.

10.1.4. Example: PI Approximation

This example shows how you can use the ClusterExecutor to estimate the value of PI.

Pi approximation can greatly benefit from parallel distributed execution via Cluster Executor. Recall that area of the square is $S_a = 4r^2$ and area of the circle is $C_a = \pi r^2$. Substituting r^2 from the second equation into the first one it turns out that $\pi = 4 * C_a / S_a$. Now, imagine that we can shoot very large number of darts into a square; if we take ratio of darts that land inside a circle over a total number of darts shot we will approximate C_a / S_a value. Since we know that $\pi = 4 * C_a / S_a$ we can easily derive approximate value of pi. The more darts we shoot the better approximation we get. In the example below we shoot 1 billion darts but instead of "shooting" them serially we parallelize work of dart shooting across the entire {brandname} cluster. Note this will work in a cluster of 1 was well, but will be slower.

```
public class PiAppx {

    public static void main (String [] arg){
        EmbeddedCacheManager cacheManager = ..
        boolean isCluster = ..

        int numPoints = 1_000_000_000;
        int numServers = isCluster ? cacheManager.getMembers().size() : 1;
        int numberPerWorker = numPoints / numServers;

        ClusterExecutor clusterExecutor = cacheManager.executor();
        long start = System.currentTimeMillis();
        // We receive results concurrently - need to handle that
        AtomicLong countCircle = new AtomicLong();
        CompletableFuture<Void> fut = clusterExecutor.submitConsumer(m -> {
            int insideCircleCount = 0;
            for (int i = 0; i < numberPerWorker; i++) {
                double x = Math.random();
                double y = Math.random();
                if (insideCircle(x, y))
                    insideCircleCount++;
            }
            return insideCircleCount;
        }, (address, count, throwable) -> {
            if (throwable != null) {
```

```

        throwable.printStackTrace();
        System.out.println("Address: " + address + " encountered an error: " +
throwable);
    } else {
        countCircle.getAndAdd(count);
    }
});
fut.whenComplete((v, t) -> {
    // This is invoked after all nodes have responded with a value or exception
    if (t != null) {
        t.printStackTrace();
        System.out.println("Exception encountered while waiting:" + t);
    } else {
        double appxPi = 4.0 * countCircle.get() / numPoints;

        System.out.println("Distributed PI appx is " + appxPi +
            " using " + numServers + " node(s), completed in " + (System
.currentTimeMillis() - start) + " ms");
    }
});

// May have to sleep here to keep alive if no user threads left
}

private static boolean insideCircle(double x, double y) {
    return (Math.pow(x - 0.5, 2) + Math.pow(y - 0.5, 2))
        <= Math.pow(0.5, 2);
}
}

```

10.2. Streams

You may want to process a subset or all data in the cache to produce a result. This may bring thoughts of Map Reduce. {brandname} allows the user to do something very similar but utilizes the standard JRE APIs to do so. Java 8 introduced the concept of a [Stream](#) which allows functional-style operations on collections rather than having to procedurally iterate over the data yourself. Stream operations can be implemented in a fashion very similar to MapReduce. Streams, just like MapReduce allow you to perform processing upon the entirety of your cache, possibly a very large data set, but in an efficient way.



Streams are the preferred method when dealing with data that exists in the cache because streams automatically adjust to cluster topology changes.

Also since we can control how the entries are iterated upon we can more efficiently perform the operations in a cache that is distributed if you want it to perform all of the operations across the cluster concurrently.

A stream is retrieved from the [entrySet](#), [keySet](#) or [values](#) collections returned from the Cache by invoking the [stream](#) or [parallelStream](#) methods.

10.2.1. Common stream operations

This section highlights various options that are present irrespective of what type of underlying cache you are using.

10.2.2. Key filtering

It is possible to filter the stream so that it only operates upon a given subset of keys. This can be done by invoking the `filterKeys` method on the `CacheStream`. This should always be used over a Predicate `filter` and will be faster if the predicate was holding all keys.

If you are familiar with the `AdvancedCache` interface you may be wondering why you even use `getAll` over this `keyFilter`. There are some small benefits (mostly smaller payloads) to using `getAll` if you need the entries as is and need them all in memory in the local node. However if you need to do processing on these elements a stream is recommended since you will get both distributed and threaded parallelism for free.

10.2.3. Segment based filtering



This is an advanced feature and should only be used with deep knowledge of {brandname} segment and hashing techniques. These segments based filtering can be useful if you need to segment data into separate invocations. This can be useful when integrating with other tools such as [Apache Spark](#).

This option is only supported for replicated and distributed caches. This allows the user to operate upon a subset of data at a time as determined by the `KeyPartitioner`. The segments can be filtered by invoking `filterKeySegments` method on the `CacheStream`. This is applied after the key filter but before any intermediate operations are performed.

10.2.4. Local/Invalidation

A stream used with a local or invalidation cache can be used just the same way you would use a stream on a regular collection. {brandname} handles all of the translations if necessary behind the scenes and works with all of the more interesting options (ie. `storeAsBinary`, `compatibility mode`, and a cache loader). Only data local to the node where the stream operation is performed will be used, for example `invalidation` only uses local entries.

10.2.5. Example

The code below takes a cache and returns a map with all the cache entries whose values contain the string "JBoss"

```
Map<Object, String> jbossValues = cache.entrySet().stream()
    .filter(e -> e.getValue().contains("JBoss"))
    .collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));
```

10.3. Distribution/Replication/Scattered

This is where streams come into their stride. When a stream operation is performed it will send the various intermediate and terminal operations to each node that has pertinent data. This allows processing the intermediate values on the nodes owning the data, and only sending the final results back to the originating nodes, improving performance.

10.3.1. Rehash Aware

Internally the data is segmented and each node only performs the operations upon the data it owns as a primary owner. This allows for data to be processed evenly, assuming segments are granular enough to provide for equal amounts of data on each node.

When you are utilizing a distributed cache, the data can be reshuffled between nodes when a new node joins or leaves. Distributed Streams handle this reshuffling of data automatically so you don't have to worry about monitoring when nodes leave or join the cluster. Reshuffled entries may be processed a second time, and we keep track of the processed entries at the key level or at the segment level (depending on the terminal operation) to limit the amount of duplicate processing.

It is possible but highly discouraged to disable rehash awareness on the stream. This should only be considered if your request can handle only seeing a subset of data if a rehash occurs. This can be done by invoking `CacheStream.disableRehashAware()`. The performance gain for most operations when a rehash doesn't occur is completely negligible. The only exceptions are for `iterator` and `forEach`, which will use less memory, since they do not have to keep track of processed keys.



Please rethink disabling rehash awareness unless you really know what you are doing.

10.3.2. Serialization

Since the operations are sent across to other nodes they must be serializable by `{brandname}` marshalling. This allows the operations to be sent to the other nodes.

The simplest way is to use a `CacheStream` instance and use a lambda just as you would normally. `{brandname}` overrides all of the various `Stream` intermediate and terminal methods to take `Serializable` versions of the arguments (ie. `SerializableFunction`, `SerializablePredicate`...) You can find these methods at `CacheStream`. This relies on the spec to pick the most specific method as defined [here](#).

In our previous example we used a `Collector` to collect all the results into a `Map`. Unfortunately the `Collectors` class doesn't produce `Serializable` instances. Thus if you need to use these, there are two ways to do so:

One option would be to use the `CacheCollectors` class which allows for a `Supplier<Collector>` to be provided. This instance could then use the `Collectors` to supply a `Collector` which is not serialized. You can read more details about how the collector performs in a distributed fashion at [distributed execution](#).

```
Map<Object, String> jbossValues = cache.entrySet().stream()
    .filter(e -> e.getValue().contains("Jboss"))
    .collect(CacheCollectors.serializableCollector(() -> Collectors.toMap
(Map.Entry::getKey, Map.Entry::getValue)));
```

Alternatively, you can avoid the use of `CacheCollectors` and instead use the overloaded `collect` methods that take `Supplier<Collector>`. These overloaded `collect` methods are only available via `CacheStream` interface.

```
Map<Object, String> jbossValues = cache.entrySet().stream()
    .filter(e -> e.getValue().contains("Jboss"))
    .collect(() -> Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue)
);
```

If however you are not able to use the `Cache` and `CacheStream` interfaces you cannot utilize `Serializable` arguments and you must instead cast the lambdas to be `Serializable` manually by casting the lambda to multiple interfaces. It is not a pretty sight but it gets the job done.

```
Map<Object, String> jbossValues = map.entrySet().stream()
    .filter((Serializable & Predicate<Map.Entry<Object, String>>) e -> e
.getValue().contains("Jboss"))
    .collect(CacheCollectors.serializableCollector(() -> Collectors.toMap
(Map.Entry::getKey, Map.Entry::getValue)));
```

The recommended and most performant way is to use an `AdvancedExternalizer` as this provides the smallest payload. Unfortunately this means you cannot use lambdas as advanced externalizers require defining the class before hand.

You can use an advanced externalizer as shown below:

```

Map<Object, String> jbossValues = cache.entrySet().stream()
    .filter(new ContainsFilter("Jboss"))
    .collect(() -> Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue)
);

class ContainsFilter implements Predicate<Map.Entry<Object, String>> {
    private final String target;

    ContainsFilter(String target) {
        this.target = target;
    }

    @Override
    public boolean test(Map.Entry<Object, String> e) {
        return e.getValue().contains(target);
    }
}

class JbossFilterExternalizer implements AdvancedExternalizer<ContainsFilter> {

    @Override
    public Set<Class<? extends ContainsFilter>> getTypeClasses() {
        return Util.asSet(ContainsFilter.class);
    }

    @Override
    public Integer getId() {
        return CUSTOM_ID;
    }

    @Override
    public void writeObject(ObjectOutput output, ContainsFilter object) throws
IOException {
        output.writeUTF(object.target);
    }

    @Override
    public ContainsFilter readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
        return new ContainsFilter(input.readUTF());
    }
}

```

You could also use an advanced externalizer for the collector supplier to reduce the payload size even further.


```

Map<Object, String> jbossValues = cache.entrySet().stream()
    .filter(new ContainsFilter("Jboss"))
    .collect(ToMapCollectorSupplier.INSTANCE);

class ToMapCollectorSupplier<K, U> implements Supplier<Collector<Map.Entry<K, U>, ?,
Map<K, U>>> {
    static final ToMapCollectorSupplier INSTANCE = new ToMapCollectorSupplier();

    private ToMapCollectorSupplier() { }

    @Override
    public Collector<Map.Entry<K, U>, ?, Map<K, U>> get() {
        return Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue);
    }
}

class ToMapCollectorSupplierExternalizer implements AdvancedExternalizer
<ToMapCollectorSupplier> {

    @Override
    public Set<Class<? extends ToMapCollectorSupplier>> getTypeClasses() {
        return Util.asSet(ToMapCollectorSupplier.class);
    }

    @Override
    public Integer getId() {
        return CUSTOM_ID;
    }

    @Override
    public void writeObject(ObjectOutput output, ToMapCollectorSupplier object)
throws IOException {
    }

    @Override
    public ToMapCollectorSupplier readObject(ObjectInput input) throws IOException,
ClassNotFoundException {
        return ToMapCollectorSupplier.INSTANCE;
    }
}

```

10.3.3. Parallel Computation

Distributed streams by default try to parallelize as much as possible. It is possible for the end user to control this and actually they always have to control one of the options. There are 2 ways these streams are parallelized.

Local to each node When a stream is created from the cache collection the end user can choose between invoking `stream` or `parallelStream` method. Depending on if the parallel stream was

picked will enable multiple threading for each node locally. Note that some operations like a rehash aware iterator and `forEach` operations will always use a sequential stream locally. This could be enhanced at some point to allow for parallel streams locally.

Users should be careful when using local parallelism as it requires having a large number of entries or operations that are computationally expensive to be faster. Also it should be noted that if a user uses a parallel stream with `forEach` that the action should not block as this would be executed on the common pool, which is normally reserved for computation operations.

Remote requests When there are multiple nodes it may be desirable to control whether the remote requests are all processed at the same time concurrently or one at a time. By default all terminal operations except the iterator perform concurrent requests. The iterator, method to reduce overall memory pressure on the local node, only performs sequential requests which actually performs slightly better.

If a user wishes to change this default however they can do so by invoking the `sequentialDistribution` or `parallelDistribution` methods on the `CacheStream`.

10.3.4. Task timeout

It is possible to set a timeout value for the operation requests. This timeout is used only for remote requests timing out and it is on a per request basis. The former means the local execution will not timeout and the latter means if you have a failover scenario as described above the subsequent requests each have a new timeout. If no timeout is specified it uses the replication timeout as a default timeout. You can set the timeout in your task by doing the following:

```
CacheStream<Object, String> stream = cache.entrySet().stream();
stream.timeout(1, TimeUnit.MINUTES);
```

For more information about this, please check the java doc in `timeout` javadoc.

10.3.5. Injection

The `Stream` has a terminal operation called `forEach` which allows for running some sort of side effect operation on the data. In this case it may be desirable to get a reference to the `Cache` that is backing this `Stream`. If your `Consumer` implements the `CacheAware` interface the `injectCache` method be invoked before the `accept` method from the `Consumer` interface.

10.3.6. Distributed Stream execution

Distributed streams execution works in a fashion very similar to map reduce. Except in this case we are sending zero to many intermediate operations (map, filter etc.) and a single terminal operation to the various nodes. The operation basically comes down to the following:

1. The desired segments are grouped by which node is the primary owner of the given segment
2. A request is generated to send to each remote node that contains the intermediate and terminal operations including which segments it should process

- a. The terminal operation will be performed locally if necessary
 - b. Each remote node will receive this request and run the operations and subsequently send the response back
3. The local node will then gather the local response and remote responses together performing any kind of reduction required by the operations themselves.
 4. Final reduced response is then returned to the user

In most cases all operations are fully distributed, as in the operations are all fully applied on each remote node and usually only the last operation or something related may be reapplied to reduce the results from multiple nodes. One important note is that intermediate values do not actually have to be serializable, it is the last value sent back that is the part desired (exceptions for various operations will be highlighted below).

Terminal operator distributed result reductions The following paragraphs describe how the distributed reductions work for the various terminal operators. Some of these are special in that an intermediate value may be required to be serializable instead of the final result.

allMatch noneMatch anyMatch

The [allMatch](#) operation is ran on each node and then all the results are logically anded together locally to get the appropriate value. The [noneMatch](#) and [anyMatch](#) operations use a logical or instead. These methods also have early termination support, stopping remote and local operations once the final result is known.

collect

The [collect](#) method is interesting in that it can do a few extra steps. The remote node performs everything as normal except it doesn't perform the final [finisher](#) upon the result and instead sends back the fully combined results. The local thread then [combines](#) the remote and local result into a value which is then finally finished. The key here to remember is that the final value doesn't have to be serializable but rather the values produced from the [supplier](#) and [combiner](#) methods.

count

The [count](#) method just adds the numbers together from each node.

findAny findFirst

The [findAny](#) operation returns just the first value they find, whether it was from a remote node or locally. Note this supports early termination in that once a value is found it will not process others. Note the [findFirst](#) method is special since it requires a sorted intermediate operation, which is detailed in the [exceptions](#) section.

max min

The [max](#) and [min](#) methods find the respective min or max value on each node then a final reduction is performed locally to ensure only the min or max across all nodes is returned.

reduce

The various reduce methods [1](#) , [2](#) , [3](#) will end up serializing the result as much as the accumulator can do. Then it will accumulate the local and remote results together locally, before

combining if you have provided that. Note this means a value coming from the combiner doesn't have to be Serializable.

10.3.7. Key based rehash aware operators

The `iterator`, `spliterator` and `forEach` are unlike the other terminal operators in that the rehash awareness has to keep track of what keys per segment have been processed instead of just segments. This is to guarantee an exactly once (`iterator` & `spliterator`) or at least once behavior (`forEach`) even under cluster membership changes.

The `iterator` and `spliterator` operators when invoked on a remote node will return back batches of entries, where the next batch is only sent back after the last has been fully consumed. This batching is done to limit how many entries are in memory at a given time. The user node will hold onto which keys it has processed and when a given segment is completed it will release those keys from memory. This is why sequential processing is preferred for the `iterator` method, so only a subset of segment keys are held in memory at once, instead of from all nodes.

The `forEach()` method also returns batches, but it returns a batch of keys after it has finished processing at least a batch worth of keys. This way the originating node can know what keys have been processed already to reduce chances of processing the same entry again. Unfortunately this means it is possible to have an at least once behavior when a node goes down unexpectedly. In this case that node could have been processing a batch and not yet completed one and those entries that were processed but not in a completed batch will be ran again when the rehash failure operation occurs. Note that adding a node will not cause this issue as the rehash failover doesn't occur until all responses are received.

These operations batch sizes are both controlled by the same value which can be configured by invoking `distributedBatchSize` method on the `CacheStream`. This value will default to the `chunkSize` configured in state transfer. Unfortunately this value is a tradeoff with memory usage vs performance vs at least once and your mileage may vary.

Using `iterator` with replicated and distributed caches

When a node is the primary or backup owner of all requested segments for a distributed stream, {brandname} performs the `iterator` or `spliterator` terminal operations locally, which optimizes performance as remote iterations are more resource intensive.

This optimization applies to both replicated and distributed caches. However, {brandname} performs iterations remotely when using cache stores that are both `shared` and have `write-behind` enabled. In this case performing the iterations remotely ensures consistency.

10.3.8. Intermediate operation exceptions

There are some intermediate operations that have special exceptions, these are `skip`, `peek`, sorted 1 2. & `distinct`. All of these methods have some sort of artificial iterator implanted in the stream processing to guarantee correctness, they are documented as below. Note this means these operations may cause possibly severe performance degradation.

Skip

An artificial iterator is implanted up to the intermediate skip operation. Then results are brought locally so it can skip the appropriate amount of elements.

Sorted

WARNING: This operation requires having all entries in memory on the local node. An artificial iterator is implanted up to the intermediate sorted operation. All results are sorted locally. There are possible plans to have a distributed sort which returns batches of elements, but this is not yet implemented.

Distinct

WARNING: This operation requires having all or nearly all entries in memory on the local node. Distinct is performed on each remote node and then an artificial iterator returns those distinct values. Then finally all of those results have a distinct operation performed upon them.

The rest of the intermediate operations are fully distributed as one would expect.

10.3.9. Examples

Word Count

Word count is a classic, if overused, example of map/reduce paradigm. Assume we have a mapping of key \rightarrow sentence stored on {brandname} nodes. Key is a String, each sentence is also a String, and we have to count occurrence of all words in all sentences available. The implementation of such a distributed task could be defined as follows:

```

public class WordCountExample {

    /**
     * In this example replace c1 and c2 with
     * real Cache references
     *
     * @param args
     */
    public static void main(String[] args) {
        Cache<String, String> c1 = ...;
        Cache<String, String> c2 = ...;

        c1.put("1", "Hello world here I am");
        c2.put("2", "Infinispan rules the world");
        c1.put("3", "JUDCon is in Boston");
        c2.put("4", "JBoss World is in Boston as well");
        c1.put("12", "JBoss Application Server");
        c2.put("15", "Hello world");
        c1.put("14", "Infinispan community");
        c2.put("15", "Hello world");

        c1.put("111", "Infinispan open source");
        c2.put("112", "Boston is close to Toronto");
        c1.put("113", "Toronto is a capital of Ontario");
        c2.put("114", "JUDCon is cool");
        c1.put("211", "JBoss World is awesome");
        c2.put("212", "JBoss rules");
        c1.put("213", "JBoss division of RedHat ");
        c2.put("214", "RedHat community");

        Map<String, Long> wordCountMap = c1.entrySet().parallelStream()
            .map(e -> e.getValue().split("\\s"))
            .flatMap(Arrays::stream)
            .collect(() -> Collectors.groupingBy(Function.identity(), Collectors.
counting()));
    }
}

```

In this case it is pretty simple to do the word count from the previous example.

However what if we want to find the most frequent word in the example? If you take a second to think about this case you will realize you need to have all words counted and available locally first. Thus we actually have a few options.

We could use a finisher on the collector, which is invoked on the user thread after all the results have been collected. Some redundant lines have been removed from the previous example.

```

public class WordCountExample {
    public static void main(String[] args) {
        // Lines removed

        String mostFrequentWord = c1.entrySet().parallelStream()
            .map(e -> e.getValue().split("\\s"))
            .flatMap(Arrays::stream)
            .collect(() -> Collectors.collectingAndThen(
                Collectors.groupingBy(Function.identity(), Collectors.counting()),
                wordCountMap -> {
                    String mostFrequent = null;
                    long maxCount = 0;
                    for (Map.Entry<String, Long> e : wordCountMap.entrySet()) {
                        int count = e.getValue().intValue();
                        if (count > maxCount) {
                            maxCount = count;
                            mostFrequent = e.getKey();
                        }
                    }
                    return mostFrequent;
                }
            ));
    }
}

```

Unfortunately the last step is only going to be ran in a single thread, which if we have a lot of words could be quite slow. Maybe there is another way to parallelize this with Streams.

We mentioned before we are in the local node after processing, so we could actually use a stream on the map results. We can therefore use a parallel stream on the results.

```

public class WordFrequencyExample {
    public static void main(String[] args) {
        // Lines removed

        Map<String, Long> wordCount = c1.entrySet().parallelStream()
            .map(e -> e.getValue().split("\\s"))
            .flatMap(Arrays::stream)
            .collect(() -> Collectors.groupingBy(Function.identity(), Collectors
                .counting()));
        Optional<Map.Entry<String, Long>> mostFrequent = wordCount.entrySet()
            .parallelStream().reduce(
                (e1, e2) -> e1.getValue() > e2.getValue() ? e1 : e2);
    }
}

```

This way you can still utilize all of the cores locally when calculating the most frequent element.

Remove specific entries

Distributed streams can also be used as a way to modify data where it lives. For example you may want to remove all entries in your cache that contain a specific word.

```
public class RemoveBadWords {
    public static void main(String[] args) {
        // Lines removed
        String word = ..

        c1.entrySet().parallelStream()
            .filter(e -> e.getValue().contains(word))
            .forEach((c, e) -> c.remove(e.getKey()));
    }
}
```

If we carefully note what is serialized and what is not, we notice that only the word along with the operations are serialized across to other nodes as it is captured by the lambda. However the real saving piece is that the cache operation is performed on the primary owner thus reducing the amount of network traffic required to remove these values from the cache. The cache is not captured by the lambda as we provide a special `BiConsumer` method override that when invoked on each node passes the cache to the `BiConsumer`

One thing to keep in mind using the `forEach` command in this manner is that the underlying stream obtains no locks. The cache remove operation will still obtain locks naturally, but the value could have changed from what the stream saw. That means that the entry could have been changed after the stream read it but the remove actually removed it.

We have specifically added a new variant which is called `LockedStream`.

Plenty of other examples

The `Streams` API is a JRE tool and there are lots of examples for using it. Just remember that your operations need to be `Serializable` in some way.

10.4. Locked Streams

TODO: need to detail Locked Streams

Chapter 11. Indexing and Querying

11.1. Overview

{brandname} supports indexing and searching of Java Pojo(s) or objects encoded via [Protocol Buffers](#) stored in the grid using powerful search APIs which complement its main Map-like API.

Querying is possible both in [library](#) and [client/server mode](#) (for Java, C#, Node.js and other clients), and {brandname} can index data using [Apache Lucene](#), offering an efficient [full-text](#) capable search engine in order to cover a wide range of data retrieval use cases.

Indexing configuration relies on a schema definition, and for that {brandname} can use annotated Java classes when in library mode, and protobuf schemas for remote clients written in other languages. By standardizing on protobuf, {brandname} allows full interoperability between Java and non-Java clients.

Apart from indexed queries, {brandname} can run queries over non-indexed data ([indexless queries](#)) and over partially indexed data ([hybrid queries](#)).

In terms of Search APIs, {brandname} has its own query language called [Ickle](#), which is a subset of JP-QL providing extensions for full-text querying. The [Query DSL](#) can be used for both embedded and remote java clients when full-text is not required; for Java embedded clients {brandname} offers the [Hibernate Search Query API](#) which supports running Lucene queries in the grid, apart from advanced search capabilities like Faceted and Spatial search.

Finally, {brandname} has support for [Continuous Queries](#), which works in a reverse manner to the other APIs: instead of creating, executing a query and obtain results, it allows a client to register queries that will be evaluated continuously as data in the cluster changes, generating notifications whenever the changed data matches the queries.

11.2. Embedded Querying

Embedded querying is available when {brandname} is used as a library. No protobuf mapping is required, and both indexing and searching are done on top of Java objects. When in library mode, it is possible to run Lucene queries directly and use all the available [Query APIs](#) and it also allows flexible indexing configurations to keep latency to a minimal.

11.2.1. Quick example

We're going to store *Book* instances in an {brandname} cache called "books". *Book* instances will be indexed, so we enable indexing for the cache, letting {brandname} [configure the indexing automatically](#):

{brandname} configuration:

infinispan.xml

```
<infinispan>
  <cache-container>
    <transport cluster="infinispan-cluster"/>
    <distributed-cache name="books">
      <indexing index="LOCAL" auto-config="true"/>
    </distributed-cache>
  </cache-container>
</infinispan>
```

Obtaining the cache:

```
import org.infinispan.Cache;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.manager.EmbeddedCacheManager;

EmbeddedCacheManager manager = new DefaultCacheManager("infinispan.xml");
Cache<String, Book> cache = manager.getCache("books");
```

Each *Book* will be defined as in the following example; we have to choose which properties are indexed, and for each property we can optionally choose advanced indexing options using the annotations defined in the Hibernate Search project.

Book.java

```
import org.hibernate.search.annotations.*;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;

//Values you want to index need to be annotated with @Indexed, then you pick which
//fields and how they are to be indexed:
@Indexed
public class Book {
  @Field String title;
  @Field String description;
  @Field @DateBridge(resolution=Resolution.YEAR) Date publicationYear;
  @IndexedEmbedded Set<Author> authors = new HashSet<Author>();
}
```

Author.java

```
public class Author {
  @Field String name;
  @Field String surname;
  // hashCode() and equals() omitted
}
```

Now assuming we stored several *Book* instances in our {brandname} *Cache* , we can search them for any matching field as in the following example.

Using a Lucene Query:

```
// get the search manager from the cache:
SearchManager searchManager = org.infinispan.query.Search.getSearchManager(cache);

// create any standard Lucene query, via Lucene's QueryParser or any other means:
org.apache.lucene.search.Query fullTextQuery = //any Apache Lucene Query

// convert the Lucene query to a CacheQuery:
CacheQuery cacheQuery = searchManager.getQuery( fullTextQuery );

// get the results:
List<Object> found = cacheQuery.list();
```

A Lucene Query is often created by parsing a query in text format such as "title:infinispan AND authors.name:sanne", or by using the query builder provided by Hibernate Search.

```
// get the search manager from the cache:
SearchManager searchManager = org.infinispan.query.Search.getSearchManager( cache );

// you could make the queries via Lucene APIs, or use some helpers:
QueryBuilder queryBuilder = searchManager.buildQueryBuilderForClass(Book.class).get();

// the queryBuilder has a nice fluent API which guides you through all options.
// this has some knowledge about your object, for example which Analyzers
// need to be applied, but the output is a fairly standard Lucene Query.
org.apache.lucene.search.Query luceneQuery = queryBuilder.phrase()
    .onField("description")
    .andField("title")
    .sentence("a book on highly scalable query engines")
    .createQuery();

// the query API itself accepts any Lucene Query, and on top of that
// you can restrict the result to selected class types:
CacheQuery query = searchManager.getQuery(luceneQuery, Book.class);

// and there are your results!
List objectList = query.list();

for (Object book : objectList) {
    System.out.println(book);
}
```

Apart from *list()* you have the option for streaming results, or use pagination.

For searches that do not require Lucene or full-text capabilities and are mostly about aggregation

and exact matches, we can use the {brandname} Query DSL API:

```
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;
import org.infinispan.query.Search;

// get the query factory:
QueryFactory queryFactory = Search.getQueryFactory(cache);

Query q = queryFactory.from(Book.class)
    .having("author.surname").eq("King")
    .build();

List<Book> list = q.list();
```

Finally, we can use an [Ickle](#) query directly, allowing for Lucene syntax in one or more predicates:

```
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

// get the query factory:
QueryFactory queryFactory = Search.getQueryFactory(cache);

Query q = queryFactory.create("from Book b where b.author.name = 'Stephen' and " +
    "b.description : ('dark' - 'tower')");

List<Book> list = q.list();
```

11.2.2. Indexing

Indexing in {brandname} happens on a per-cache basis and by default a cache is not indexed. Enabling indexing is not mandatory but queries using an index will have a vastly superior performance. On the other hand, enabling indexing can impact negatively the write throughput of a cluster, so make sure to check the [query performance guide](#) for some strategies to minimize this impact depending on the cache type and use case.

Configuration

General format

To enable indexing via XML, you need to add the `<indexing>` element plus the `index` ([index mode](#)) to your cache configuration, and optionally pass additional properties.

```

<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="default">
      <indexing index="ALL">
        <property name="property.name">some value</property>
      </indexing>
    </replicated-cache>
  </cache-container>
</infinispan>

```

Programmatic:

```

import org.infinispan.configuration.cache.*;

ConfigurationBuilder cacheCfg = ...
cacheCfg.indexing().index(Index.ALL)
    .addProperty("property name", "property value")

```

Index names

Each property inside the `index` element is prefixed with the index name, for the index named `org.infinispan.sample.Car` the `directory_provider` is `local-heap`:

```

...
<indexing index="ALL">
  <property name="org.infinispan.sample.Car.directory_provider">local-
heap</property>
</indexing>
...
</infinispan>

```

```

cacheCfg.indexing()
    .index(Index.ALL)
    .addProperty("org.infinispan.sample.Car.directory_provider", "local-heap")

```

{brandname} creates an index for each entity existent in a cache, and it allows to configure those indexes independently. For a class annotated with `@Indexed`, the index name is the fully qualified class name, unless overridden with the `name` argument in the annotation.

In the snippet below, the default storage for all entities is `infinispan`, but `Boat` instances will be stored on `local-heap` in an index named `boatIndex`. `Airplane` entities will also be stored in `local-heap`. Any other entity's index will be configured with the property prefixed by `default`.

```

package org.infinispan.sample;

@Indexed(name = "boatIndex")
public class Boat {

}

@Indexed
public class Airplane {

}

```

```

...
<indexing index="ALL">
  <property name="default.directory_provider">infinispan</property>
  <property name="boatIndex.directory_provider">local-heap</property>
  <property name="org.infinispan.sample.Airplane.directory_provider">
    ram
  </property>
</indexing>
...
</infinispan>

```

Specifying indexed Entities

{brandname} can automatically recognize and manage indexes for different entity types in a cache. Future versions of {brandname} will remove this capability so it's recommended to declare upfront which types are going to be indexed (list them by their fully qualified class name). This can be done via xml:

```

<infinispan>
  <cache-container default-cache="default">
    <replicated-cache name="default">
      <indexing index="ALL">
        <indexed-entities>
          <indexed-entity>com.acme.query.test.Car</indexed-entity>
          <indexed-entity>com.acme.query.test.Truck</indexed-entity>
        </indexed-entities>
      </indexing>
    </replicated-cache>
  </cache-container>
</infinispan>

```

or programmatically:

```
cacheCfg.indexing()
    .index(Index.ALL)
    .addIndexedEntity(Car.class)
    .addIndexedEntity(Truck.class)
```

In server mode, the class names listed under the 'indexed-entities' element must use the 'extended' class name format which is composed of a JBoss Modules module identifier, a slot name, and the fully qualified class name, these three components being separated by the ':' character, (eg. "com.acme.my-module-with-entity-classes:my-slot:com.acme.query.test.Car"). The entity classes must be located in the referenced module, which can be either a user supplied module deployed in the 'modules' folder of your server or a plain jar deployed in the 'deployments' folder. The module in question will become an automatic dependency of your Cache, so its eventual redeployment will cause the cache to be restarted.



Only for server, if you fail to follow the requirement of using 'extended' class names and use a plain class name its resolution will fail due to missing class because the wrong ClassLoader is being used (the {brandname}'s internal class path is being used).

Index mode

An {brandname} node typically receives data from two sources: local and remote. Local translates to clients manipulating data using the map API in the same JVM; remote data comes from other {brandname} nodes during replication or rebalancing.

The index mode configuration defines, from a node in the cluster point of view, which data gets indexed.

Possible values:

- ALL: all data is indexed, local and remote.
- LOCAL: only local data is indexed.
- PRIMARY_OWNER: Only entries containing keys that the node is primary owner will be indexed, regardless of local or remote origin.
- NONE: no data is indexed. Equivalent to not configure indexing at all.

Index Managers

Index managers are central components in {brandname} Querying responsible for the indexing configuration, distribution and internal lifecycle of several query components such as Lucene's *IndexReader* and *IndexWriter*. Each Index Manager is associated with a *Directory Provider*, which defines the physical storage of the index.

Regarding index distribution, {brandname} can be configured with shared or non-shared indexes.

Shared indexes

A shared index is a single, distributed, cluster-wide index for a certain cache. The main advantage is that the index is visible from every node and can be queried as if the index were local, there is no need to [broadcast](#) queries to all members and aggregate the results. The downside is that Lucene does not allow more than a single process writing to the index at the same time, and the coordination of lock acquisitions needs to be done by a proper shared index capable index manager. In any case, having a single write lock cluster-wise can lead to some degree of contention under heavy writing.

{brandname} supports shared indexes leveraging the [{brandname} Directory Provider](#), which stores indexes in a separate set of caches. Two index managers are available to use shared indexes: `InfinispanIndexManager` and `AffinityIndexManager`.

Effect of the index mode

Shared indexes should not use the **ALL** index mode since it'd lead to redundant indexing: since there is a single index cluster wide, the entry would get indexed when inserted via Cache API, and another time when {brandname} replicates it to another node. The **ALL** mode is usually associates with [non-shared indexes](#) in order to create full index replicas on each node.

`InfinispanIndexManager`

This index manager uses the [{brandname} Directory Provider](#), and is suitable for creating shared indexes. Index mode should be set to **LOCAL** in this configuration.

Configuration:


```

<distributed-cache name="default" >
  <indexing index="LOCAL">
    <property name="default.indexmanager">
      org.infinispan.query.indexmanager.InfinispanIndexManager
    </property>
    <!-- optional: tailor each index cache -->
    <property name="default.locking_cachename">
      LuceneIndexesLocking_custom</property>
    <property name="default.data_cachename">LuceneIndexesData_custom</property>
    <property name="default.metadata_cachename">
      LuceneIndexesMetadata_custom</property>
    </indexing>
  </distributed-cache>

  <!-- Optional -->
  <replicated-cache name="LuceneIndexesLocking_custom">
    <indexing index="NONE" />
    <-- extra configuration -->
  </replicated-cache>

  <!-- Optional -->
  <replicated-cache name="LuceneIndexesMetadata_custom">
    <indexing index="NONE" />
    <-- extra configuration -->
  </replicated-cache>

  <!-- Optional -->
  <distributed-cache name="LuceneIndexesData_custom">
    <-- extra configuration -->
    <indexing index="NONE" />
  </distributed-cache>

```

Indexes are stored in a set of clustered caches, called by default *LuceneIndexesData*, *LuceneIndexesMetadata* and *LuceneIndexesLocking*.

The *LuceneIndexesLocking* cache is used to store Lucene locks, and it is a very small cache: it will contain one entry per entity (index).

The *LuceneIndexesMetadata* cache is used to store info about the logical files that are part of the index, such as names, chunks and sizes and it is also small in size.

The *LuceneIndexesData* cache is where most of the index is located: it is much bigger than the other two but should be smaller than the data in the cache itself, thanks to Lucene's efficient storing techniques.

It's not necessary to redefine the configuration of those 3 cases, {brandname} will pick sensible defaults. Reasons re-define them would be performance tuning for a specific scenario, or for example to make them persistent by configuring a cache store.

In order to avoid index corruption when two or more nodes of the cluster try to write to the index

at the same time, the *InfinispanIndexManager* internally elects a master in the cluster (which is the JGroups coordinator) and forwards all indexing works to this master.

AffinityIndexManager

The *AffinityIndexManager* is an **experimental** index manager used for shared indexes that also stores indexes using the [{brandname} Directory Provider](#). Unlike the *InfinispanIndexManager*, it does not have a single node (master) that handles all the indexing cluster wide, but rather splits the index using multiple shards, each shard being responsible for indexing data associated with one or more {brandname} segments. For an in-depth description of the inner workings, please see the [design doc](#).

The PRIMARY_OWNER index mode is required, together with a special kind of *KeyPartitioner*.

XML Configuration:

```
<distributed-cache name="default"
                  key-partitioner=
"org.infinispan.distribution.ch.impl.AffinityPartitioner">
  <indexing index="PRIMARY_OWNER">
    <property name="default.indexmanager">
      org.infinispan.query.affinity.AffinityIndexManager
    </property>
    <!-- optional: control the number of shards, the default is 4 -->
    <property name="default.sharding_strategy.nbr_of_shards">10</property>
  </indexing>
</distributed-cache>
```

Programmatic:

```
import org.infinispan.distribution.ch.impl.AffinityPartitioner;
import org.infinispan.query.affinity.AffinityIndexManager;

ConfigurationBuilder cacheCfg = ...
cacheCfg.clustering().hash().keyPartitioner(new AffinityPartitioner());
cacheCfg.indexing()
    .index(Index.PRIMARY_OWNER)
    .addProperty("default.indexmanager", AffinityIndexManager.class.getName())
    .addProperty("default.sharding_strategy.nbr_of_shards", "10")
```

The *AffinityIndexManager* by default will have as many shards as {brandname} segments, but this value is configurable as seen in the example above.

The number of shards affects directly the query performance and writing throughput: generally speaking, a high number of shards offers better write throughput but has an adverse effect on query performance.

Non-shared indexes

Non-shared indexes are independent indexes at each node. This setup is particularly advantageous for replicated caches where each node has all the cluster data and thus can hold all the indexes as well, offering optimal query performance with zero network latency when querying. Another advantage is, since the index is local to each node, there is less contention during writes due to the fact that each node is subjected to its own index lock, not a cluster wide one.

Since each node might hold a partial index, it may be necessary to link#query_clustered_query_api[broadcast] queries in order to get correct search results, which can add latency. If the cache is REPL, though, the broadcast is not necessary: each node can hold a full local copy of the index and queries runs at optimal speed taking advantage of a local index.

{brandname} has two index managers suitable for non-shared indexes: **directory-based** and **near-real-time**. Storage wise, non-shared indexes can be located in ram, filesystem, or {brandname} local caches.

Effect of the index mode

The **directory-based** and **near-real-time** index managers can be associated with different **index modes**, resulting in different index distributions.

REPL caches combined with the **ALL** index mode will result in a full copy of the cluster-wide index on each node. This mode allows queries to become effectively local without network latency. This is the recommended mode to index any REPL cache, and that's the choice picked by the **auto-config** when the a REPL cache is detected. The **ALL** mode should not be used with DIST caches.

REPL or DIST caches combined with **LOCAL** index mode will cause each node to index only data inserted from the same JVM, causing an uneven distribution of the index. In order to obtain correct query results, it's necessary to use **broadcast** queries.

REPL or DIST caches combined with **PRIMARY_OWNER** will also need broadcast queries. Differently from the **LOCAL** mode, each node's index will contain indexed entries which key is primarily owned by the node according to the consistent hash, leading to a more evenly distributed indexes among the nodes.

directory-based index manager

This is the default Index Manager used when no index manager is configured. The **directory-based** index manager is used to manage indexes backed by a local lucene directory. It supports *ram*, *filesystem* and non-clustered *infinispan* storage.

Filesystem storage

This is the default storage, and used when index manager configuration is omitted. The index is stored in the filesystem using a **MMapDirectory**. It is the recommended storage for local indexes. Although indexes are persistent on disk, they get memory mapped by Lucene and thus offer decent query performance.

Configuration:

```
<replicated-cache name="myCache">
  <indexing index="ALL">
    <!-- Optional: define base folder for indexes -->
    <property name="default.indexBase">${java.io.tmpdir}/baseDir</property>
  </indexing>
</replicated-cache>
```

{brandname} will create a different folder under `default.indexBase` for each entity (index) present in the cache.

Ram storage

Index is stored in memory using a [Lucene RAMDirectory](#). Not recommended for large indexes or highly concurrent situations. Indexes stored in Ram are not persistent, so after a cluster shutdown a [re-index](#) is needed. Configuration:

```
<replicated-cache name="myCache">
  <indexing index="ALL">
    <property name="default.directory_provider">local-heap</property>
  </indexing>
</replicated-cache>
```

{brandname} storage

{brandname} storage makes use of the {brandname} [Lucene directory](#) that saves the indexes to a set of caches; those caches can be configured like any other {brandname} cache, for example by adding a cache store to have indexes persisted elsewhere apart from memory. In order to use {brandname} storage with a non-shared index, it's necessary to use LOCAL caches for the indexes:

```

<replicated-cache name="default">
  <indexing index="ALL">
    <property name="default.locking_cachename">
LuceneIndexesLocking_custom</property>
    <property name="default.data_cachename">LuceneIndexesData_custom</property>
    <property name="default.metadata_cachename">
LuceneIndexesMetadata_custom</property>
  </indexing>
</replicated-cache>

<local-cache name="LuceneIndexesLocking_custom">
  <indexing index="NONE" />
</local-cache>

<local-cache name="LuceneIndexesMetadata_custom">
  <indexing index="NONE" />
</local-cache>

<local-cache name="LuceneIndexesData_custom">
  <indexing index="NONE" />
</local-cache>

```

near-real-time index manager

Similar to the **directory-based** index manager but takes advantage of the Near-Real-Time features of Lucene. It has better write performance than the **directory-based** because it flushes the index to the underlying store less often. The drawback is that unflushed index changes can be lost in case of a non-clean shutdown. Can be used in conjunction with **local-heap**, **filesystem** and local infinispn storage. Configuration for each different storage type is the same as the **directory-based** index manager.

Example with ram:

```

<replicated-cache name="default">
  <indexing index="ALL">
    <property name="default.indexmanager">near-real-time</property>
    <property name="default.directory_provider">local-heap</property>
  </indexing>
</replicated-cache>

```

Example with filesystem:

```

<replicated-cache name="default">
  <indexing index="ALL">
    <property name="default.indexmanager">near-real-time</property>
  </indexing>
</replicated-cache>

```

External indexes

Apart from having shared and non-shared indexes managed by {brandname} itself, it is possible to offload indexing to a third party search engine: currently {brandname} supports Elasticsearch as an external index storage.

Elasticsearch IndexManager (experimental)

This index manager forwards all indexes to an external Elasticsearch server. This is an experimental integration and some features may not be available, for example `indexNullAs` for `@IndexedEmbedded` annotations is **not currently supported**.

Configuration:

```
<indexing index="LOCAL">
  <property name="default.indexmanager">elasticsearch</property>
  <property name="default.elasticsearch.host">
link:http://elasticHost:9200</property>
  <!-- other elasticsearch configurations -->
</indexing>
```

The index mode should be set to `LOCAL`, since {brandname} considers Elasticsearch as a single shared index. More information about Elasticsearch integration, including the full description of the configuration properties can be found at the [Hibernate Search manual](#).

Automatic configuration

The attribute `auto-config` provides a simple way of configuring indexing based on the cache type. For replicated and local caches, the indexing is configured to be persisted on disk and not shared with any other processes. Also, it is configured so that minimum delay exists between the moment an object is indexed and the moment it is available for searches (near real time).

```
<local-cache name="default">
  <indexing index="LOCAL" auto-config="true">
  </indexing>
</local-cache>
```



it is possible to redefine any property added via `auto-config`, and also add new properties, allowing for advanced tuning.

The auto config adds the following properties for replicated and local caches:

Property name	value	description
default.directory_provider	filesystem	Filesystem based index. More details at Hibernate Search documentation

Property name	value	description
default.exclusive_index_use	true	indexing operation in exclusive mode, allowing Hibernate Search to optimize writes
default.indexmanager	near-real-time	make use of Lucene near real time feature, meaning indexed objects are promptly available to searches
default.reader.strategy	shared	Reuse index reader across several queries, thus avoiding reopening it

For distributed caches, the auto-config configure indexes in {brandname} itself, internally handled as a master-slave mechanism where indexing operations are sent to a single node which is responsible to write to the index.

The auto config properties for distributed caches are:

Property name	value	description
default.directory_provider	infinispan	Indexes stored in {brandname}. More details at Hibernate Search documentation
default.exclusive_index_use	true	indexing operation in exclusive mode, allowing Hibernate Search to optimize writes
default.indexmanager	org.infinispan.query.indexmanager.InfinispanIndexManager	Delegates index writing to a single node in the {brandname} cluster
default.reader.strategy	shared	Reuse index reader across several queries, avoiding reopening it

Re-indexing

Occasionally you might need to rebuild the Lucene index by reconstructing it from the data stored in the Cache. You need to rebuild the index if you change the definition of what is indexed on your types, or if you change for example some *Analyzer* parameter, as Analyzers affect how the index is written. Also, you might need to rebuild the index if you had it destroyed by some system administration mistake. To rebuild the index just get a reference to the MassIndexer and start it; beware it might take some time as it needs to reprocess all data in the grid!

```
// Blocking execution
SearchManager searchManager = Search.getSearchManager(cache);
searchManager.getMassIndexer().start();

// Non blocking execution
CompletableFuture<Void> future = searchManager.getMassIndexer().startAsync();
```



This is also available as a `start` JMX operation on the [MassIndexer MBean](#) registered under the name `org.infinispan:type=Query,manager="{name-of-cache-manager}",cache="{name-of-cache}",component=MassIndexer`.

Indexless

TODO

Hybrid

TODO

Mapping Entities

{brandname} relies on the rich API of [Hibernate Search](#) in order to define fine grained configuration for indexing at entity level. This configuration includes which fields are annotated, which analyzers should be used, how to map nested objects and so on. Detailed documentation is available at [the Hibernate Search manual](#).

@DocumentId

Unlike Hibernate Search, using `@DocumentId` to mark a field as identifier does not apply to {brandname} values; in {brandname} the identifier for all `@Indexed` objects is the key used to store the value. You can still customize how the key is indexed using a combination of `@Transformable`, custom types and custom `FieldBridge` implementations.

@Transformable keys

The key for each value needs to be indexed as well, and the key instance must be transformed in a *String*. {brandname} includes some default transformation routines to encode common primitives, but to use a custom key you must provide an implementation of `org.infinispan.query.Transformer`.

Registering a key Transformer via annotations

You can annotate your key class with `org.infinispan.query.Transformable` and your custom transformer implementation will be picked up automatically:


```

@Transformable(transformer = CustomTransformer.class)
public class CustomKey {
    ...
}

public class CustomTransformer implements Transformer {
    @Override
    public Object fromString(String s) {
        ...
        return new CustomKey(...);
    }

    @Override
    public String toString(Object customType) {
        CustomKey ck = (CustomKey) customType;
        return ...
    }
}

```

Registering a key Transformer via the cache indexing configuration

You can use the *key-transformers* xml element in both embedded and server config:

```

<replicated-cache name="test">
    <indexing index="ALL" auto-config="true">
        <key-transformers>
            <key-transformer key="com.mycompany.CustomKey" transformer=
"com.mycompany.CustomTransformer"/>
        </key-transformers>
    </indexing>
</replicated-cache>

```

or alternatively, you can achieve the same effect by using the Java configuration API (embedded mode):

```

ConfigurationBuilder builder = ...
builder.indexing().autoConfig(true)
    .addKeyTransformer(CustomKey.class, CustomTransformer.class);

```

Registering a Transformer programmatically at runtime

Using this technique, you don't have to annotate your custom key type and you also do not add the transformer to the, cache indexing configuration, instead, you can add it to the *SearchManagerImplementor* dynamically at runtime by invoking *org.infinispan.query.spi.SearchManagerImplementor.registerKeyTransformer(Class<?>, Class<? extends Transformer>)*:

```
org.infinispan.query.spi.SearchManagerImplementor manager = Search.getSearchManager  
(cache).unwrap(SearchManagerImplementor.class);  
manager.registerKeyTransformer(keyClass, keyTransformerClass);
```



This approach is deprecated since 10.0 because it can lead to situations when a newly started node receives cache entries via initial state transfer and is not able to index them because the needed key transformers are not yet registered (and can only be registered after the Cache has been fully started). This undesirable situation is avoided if you register your key transformers using the other available approaches (configuration and annotation).

Programmatic mapping

Instead of using annotations to map an entity to the index, it's also possible to configure it programmatically.

In the following example we map an object *Author* which is to be stored in the grid and made searchable on two properties but without annotating the class.

```

import org.apache.lucene.search.Query;
import org.hibernate.search.cfg.Environment;
import org.hibernate.search.cfg.SearchMapping;
import org.hibernate.search.query.dsl.QueryBuilder;
import org.infinispan.Cache;
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.configuration.cache.Index;
import org.infinispan.manager.DefaultCacheManager;
import org.infinispan.query.CacheQuery;
import org.infinispan.query.Search;
import org.infinispan.query.SearchManager;

import java.io.IOException;
import java.lang.annotation.ElementType;
import java.util.Properties;

SearchMapping mapping = new SearchMapping();
mapping.entity(Author.class).indexed()
    .property("name", ElementType.METHOD).field()
    .property("surname", ElementType.METHOD).field();

Properties properties = new Properties();
properties.put(Environment.MODEL_MAPPING, mapping);
properties.put("hibernate.search.[other options]", "[...]");

Configuration infinispanConfiguration = new ConfigurationBuilder()
    .indexing().index(Index.LOCAL)
    .withProperties(properties)
    .build();

DefaultCacheManager cacheManager = new DefaultCacheManager(infinispanConfiguration);

Cache<Long, Author> cache = cacheManager.getCache();
SearchManager sm = Search.getSearchManager(cache);

Author author = new Author(1, "Manik", "Surtani");
cache.put(author.getId(), author);

QueryBuilder qb = sm.buildQueryBuilderForClass(Author.class).get();
Query q = qb.keyword().onField("name").matching("Manik").createQuery();
CacheQuery cq = sm.getQuery(q, Author.class);
assert cq.getResultSize() == 1;

```

11.2.3. Querying APIs

{brandname} allows to query using Lucene queries directly and its own query language called Ickle, a subset of JP-QL with full-text extensions.

In terms of DSL, {brandname} exposes the Hibernate Search DSL (which produces Lucene queries) and has its own DSL which internally generates an Ickle query.

Finally, when using Lucene or Hibernate Search Query API, it is possible to query a single node or to broadcast a query to multiple nodes combining the results.

Hibernate Search

Apart from supporting Hibernate Search annotations to configure indexing, it's also possible to query the cache using other Hibernate Search APIs

Running Lucene queries

To run a Lucene query directly, simply create and wrap it in a *CacheQuery*:

```
import org.infinispan.query.Search;
import org.infinispan.query.SearchManager;
import org.apache.lucene.Query;

SearchManager searchManager = Search.getSearchManager(cache);
Query query = searchManager.buildQueryBuilderForClass(Book.class).get()
    .keyword().wildcard().onField("description").matching("*test*")
    .createQuery();
CacheQuery<Book> cacheQuery = searchManager.getQuery(query);
```

Using the Hibernate Search DSL

The Hibernate Search DSL can be used to create the Lucene Query, example:

```
import org.infinispan.query.Search;
import org.infinispan.query.SearchManager;
import org.apache.lucene.search.Query;

Cache<String, Book> cache = ...

SearchManager searchManager = Search.getSearchManager(cache);

Query luceneQuery = searchManager
    .buildQueryBuilderForClass(Book.class).get()
    .range().onField("year").from(2005).to(2010)
    .createQuery();

List<Object> results = searchManager.getQuery(luceneQuery).list();
```

For a detailed description of the query capabilities of this DSL, see the relevant section of the [Hibernate Search manual](#).

Faceted Search

{brandname} support [Faceted Searches](#) by using the Hibernate Search [FacetManager](#):

```
// Cache is indexed
Cache<Integer, Book> cache = ...

// Obtain the Search Manager
SearchManager searchManager = Search.getSearchManager(cache);

// Create the query builder
QueryBuilder queryBuilder = searchManager.buildQueryBuilderForClass(Book.class).get();

// Build any Lucene Query. Here it's using the DSL to do a Lucene term query on a book
// name
Query luceneQuery = queryBuilder.keyword().wildcard().onField("name").matching(
    "bitcoin").createQuery();

// Wrap into a cache Query
CacheQuery<Book> query = searchManager.getQuery(luceneQuery);

// Define the Facet characteristics
FacetingRequest request = queryBuilder.facet()
    .name("year_facet")
    .onField("year")
    .discrete()
    .orderBy(FacetSortOrder.COUNT_ASC)
    .createFacetingRequest();

// Associated the FacetRequest with the query
FacetManager facetManager = query.getFacetManager().enableFaceting(request);

// Obtain the facets
List<Facet> facetList = facetManager.getFacets("year_facet");
```

A Faceted search like above will return the number books that match 'bitcoin' released on a yearly basis, for example:

```
AbstractFacet{facetingName='year_facet', fieldName='year', value='2008', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2009', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2010', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2011', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2012', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2016', count=1}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2015', count=2}
AbstractFacet{facetingName='year_facet', fieldName='year', value='2013', count=3}
```

For more info about Faceted Search, see [Hibernate Search Faceting](#)

Spatial Queries

{brandname} also supports [Spatial Queries](#), allowing to combining full-text with restrictions based on distances, geometries or geographic coordinates.

Example, we start by using the `@Spatial` annotation in our entity that will be searched, together with `@Latitude` and `@Longitude`:

```
@Indexed
@Spatial
public class Restaurant {

    @Latitude
    private Double latitude;

    @Longitude
    private Double longitude;

    @Field(store = Store.YES)
    String name;

    // Getters, Setters and other members omitted

}
```

to run spatial queries, the Hibernate Search DSL can be used:

```
// Cache is configured as indexed
Cache<String, Restaurant> cache = ...

// Obtain the SearchManager
Searchmanager searchManager = Search.getSearchManager(cache);

// Build the Lucene Spatial Query
Query query = Search.getSearchManager(cache).buildQueryBuilderForClass(Restaurant
.class).get()
    .spatial()
    .within( 2, Unit.KM )
    .ofLatitude( centerLatitude )
    .andLongitude( centerLongitude )
    .createQuery();

// Wrap in a cache Query
CacheQuery<Restaurant> cacheQuery = searchManager.getQuery(query);

List<Restaurant> nearBy = cacheQuery.list();
```

More info on [Hibernate Search manual](#)

IndexedQueryMode

It's possible to specify a query mode for indexed queries. `IndexedQueryMode.BROADCAST` allows to broadcast a query to each node of the cluster, retrieve the results and combine them before returning to the caller. It is suitable for use in conjunction with [non-shared indexes](#), since each node's local index will have only a subset of the data indexed.

`IndexedQueryMode.FETCH` will execute the query in the caller. If all the indexes for the cluster wide data are available locally, performance will be optimal, otherwise this query mode may involve fetching indexes data from remote nodes.

The `IndexedQueryMode` is supported for Lucene Queries and Ickle String queries at the moment (no `{brandname}` Query DSL).

Example:

```
CacheQuery<Person> broadcastQuery = Search.getSearchManager(cache).getQuery(new
MatchAllDocsQuery(), IndexedQueryMode.BROADCAST);

List<Person> result = broadcastQuery.list();
```

{brandname} Query DSL

`{brandname}` provides its own query DSL, independent of Lucene and Hibernate Search. Decoupling the query API from the underlying query and indexing mechanism makes it possible to introduce new alternative engines in the future, besides Lucene, and still being able to use the same uniform query API. The current implementation of indexing and searching is still based on Hibernate Search and Lucene so all indexing related aspects presented in this chapter still apply.

The new API simplifies the writing of queries by not exposing the user to the low level details of constructing Lucene query objects and also has the advantage of being available to remote Hot Rod clients. But before delving into further details, let's examine first a simple example of writing a query for the *Book* entity from the previous example.

Query example using {brandname}'s query DSL

```
import org.infinispan.query.dsl.*;

// get the DSL query factory from the cache, to be used for constructing the Query
object:
QueryFactory qf = org.infinispan.query.Search.getQueryFactory(cache);

// create a query for all the books that have a title which contains "engine":
org.infinispan.query.dsl.Query query = qf.from(Book.class)
    .having("title").like("%engine%")
    .build();

// get the results:
List<Book> list = query.list();
```

The API is located in the `org.infinispan.query.dsl` package. A query is created with the help of the `QueryFactory` instance which is obtained from the per-cache `SearchManager`. Each `QueryFactory` instance is bound to the same `Cache` instance as the `SearchManager`, but it is otherwise a stateless and thread-safe object that can be used for creating multiple queries in parallel.

Query creation starts with the invocation of the `from(Class entityType)` method which returns a `QueryBuilder` object that is further responsible for creating queries targeted to the specified entity class from the given cache.



A query will always target a single entity type and is evaluated over the contents of a single cache. Running a query over multiple caches or creating queries that target several entity types (joins) is not supported.

The `QueryBuilder` accumulates search criteria and configuration specified through the invocation of its DSL methods and is ultimately used to build a `Query` object by the invocation of the `QueryBuilder.build()` method that completes the construction. Being a stateful object, it cannot be used for constructing multiple queries at the same time (except for [nested queries](#)) but can be reused afterwards.



This `QueryBuilder` is different from the one from Hibernate Search but has a somewhat similar purpose, hence the same name. We are considering renaming it in near future to prevent ambiguity.

Executing the query and fetching the results is as simple as invoking the `list()` method of the `Query` object. Once executed the `Query` object is not reusable. If you need to re-execute it in order to obtain fresh results then a new instance must be obtained by calling `QueryBuilder.build()`.

Filtering operators

Constructing a query is a hierarchical process of composing multiple criteria and is best explained following this hierarchy.

The simplest possible form of a query criteria is a restriction on the values of an entity attribute according to a filtering operator that accepts zero or more arguments. The entity attribute is specified by invoking the `having(String attributePath)` method of the query builder which returns an intermediate context object ([FilterConditionEndContext](#)) that exposes all the available operators. Each of the methods defined by `FilterConditionEndContext` is an operator that accepts an argument, except for `between` which has two arguments and `isNull` which has no arguments. The arguments are statically evaluated at the time the query is constructed, so if you're looking for a feature similar to SQL's correlated sub-queries, that is not currently available.

```
// a single query criterion
QueryBuilder qb = ...
qb.having("title").eq("Hibernate Search in Action");
```

Table 5. `FilterConditionEndContext` exposes the following filtering operators:

Filter	Arguments	Description
in	Collection values	Checks that the left operand is equal to one of the elements from the Collection of values given as argument.
in	Object... values	Checks that the left operand is equal to one of the (fixed) list of values given as argument.
contains	Object value	Checks that the left argument (which is expected to be an array or a Collection) contains the given element.
containsAll	Collection values	Checks that the left argument (which is expected to be an array or a Collection) contains all the elements of the given collection, in any order.
containsAll	Object... values	Checks that the left argument (which is expected to be an array or a Collection) contains all of the the given elements, in any order.
containsAny	Collection values	Checks that the left argument (which is expected to be an array or a Collection) contains any of the elements of the given collection.
containsAny	Object... values	Checks that the left argument (which is expected to be an array or a Collection) contains any of the the given elements.
isNull		Checks that the left argument is null.
like	String pattern	Checks that the left argument (which is expected to be a String) matches a wildcard pattern that follows the JPA rules.
eq	Object value	Checks that the left argument is equal to the given value.
equal	Object value	Alias for eq.
gt	Object value	Checks that the left argument is greater than the given value.
gte	Object value	Checks that the left argument is greater than or equal to the given value.

Filter	Arguments	Description
lt	Object value	Checks that the left argument is less than the given value.
lte	Object value	Checks that the left argument is less than or equal to the given value.
between	Object from, Object to	Checks that the left argument is between the given range limits.

It's important to note that query construction requires a multi-step chaining of method invocation that must be done in the proper sequence, must be properly completed exactly *once* and must not be done twice, or it will result in an error. The following examples are invalid, and depending on each case they lead to criteria being ignored (in benign cases) or an exception being thrown (in more serious ones).

```
// Incomplete construction. This query does not have any filter on "title" attribute yet,
// although the author may have intended to add one.
QueryBuilder qb1 = ...
qb1.having("title");
Query q1 = qb1.build(); // consequently, this query matches all Book instances regardless of title!

// Duplicated completion. This results in an exception at run-time.
// Maybe the author intended to connect two conditions with a boolean operator,
// but this does NOT actually happen here.
QueryBuilder qb2 = ...
qb2.having("title").like("%Data Grid%");
qb2.having("description").like("%clustering%"); // will throw
java.lang.IllegalStateException: Sentence already started. Cannot use 'having(..)' again.
Query q2 = qb2.build();
```

Filtering based on attributes of embedded entities

The **having** method also accepts dot separated attribute paths for referring to *embedded entity* attributes, so the following is a valid query:

```
// match all books that have an author named "Manik"
Query query = queryFactory.from(Book.class)
    .having("author.name").eq("Manik")
    .build();
```

Each part of the attribute path must refer to an existing indexed attribute in the corresponding entity or embedded entity class respectively. It's possible to have multiple levels of embedding.

Boolean conditions

Combining multiple attribute conditions with logical conjunction (**and**) and disjunction (**or**) operators in order to create more complex conditions is demonstrated in the following example. The well known operator precedence rule for boolean operators applies here, so the order of DSL method invocations during construction is irrelevant. Here **and** operator still has higher priority than **or** even though **or** was invoked first.

```
// match all books that have "Data Grid" in their title
// or have an author named "Manik" and their description contains "clustering"
Query query = queryFactory.from(Book.class)
    .having("title").like("%Data Grid%")
    .or().having("author.name").eq("Manik")
    .and().having("description").like("%clustering%")
    .build();
```

Boolean negation is achieved with the **not** operator, which has highest precedence among logical operators and applies only to the next simple attribute condition.

```
// match all books that do not have "Data Grid" in their title and are authored by
// "Manik"
Query query = queryFactory.from(Book.class)
    .not().having("title").like("%Data Grid%")
    .and().having("author.name").eq("Manik")
    .build();
```

Nested conditions

Changing the precedence of logical operators is achieved with nested filter conditions. Logical operators can be used to connect two simple attribute conditions as presented before, but can also connect a simple attribute condition with the subsequent complex condition created with the same query factory.

```
// match all books that have an author named "Manik" and their title contains
// "Data Grid" or their description contains "clustering"
Query query = queryFactory.from(Book.class)
    .having("author.name").eq("Manik")
    .and(queryFactory.having("title").like("%Data Grid%")
        .or().having("description").like("%clustering%"))
    .build();
```

Projections

In some use cases returning the whole domain object is overkill if only a small subset of the attributes are actually used by the application, especially if the domain entity has embedded entities. The query language allows you to specify a subset of attributes (or attribute paths) to return - the projection. If projections are used then the `Query.list()` will not return the whole

domain entity but will return a *List of Object[]*, each slot in the array corresponding to a projected attribute.

TODO document what needs to be configured for an attribute to be available for projection.

```
// match all books that have "Data Grid" in their title or description
// and return only their title and publication year
Query query = queryFactory.from(Book.class)
    .select("title", "publicationYear")
    .having("title").like("%Data Grid%")
    .or().having("description").like("%Data Grid%"))
    .build();
```

Sorting

Ordering the results based on one or more attributes or attribute paths is done with the `QueryBuilder.orderBy()` method which accepts an attribute path and a sorting direction. If multiple sorting criteria are specified, then the order of invocation of `orderBy` method will dictate their precedence. But you have to think of the multiple sorting criteria as acting together on the tuple of specified attributes rather than in a sequence of individual sorting operations on each attribute.

TODO document what needs to be configured for an attribute to be available for sorting.

```
// match all books that have "Data Grid" in their title or description
// and return them sorted by the publication year and title
Query query = queryFactory.from(Book.class)
    .orderBy("publicationYear", SortOrder.DESC)
    .orderBy("title", SortOrder.ASC)
    .having("title").like("%Data Grid%")
    .or().having("description").like("%Data Grid%"))
    .build();
```

Pagination

You can limit the number of returned results by setting the *maxResults* property of *QueryBuilder*. This can be used in conjunction with setting the *startOffset* in order to achieve pagination of the result set.

```
// match all books that have "clustering" in their title
// sorted by publication year and title
// and return 3'rd page of 10 results
Query query = queryFactory.from(Book.class)
    .orderBy("publicationYear", SortOrder.DESC)
    .orderBy("title", SortOrder.ASC)
    .startOffset(20)
    .maxResults(10)
    .having("title").like("%clustering%")
    .build();
```



Even if the results being fetched are limited to *maxResults* you can still find the total number of matching results by calling *Query.getResultSize()*.

TODO Does pagination make sense if no stable sort criteria is defined? Luckily when running on Lucene and no sort criteria is specified we still have the order of relevance, but this has to be defined for other search engines.

Grouping and Aggregation

{brandname} has the ability to group query results according to a set of grouping fields and construct aggregations of the results from each group by applying an aggregation function to the set of values that fall into each group. Grouping and aggregation can only be applied to projection queries. The supported aggregations are: avg, sum, count, max, min. The set of grouping fields is specified with the *groupBy(field)* method, which can be invoked multiple times. The order used for defining grouping fields is not relevant. All fields selected in the projection must either be grouping fields or else they must be aggregated using one of the grouping functions described below. A projection field can be aggregated and used for grouping at the same time. A query that selects only grouping fields but no aggregation fields is legal.

Example: Grouping Books by author and counting them.

```
Query query = queryFactory.from(Book.class)
    .select(Expression.property("author"), Expression.count("title"))
    .having("title").like("%engine%")
    .groupBy("author")
    .build();
```



A projection query in which all selected fields have an aggregation function applied and no fields are used for grouping is allowed. In this case the aggregations will be computed globally as if there was a single global group.

Aggregations

The following aggregation functions may be applied to a field: avg, sum, count, max, min

- avg() - Computes the average of a set of numbers. Accepted values are primitive numbers and

instances of *java.lang.Number*. The result is represented as *java.lang.Double*. If there are no non-null values the result is *null* instead.

- `count()` - Counts the number of non-null rows and returns a *java.lang.Long*. If there are no non-null values the result is *0* instead.
- `max()` - Returns the greatest value found. Accepted values must be instances of *java.lang.Comparable*. If there are no non-null values the result is *null* instead.
- `min()` - Returns the smallest value found. Accepted values must be instances of *java.lang.Comparable*. If there are no non-null values the result is *null* instead.
- `sum()` - Computes the sum of a set of Numbers. If there are no non-null values the result is *null* instead. The following table indicates the return type based on the specified field.

Table 6. Table sum return type

Field Type	Return Type
Integral (other than BigInteger)	Long
Float or Double	Double
BigInteger	BigInteger
BigDecimal	BigDecimal

Evaluation of queries with grouping and aggregation

Aggregation queries can include filtering conditions, like usual queries. Filtering can be performed in two stages: before and after the grouping operation. All filter conditions defined before invoking the *groupBy* method will be applied before the grouping operation is performed, directly to the cache entries (not to the final projection). These filter conditions may reference any fields of the queried entity type, and are meant to restrict the data set that is going to be the input for the grouping stage. All filter conditions defined after invoking the *groupBy* method will be applied to the projection that results from the projection and grouping operation. These filter conditions can either reference any of the *groupBy* fields or aggregated fields. Referencing aggregated fields that are not specified in the select clause is allowed; however, referencing non-aggregated and non-grouping fields is forbidden. Filtering in this phase will reduce the amount of groups based on their properties. Sorting may also be specified similar to usual queries. The ordering operation is performed after the grouping operation and can reference any of the *groupBy* fields or aggregated fields.

Using Named Query Parameters

Instead of building a new Query object for every execution it is possible to include named parameters in the query which can be substituted with actual values before execution. This allows a query to be defined once and be efficiently executed many times. Parameters can only be used on the right-hand side of an operator and are defined when the query is created by supplying an object produced by the *org.infinispan.query.dsl.Expression.param(String paramName)* method to the operator instead of the usual constant value. Once the parameters have been defined they can be set by invoking either *Query.setParameter(parameterName, value)* or *Query.setParameters(parameterMap)* as shown in the examples below.

```

import org.infinispan.query.Search;
import org.infinispan.query.dsl.*;
[...]

QueryFactory queryFactory = Search.getQueryFactory(cache);
// Defining a query to search for various authors and publication years
Query query = queryFactory.from(Book.class)
    .select("title")
    .having("author").eq(Expression.param("authorName"))
    .and()
    .having("publicationYear").eq(Expression.param("publicationYear"))
    .build();

// Set actual parameter values
query.setParameter("authorName", "Doe");
query.setParameter("publicationYear", 2010);

// Execute the query
List<Book> found = query.list();

```

Alternatively, multiple parameters may be set at once by supplying a map of actual parameter values:

Setting multiple named parameters at once

```

import java.util.Map;
import java.util.HashMap;

[...]

Map<String, Object> parameterMap = new HashMap<>();
parameterMap.put("authorName", "Doe");
parameterMap.put("publicationYear", 2010);

query.setParameters(parameterMap);

```



A significant portion of the query parsing, validation and execution planning effort is performed during the first execution of a query with parameters. This effort is not repeated during subsequent executions leading to better performance compared to a similar query using constant values instead of query parameters.

More Query DSL samples

Probably the best way to explore using the Query DSL API is to have a look at our tests suite. [QueryDslConditionsTest](#) is a fine example.

Ickle

Using Ickle, a light and small subset of JP-QL with full-text extensions, it is possible to create relational and full-text queries in both Library and Remote Client-Server mode. Ickle is a string-based querying language, and has the following characteristics:

- Query Java classes and supports Protocol Buffers.
- Queries can target a single entity type.
- Queries can filter on properties of embedded objects, including collections.
- Supports projections, aggregations, sorting, named parameters.
- Supports indexed and non-indexed execution.
- Supports complex boolean expressions.
- Supports full-text queries.
- Does not support computations in expressions, such as `user.age > sqrt(user.shoeSize+3)`.
- Does not support joins.
- Does not support subqueries.
- Is supported across various `{{brandname}}` APIs. Whenever a Query is produced by the QueryBuilder is accepted, including continuous queries or in event filters for listeners.

To use the API, first obtain a QueryFactory to the cache and then call the `.create()` method, passing in the string to use in the query. For instance:

```
QueryFactory qf = Search.getQueryFactory(remoteCache);
Query q = qf.create("from sample_bank_account.Transaction where amount > 20");
```

When using Ickle all fields used with full-text operators must be both **Indexed** and **Analysed**.

Deviations from the Lucene Query Parser Syntax

While Ickle is a subset of JP-QL it does have the following deviations in its query syntax:

- Whitespace is not significant.
- There is no support for wildcards in field names.
- A field name or path must always be specified, as there is no default field.
- **&&** and **||** are accepted instead of **AND** or **OR** in both full-text and JPA predicates.
- **!** may be used instead of **NOT**.
- A missing boolean operator is interpreted as **OR**.
- String terms must be enclosed with either single or double quotes.
- Fuzziness and boosting are not accepted in arbitrary order; fuzziness always comes first.
- **!=** is accepted instead of **<>**.
- Boosting cannot be applied to **>, >=, <, <=** operators. Ranges may be used to achieve the same

result.

Fuzzy Queries

To execute a fuzzy query add ~ along with an integer, representing the distance from the term used, after the term. For instance

```
Query fuzzyQuery = qf.create("from sample_bank_account.Transaction where description :  
'cofee'~2");
```

Range Queries

To execute a range query define the given boundaries within a pair of braces, as seen in the following example:

```
Query rangeQuery = qf.create("from sample_bank_account.Transaction where amount : [20 to 50]");
```

Phrase Queries

A group of words may be searched by surrounding them in quotation marks, as seen in the following example:

```
Query q = qf.create("from sample_bank_account.Transaction where description : 'bus fare'");
```

Proximity Queries

To execute a proximity query, finding two terms within a specific distance, add a ~ along with the distance after the phrase. For instance, the following example will find the words canceling and fee provided they are not more than 3 words apart:

```
Query proximityQuery = qf.create("from sample_bank_account.Transaction where  
description : 'canceling fee'~3 ");
```

Wildcard Queries

Both single-character and multi-character wildcard searches may be performed:

- A single-character wildcard search may be used with the ? character.
- A multi-character wildcard search may be used with the * character.

To search for text or test the following single-character wildcard search would be used:

```
Query wildcardQuery = qf.create("from sample_bank_account.Transaction where  
description : 'te?t'");
```

To search for test, tests, or tester the following multi-character wildcard search would be used:

```
Query wildcardQuery = qf.create("from sample_bank_account.Transaction where  
description : 'test*'");
```

Regular Expression Queries

Regular expression queries may be performed by specifying a pattern between /. Ickle uses Lucene's regular expression syntax, so to search for the words moat or boat the following could be used:

```
Query regExpQuery = qf.create("from sample_library.Book where title : /[mb]oat/");
```

Boosting Queries

Terms may be boosted by adding a ^ after the term to increase their relevance in a given query, the higher the boost factor the more relevant the term will be. For instance to search for titles containing beer and wine with a higher relevance on beer, by a factor of 3, the following could be used:

```
Query boostedQuery = qf.create("from sample_library.Book where title : beer^3 OR wine  
");
```

Continuous Query

Continuous Queries allow an application to register a listener which will receive the entries that currently match a query filter, and will be continuously notified of any changes to the queried data set that result from further cache operations. This includes incoming matches, for values that have joined the set, updated matches, for matching values that were modified and continue to match, and outgoing matches, for values that have left the set. By using a Continuous Query the application receives a steady stream of events instead of having to repeatedly execute the same query to discover changes, resulting in a more efficient use of resources. For instance, all of the following use cases could utilize Continuous Queries:

- Return all persons with an age between 18 and 25 (assuming the Person entity has an *age* property and is updated by the user application).
- Return all transactions higher than \$2000.
- Return all times where the lap speed of F1 racers were less than 1:45.00s (assuming the cache contains Lap entries and that laps are entered live during the race).

Continuous Query Execution

A continuous query uses a listener that is notified when:

- An entry starts matching the specified query, represented by a *Join* event.
- A matching entry is updated and continues to match the query, represented by an *Update* event.
- An entry stops matching the query, represented by a *Leave* event.

When a client registers a continuous query listener it immediately begins to receive the results currently matching the query, received as *Join* events as described above. In addition, it will receive subsequent notifications when other entries begin matching the query, as *Join* events, or stop matching the query, as *Leave* events, as a consequence of any cache operations that would normally generate creation, modification, removal, or expiration events. Updated cache entries will generate *Update* events if the entry matches the query filter before and after the operation. To summarize, the logic used to determine if the listener receives a *Join*, *Update* or *Leave* event is:

1. If the query on both the old and new values evaluate false, then the event is suppressed.
2. If the query on the old value evaluates false and on the new value evaluates true, then a *Join* event is sent.
3. If the query on both the old and new values evaluate true, then an *Update* event is sent.
4. If the query on the old value evaluates true and on the new value evaluates false, then a *Leave* event is sent.
5. If the query on the old value evaluates true and the entry is removed or expired, then a *Leave* event is sent.



Continuous Queries can use the full power of the Query DSL except: grouping, aggregation, and sorting operations.

Running Continuous Queries

To create a continuous query you'll start by creating a Query object first. This is described in [the Query DSL section](#). Then you'll need to obtain the ContinuousQuery (`org.infinispan.query.api.continuous.ContinuousQuery`) object of your cache and register the query and a continuous query listener (`org.infinispan.query.api.continuous.ContinuousQueryListener`) with it. A ContinuousQuery object associated to a cache can be obtained by calling the static method `org.infinispan.client.hotrod.Search.getContinuousQuery(RemoteCache<K, V> cache)` if running in remote mode or `org.infinispan.query.Search.getContinuousQuery(Cache<K, V> cache)` when running in embedded mode. Once the listener has been created it may be registered by using the `addContinuousQueryListener` method of ContinuousQuery:

```
continuousQuery.addContinuousQueryListener(query, listener);
```

The following example demonstrates a simple continuous query use case in embedded mode:

Registering a Continuous Query

```
import org.infinispan.query.api.continuous.ContinuousQuery;
import org.infinispan.query.api.continuous.ContinuousQueryListener;
import org.infinispan.query.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
```

[...]

```
// We have a cache of Persons
Cache<Integer, Person> cache = ...
```

```
// We begin by creating a ContinuousQuery instance on the cache
ContinuousQuery<Integer, Person> continuousQuery = Search.getContinuousQuery(cache);
```

```
// Define our query. In this case we will be looking for any Person instances under 21
years of age.
```

```
QueryFactory queryFactory = Search.getQueryFactory(cache);
Query query = queryFactory.from(Person.class)
    .having("age").lt(21)
    .build();
```

```
final Map<Integer, Person> matches = new ConcurrentHashMap<Integer, Person>();
```

```
// Define the ContinuousQueryListener
```

```
ContinuousQueryListener<Integer, Person> listener = new ContinuousQueryListener
<Integer, Person>() {
```

```
    @Override
    public void resultJoining(Integer key, Person value) {
        matches.put(key, value);
    }
}
```

```
    @Override
    public void resultUpdated(Integer key, Person value) {
        // we do not process this event
    }
}
```

```
    @Override
    public void resultLeaving(Integer key) {
        matches.remove(key);
    }
}
```

```
};
```

```
// Add the listener and the query
continuousQuery.addContinuousQueryListener(query, listener);
```

[...]

```
// Remove the listener to stop receiving notifications
continuousQuery.removeContinuousQueryListener(listener);
```

As Person instances having an age less than 21 are added to the cache they will be received by the listener and will be placed into the *matches* map, and when these entries are removed from the cache or their age is modified to be greater or equal than 21 they will be removed from *matches*.

Removing Continuous Queries

To stop the query from further execution just remove the listener:

```
continuousQuery.removeContinuousQueryListener(listener);
```

Notes on performance of Continuous Queries

Continuous queries are designed to provide a constant stream of updates to the application, potentially resulting in a very large number of events being generated for particularly broad queries. A new temporary memory allocation is made for each event. This behavior may result in memory pressure, potentially leading to *OutOfMemoryErrors* (especially in remote mode) if queries are not carefully designed. To prevent such issues it is strongly recommended to ensure that each query captures the minimal information needed both in terms of number of matched entries and size of each match (projections can be used to capture the interesting properties), and that each *ContinuousQueryListener* is designed to quickly process all received events without blocking and to avoid performing actions that will lead to the generation of new matching events from the cache it listens to.

11.3. Remote Querying

Apart from supporting indexing and searching of Java entities to embedded clients, {brandname} introduced support for remote, language neutral, querying.

This leap required two major changes:

- Since non-JVM clients cannot benefit from directly using [Apache Lucene's](#) Java API, {brandname} defines its own new [query language](#), based on an internal DSL that is easily implementable in all languages for which we currently have an implementation of the Hot Rod client.
- In order to enable indexing, the entities put in the cache by clients can no longer be opaque binary blobs understood solely by the client. Their structure has to be known to both server and client, so a common way of encoding structured data had to be adopted. Furthermore, allowing multi-language clients to access the data requires a language and platform-neutral encoding. Google's [Protocol Buffers](#) was elected as an encoding format for both over-the-wire and storage due to its efficiency, robustness, good multi-language support and support for schema evolution.

11.3.1. Storing Protobuf encoded entities

Remote clients that want to be able to index and query their stored entities must do so using the Protobuf encoding format. This is *key* for the search capability to work. But it's also possible to store Protobuf entities just for gaining the benefit of platform independence and not enable indexing if you do not need it.

Protobuf is all about structured data, so first thing you do to use it is define the structure of your data. This is accomplished by declaring protocol buffer message types in .proto files, like in the following example. Protobuf is a broad subject, we will not detail it here, so please consult the Protobuf [Developer Guide](#) for an in-depth explanation. It suffices to say for now that our example

defines an entity (message type in protobuf speak) named *Book*, placed in a package named *book_sample*. Our entity declares several fields of primitive types and a repeatable field (an array basically) named *authors*. The *Author* message instances are embedded in the *Book* message instance.

library.proto

```
package book_sample;

message Book {
    required string title = 1;
    required string description = 2;
    required int32 publicationYear = 3; // no native Date type available in Protobuf

    repeated Author authors = 4;
}

message Author {
    required string name = 1;
    required string surname = 2;
}
```

There are a few important notes we need to make about Protobuf messages:

- nesting of messages is possible, but the resulting structure is strictly a tree, never a graph
- there is no concept of type inheritance
- collections are not supported but arrays can be easily emulated using repeated fields

Using Protobuf with the Java Hot Rod client is a two step process. First, the client must be configured to use a dedicated marshaller, [ProtoStreamMarshaller](#). This marshaller uses the [ProtoStream](#) library to assist you in encoding your objects. The second step is instructing *ProtoStream* library on how to marshall your message types. The following example highlights this process.

```

import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.marshall.ProtoStreamMarshaller;
import org.infinispan.protostream.SerializationContext;
...

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("10.1.2.3").port(11234)
    .marshaller(new ProtoStreamMarshaller());

RemoteCacheManager remoteCacheManager = new RemoteCacheManager(clientBuilder.build());

SerializationContext serCtx = ProtoStreamMarshaller.getSerializationContext
(remoteCacheManager);

FileDescriptorSource fds = new FileDescriptorSource();
fds.addProtoFiles("/library.proto");
serCtx.registerProtoFiles(fds);
serCtx.registerMarshaller(new BookMarshaller());
serCtx.registerMarshaller(new AuthorMarshaller());

// Book and Author classes omitted for brevity

```

The interesting part in this sample is obtaining the *SerializationContext* associated to the *RemoteCacheManager* and then instructing *ProtoStream* about the protobuf types we want to marshal. The *SerializationContext* is provided by the library for this purpose. The *SerializationContext.registerProtoFiles* method receives the name of one or more classpath resources that is expected to be a protobuf definition containing our type declarations.



A *RemoteCacheManager* has no *SerializationContext* associated with it unless it was configured to use a *ProtoStreamMarshaller*.

The next relevant part is the registration of per entity marshallers for our domain model types. They must be provided by the user for each type or marshalling will fail. Writing marshallers is a simple process. The *BookMarshaller* example should get you started. The most important thing you need to consider is they need to be stateless and threadsafe as a single instance of them is being used.

```

import org.infinispan.protostream.MessageMarshaller;
...

public class BookMarshaller implements MessageMarshaller<Book> {

    @Override
    public String getTypeName() {
        return "book_sample.Book";
    }

    @Override
    public Class<? extends Book> getJavaClass() {
        return Book.class;
    }

    @Override
    public void writeTo(ProtoStreamWriter writer, Book book) throws IOException {
        writer.writeString("title", book.getTitle());
        writer.writeString("description", book.getDescription());
        writer.writeInt("publicationYear", book.getPublicationYear());
        writer.writeCollection("authors", book.getAuthors(), Author.class);
    }

    @Override
    public Book readFrom(ProtoStreamReader reader) throws IOException {
        String title = reader.readString("title");
        String description = reader.readString("description");
        int publicationYear = reader.readInt("publicationYear");
        Set<Author> authors = reader.readCollection("authors", new HashSet<>(), Author
.class);
        return new Book(title, description, publicationYear, authors);
    }
}

```

Once you've followed these steps to setup your client you can start reading and writing Java objects to the remote cache and the actual data stored in the cache will be protobuf encoded provided that marshallers were registered with the remote client for all involved types (*Book* and *Author* in our example). Keeping your objects stored in protobuf format has the benefit of being able to consume them with compatible clients written in different languages.

TODO Add reference to sample in C++ client user guide

11.3.2. Using annotations

TODO

11.3.3. Indexing of Protobuf encoded entries

After configuring the client as described in the previous section you can start configuring indexing for your caches on the server side. Activating indexing and the various indexing specific configurations is identical to embedded mode and is detailed in the [Querying {brandname}](#) chapter.

There is however an extra configuration step involved. While in embedded mode the indexing metadata is obtained via Java reflection by analyzing the presence of various Hibernate Search annotations on the entry's class, this is obviously not possible if the entry is protobuf encoded. The server needs to obtain the relevant metadata from the same descriptor (.proto file) as the client. The descriptors are stored in a dedicated cache on the server named '`__protobuf_metadata`'. Both keys and values in this cache are plain strings. Registering a new schema is therefore as simple as performing a *put* operation on this cache using the schema's name as key and the schema file itself as the value. Alternatively you can use the CLI (via the `cache-container=*.register-proto-schemas()` operation), the Management Console or the *ProtobufMetadataManager* MBean via JMX. Be aware that, when security is enabled, access to the schema cache via the remote protocols requires that the user belongs to the '`__schema_manager`' role.



Once indexing is enabled for a cache all fields of Protobuf encoded entries will be fully indexed unless you use the `@Indexed` and `@Field` protobuf schema pseudo-annotations in order to control precisely what fields need to get indexed. The default behaviour can be very inefficient when dealing with types having many or very larger fields so we encourage you to always specify what fields should be indexed instead of relying on the default indexing behaviour. The indexing behaviour for protobuf message types that are not annotated can also be modified per each schema file by setting the protobuf schema option '`indexed_by_default`' to *false* (its default value is considered *true*) at the beginning of your schema file.

```
option indexed_by_default = false; // This disables indexing of types that are not
annotated for indexing
```

11.3.4. A remote query example

You've managed to configure both client and server to talk protobuf and you've enabled indexing. Let's put some data in the cache and try to search for it then!

```

import org.infinispan.client.hotrod.*;
import org.infinispan.query.dsl.*;
...

RemoteCacheManager remoteCacheManager = ...;
RemoteCache<Integer, Book> remoteCache = remoteCacheManager.getCache();

Book book1 = new Book();
book1.setTitle("Hibernate in Action");
remoteCache.put(1, book1);

Book book2 = new Book();
book2.setTile("Hibernate Search in Action");
remoteCache.put(2, book2);

QueryFactory qf = Search.getQueryFactory(remoteCache);
Query query = qf.from(Book.class)
    .having("title").like("%Hibernate Search%")
    .build();

List<Book> list = query.list(); // Voila! We have our book back from the cache!

```

The key part of creating a query is obtaining the *QueryFactory* for the remote cache using the *org.infinispan.client.hotrod.Search.getQueryFactory()* method. Once you have this creating the query is similar to embedded mode which is covered in [this](#) section.

11.3.5. Analysis

Analysis is a process that converts input data into one or more terms that you can index and query.

Default Analyzers

{brandname} provides a set of default analyzers as follows:

Definition	Description
standard	Splits text fields into tokens, treating whitespace and punctuation as delimiters.
simple	Tokenizes input streams by delimiting at non-letters and then converting all letters to lowercase characters. Whitespace and non-letters are discarded.
whitespace	Splits text streams on whitespace and returns sequences of non-whitespace characters as tokens.
keyword	Treats entire text fields as single tokens.
stemmer	Stems English words using the Snowball Porter filter.

Definition	Description
<code>ngram</code>	Generates n-gram tokens that are 3 grams in size by default.

These analyzer definitions are based on Apache Lucene and are provided "as-is". For more information about tokenizers, filters, and CharFilters, see the appropriate Lucene documentation.

Using Analyzer Definitions

To use analyzer definitions, reference them by name in the *.proto* schema file.

1. Include the `Analyze.YES` attribute to indicate that the property is analyzed.
2. Specify the analyzer definition with the `@Analyzer` annotation.

The following example shows referenced analyzer definitions:

```
/* @Indexed */
message TestEntity {

    /* @Field(store = Store.YES, analyze = Analyze.YES, analyzer =
    @Analyzer(definition = "keyword")) */
    optional string id = 1;

    /* @Field(store = Store.YES, analyze = Analyze.YES, analyzer =
    @Analyzer(definition = "simple")) */
    optional string name = 2;
}
```

Creating Custom Analyzer Definitions

If you require custom analyzer definitions, do the following:

1. Create an implementation of the `ProgrammaticSearchMappingProvider` interface packaged in a `JAR` file.
2. Provide a file named `org.infinispan.query.spi.ProgrammaticSearchMappingProvider` in the `META-INF/services/` directory of your `JAR`. This file should contain the fully qualified class name of your implementation.
3. Copy the `JAR` to the `standalone/deployments` directory of your {brandname} installation.



Your deployment must be available to the {brandname} server during startup. You cannot add the deployment if the server is already running.

The following is an example implementation of the `ProgrammaticSearchMappingProvider` interface:

```

import org.apache.lucene.analysis.core.LowerCaseFilterFactory;
import org.apache.lucene.analysis.core.StopFilterFactory;
import org.apache.lucene.analysis.standard.StandardFilterFactory;
import org.apache.lucene.analysis.standard.StandardTokenizerFactory;
import org.hibernate.search.cfg.SearchMapping;
import org.infinispan.Cache;
import org.infinispan.query.spi.ProgrammaticSearchMappingProvider;

public final class MyAnalyzerProvider implements ProgrammaticSearchMappingProvider
{
    @Override
    public void defineMappings(Cache cache, SearchMapping searchMapping) {
        searchMapping
            .analyzerDef("standard-with-stop", StandardTokenizerFactory.class)
            .filter(StandardFilterFactory.class)
            .filter(LowerCaseFilterFactory.class)
            .filter(StopFilterFactory.class);
    }
}

```

4. Specify the **JAR** in the cache container configuration, for example:

```

<cache-container name="mycache" default-cache="default">
    <modules>
        <module name="deployment.analyzers.jar"/>
    </modules>
    ...

```

11.4. Statistics

Query *Statistics* can be obtained from the *SearchManager*, as demonstrated in the following code snippet.

```

SearchManager searchManager = Search.getSearchManager(cache);
org.hibernate.search.stat.Statistics statistics = searchManager.getStatistics();

```



This data is also available via JMX through the **Hibernate Search StatisticsInfoMBean** registered under the name `org.infinispan:type=Query,manager="{name-of-cache-manager}",cache="{name-of-cache}",component=Statistics`. Please note this MBean is always registered by `{brandname}` but the statistics are collected only if **statistics collection is enabled** at cache level.



Hibernate Search has its own configuration properties `hibernate.search.jmx_enabled` and `hibernate.search.generate_statistics` for JMX statistics as explained [here](#). Using them with {brandname} Query is forbidden as it will only lead to duplicated MBeans and unpredictable results.

11.5. Performance Tuning

11.5.1. Batch writing in SYNC mode

By default, the [Index Managers](#) work in sync mode, meaning when data is written to {brandname}, it will perform the indexing operations synchronously. This synchronicity guarantees indexes are always consistent with the data (and thus visible in searches), but can slowdown write operations since it will also perform a commit to the index. Committing is an extremely expensive operation in Lucene, and for that reason, multiple writes from different nodes can be automatically batched into a single commit to reduce the impact.

So, when doing data loads to {brandname} with index enabled, try to use multiple threads to take advantage of this batching.

If using multiple threads does not result in the required performance, an alternative is to load data with indexing temporarily disabled and run a [re-indexing](#) operation afterwards. This can be done writing data with the `SKIP_INDEXING` flag:

```
cache.getAdvancedCache().withFlags(Flag.SKIP_INDEXING).put("key", "value");
```

11.5.2. Writing using async mode

If it's acceptable a small delay between data writes and when that data is visible in queries, an index manager can be configured to work in **async mode**. The async mode offers much better writing performance, since in this mode commits happen at a configurable interval.

Configuration:

```
<distributed-cache name="default">
  <indexing index="LOCAL">
    <property name="default.indexmanager">
      org.infinispan.query.indexmanager.InfinispanIndexManager
    </property>
    <!-- Index data in async mode -->
    <property name="default.worker.execution">async</property>
    <!-- Optional: configure the commit interval, default is 1000ms -->
    <property name="default.index_flush_interval">500</property>
  </indexing>
</distributed-cache>
```

11.5.3. Index reader async strategy

Lucene internally works with snapshots of the index: once an *IndexReader* is opened, it will only see the index changes up to the point it was opened; further index changes will not be visible until the *IndexReader* is refreshed. The Index Managers used in {brandname} by default will check the freshness of the index readers before every query and refresh them if necessary.

It is possible to tune this strategy to relax this freshness checking to a pre-configured interval by using the `reader.strategy` configuration set as `async`:

```
<distributed-cache name="default"
                  key-partitioner=
"org.infinispan.distribution.ch.impl.AffinityPartitioner">
  <indexing index="PRIMARY_OWNER">
    <property name="default.indexmanager">
      org.infinispan.query.affinity.AffinityIndexManager
    </property>
    <property name="default.reader.strategy">async</property>
    <!-- refresh reader every 1s, default is 5s -->
    <property name="default.reader.async_refresh_period_ms">1000</property>
  </indexing>
</distributed-cache>
```

The async reader strategy is particularly useful for Index Managers that rely on shards, such as the `AffinityIndexManager`.

11.5.4. Lucene Options

It is possible to apply tuning options in Lucene directly. For more details, see the [Hibernate Search manual](#).

Chapter 12. Clustered Counters

Clustered counters are counters which are distributed and shared among all nodes in the {brandname} 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.

12.1. Installation and Configuration

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

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-counter</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

The counters can be configured {brandname} configuration file or on-demand via the `CounterManager` interface detailed later in this document. A counter configured in {brandname} 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.

```

<?xml version="1.0" encoding="UTF-8"?>
<infinispan>
  <cache-container ...>
    <!-- if needed to persist counter, global state needs to be configured -->
    <global-state>
      ...
    </global-state>
    <!-- your caches configuration goes here -->
    <counters xmlns="urn:infinispan:config:counters:9.2" 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 value="0"/>
      </strong-counter>
      <strong-counter name="c3" initial-value="3" storage="PERSISTENT">
        <upper-bound value="5"/>
      </strong-counter>
      <strong-counter name="c4" initial-value="4" storage="VOLATILE">
        <lower-bound value="0"/>
        <upper-bound value="10"/>
      </strong-counter>
      <weak-counter name="c5" 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.VO
LATILE);
builder.addStrongCounter().name("c3").initialValue(3).upperBound(5).storage(Storage.PE
RSISTENT);
builder.addStrongCounter().name("c4").initialValue(4).lowerBound(0).upperBound(10).sto
rage(Storage.VOLATILE);
builder.addWeakCounter().name("c5").initialValue(5).concurrencyLevel(1).storage(Storag
e.PERSISTENT);

```

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).storage(Storage.VOLATILE).build());
manager.defineCounter("c3", CounterConfiguration.builder(CounterType.BOUNDED_STRONG)
    .initialValue(3).upperBound(5).storage(Storage.PERSISTENT).build());
manager.defineCounter("c4", CounterConfiguration.builder(CounterType.BOUNDED_STRONG)
    .initialValue(4).lowerBound(0).upperBound(10).storage(Storage.VOLATILE).build());
manager.defineCounter("c2", CounterConfiguration.builder(CounterType.WEAK)
    .initialValue(5).concurrencyLevel(1).storage(Storage.PERSISTENT).build());
```



`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", ...);
}
```

Per cluster attributes:

- `num-owners`: Sets the number of counter's copies to keep cluster-wide. A smaller number will make update operations faster but will support a lower number of server crashes. It **must be positive** and its default value is `2`.
- `reliability`: Sets the counter's update behavior in a network partition. Default value is `AVAILABLE` and valid values are:
 - `AVAILABLE`: all partitions are able to read and update the counter's value.
 - `CONSISTENT`: only the primary partition (majority of nodes) will be able to read and update the counter's value. The remaining partitions can only read its value.

Per counter attributes:

- `initial-value` [common]: Sets the counter's initial value. Default is `0` (zero).
- `storage` [common]: Sets the counter's behavior when the cluster is shutdown and restarted. Default value is `VOLATILE` and valid values are:
 - `VOLATILE`: the counter's value is only available in memory. The value will be lost when a cluster is shutdown.
 - `PERSISTENT`: the counter's value is stored in a private and local persistent store. The value is kept when the cluster is shutdown and restored after a restart.



On-demand and **VOLATILE** counters will lose its value and configuration after a cluster shutdown. They must be defined again after the restart.

- **lower-bound** [strong]: Sets the strong consistent counter's lower bound. Default value is `Long.MIN_VALUE`.
- **upper-bound** [strong]: Sets the strong consistent counter's upper bound. Default value is `Long.MAX_VALUE`.



If neither the **lower-bound** or **upper-bound** are configured, the strong counter is set as unbounded.



The **initial-value** must be between **lower-bound** and **upper-bound** inclusive.

- **concurrency-level** [weak]: Sets the number of concurrent updates. Its value **must be positive** and the default value is **16**.

12.1.1. List counter names

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

12.2. The **CounterManager** interface.

The **CounterManager** interface is the entry point to define, retrieve and remove a counter. It 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);
```

For Hot Rod client, the **CounterManager** is registered in the **RemoteCacheManager** and it can be retrieved like:

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

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



Hot Rod messages format can be found in [Hot Rod Protocol 2.7](#)

12.2.1. Remove a counter via CounterManager



use with caution.

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.



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

12.3. 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.



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

12.3.1. The `StrongCounter` interface: when the consistency or bounds matters.

The strong counter provides uses a single key stored in {brandname} 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");
```



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.



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



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.

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();
```

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)

Usage Examples

```

StrongCounter counter = counterManager.getStrongCounter("unbounded_coutner");

// 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;
}).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);
```

12.3.2. The WeakCounter interface: when speed is needed

The **WeakCounter** stores the counter's value in multiple keys in {brandname} 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. Its main advantage over the **StrongCounter** is the lower contention in the cache. On the other hand, the read of its value is more expensive and bounds are not allowed.



The reset operation should be handled with caution. It is **not** atomic and it produces intermediate values. These values 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");
```

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.

Uses cases

The weak counter fits best in use 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.

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().get());

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());
```

12.4. 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 registerer by the following method:

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

The `CounterLister` 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();  
}
```



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.



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

Chapter 13. Clustered Lock

A *clustered lock* is a lock which is distributed and shared among all nodes in the {brandname} cluster and currently provides a way to execute code that will be synchronized between the nodes in a given cluster.

13.1. Installation

In order to start using the clustered locks, you need to add the dependency in your Maven `pom.xml` file:

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-lock</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

13.2. ClusteredLock Configuration

Currently there is a single type of `ClusteredLock` supported : non reentrant, NODE ownership lock.

13.2.1. Ownership

- **NODE** When a `ClusteredLock` is defined, this lock can be used from all the nodes in the {brandname} cluster. When the ownership is NODE type, this means that the owner of the lock is the {brandname} node that acquired the lock at a given time. This means that each time we get a `ClusteredLock` instance with the `ClusteredCacheManager`, this instance will be the same instance for each {brandname} node. This lock can be used to synchronize code between {brandname} nodes. The advantage of this lock is that any thread in the node can release the lock at a given time.
- **INSTANCE** - not yet supported

When a `ClusteredLock` is defined, this lock can be used from all the nodes in the {brandname} cluster. When the ownership is INSTANCE type, this means that the owner of the lock is the actual instance we acquired when `ClusteredLockManager.get("lockName")` is called.

This means that each time we get a `ClusteredLock` instance with the `ClusteredCacheManager`, this instance will be a new instance. This lock can be used to synchronize code between {brandname} nodes and inside each {brandname} node. The advantage of this lock is that only the instance that called 'lock' can release the lock.

13.2.2. Reentrancy

When a `ClusteredLock` is configured reentrant, the owner of the lock can reacquire the lock as many consecutive times as it wants while holding the lock.

Currently, only non reentrant locks are supported. This means that when two consecutive `lock` calls are sent for the same owner, the first call will acquire the lock if it's available, and the second call will block.

13.3. ClusteredLockManager Interface

The `ClusteredLockManager` interface, **marked as experimental**, is the entry point to define, retrieve and remove a lock. It automatically listen to the creation of `EmbeddedCacheManager` and proceeds with the registration of an instance of it per `EmbeddedCacheManager`. It starts the internal caches needed to store the lock state.

Retrieving the `ClusteredLockManager` is as simple as invoking the `EmbeddedClusteredLockManagerFactory.from(EmbeddedCacheManager)` as shown in the example below:

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

// retrieve the ClusteredLockManager
ClusteredLockManager clusteredLockManager = ClusteredLockManagerFactory.from(manager);
```

```
@Experimental
public interface ClusteredLockManager {

    boolean defineLock(String name);

    boolean defineLock(String name, ClusteredLockConfiguration configuration);

    ClusteredLock get(String name);

    ClusteredLockConfiguration getConfiguration(String name);

    boolean isDefined(String name);

    CompletableFuture<Boolean> remove(String name);

    CompletableFuture<Boolean> forceRelease(String name);
}
```

- `defineLock` : Defines a lock with the specified name and the default `ClusteredLockConfiguration`. It does not overwrite existing configurations.
- `defineLock(String name, ClusteredLockConfiguration configuration)` : Defines a lock with the specified name and `ClusteredLockConfiguration`. It does not overwrite existing configurations.

- `ClusteredLock get(String name)` : Get's a `ClusteredLock` by it's name. A call of `defineLock` must be done at least once in the cluster. See [ownership](#) level section to understand the implications of `get` method call.

Currently, the only ownership level supported is **NODE**.

- `ClusteredLockConfiguration getConfiguration(String name)` :

Returns the configuration of a `ClusteredLock`, if such exists.

- `boolean isDefined(String name)` : Checks if a lock is already defined.
- `CompletableFuture<Boolean> remove(String name)` : Removes a `ClusteredLock` if such exists.
- `CompletableFuture<Boolean> forceRelease(String name)` : Releases - or unlocks - a `ClusteredLock`, if such exists, no matter who is holding it at a given time. Calling this method may cause concurrency issues and has to be used in **exceptional situations**.

13.4. ClusteredLock Interface

`ClusteredLock` interface, **marked as experimental**, is the interface that implements the clustered locks.

```
@Experimental
public interface ClusteredLock {

    CompletableFuture<Void> lock();

    CompletableFuture<Boolean> tryLock();

    CompletableFuture<Boolean> tryLock(long time, TimeUnit unit);

    CompletableFuture<Void> unlock();

    CompletableFuture<Boolean> isLocked();

    CompletableFuture<Boolean> isLockedByMe();
}
```

- **lock** : Acquires the lock. If the lock is not available then call blocks until the lock is acquired. Currently, **there is no maximum time specified for a lock request to fail**, so this could cause thread starvation.
- **tryLock** Acquires the lock only if it is free at the time of invocation, and returns **true** in that case. This method does not block (or wait) for any lock acquisition.
- **tryLock(long time, TimeUnit unit)** If the lock is available this method returns immediately with **true**. If the lock is not available then the call waits until :
 - The lock is acquired
 - The specified waiting time elapses

If the time is less than or equal to zero, the method will not wait at all.

- **unlock**

Releases the lock. Only the holder of the lock may release the lock.

- **isLocked** Returns **true** when the lock is locked and **false** when the lock is released.
- **isLockedByMe** Returns **true** when the lock is owned by the caller and **false** when the lock is owned by someone else or it's released.

13.4.1. Usage Examples

```
EmbeddedCache cm = ...;
ClusteredLockManager cclm = ClusteredLockManagerFactory.from(cm);

lock.tryLock()
  .thenCompose(result -> {
    if (result) {
      try {
        // manipulate protected state
      } finally {
        return lock.unlock();
      }
    } else {
      // Do something else
    }
  });
}
```

13.5. ClusteredLockManager Configuration

You can configure **ClusteredLockManager** to use different strategies for locks, either declaratively or programmatically, with the following attributes:

num-owners

Defines the total number of nodes in each cluster that store the states of clustered locks. The default value is **-1**, which replicates the value to all nodes.

reliability

Controls how clustered locks behave when clusters split into partitions or multiple nodes leave a cluster. You can set the following values:

- **AVAILABLE**: Nodes in any partition can concurrently operate on locks.
- **CONSISTENT**: Only nodes that belong to the majority partition can operate on locks. This is the default value.

The following is an example declarative configuration for **ClusteredLockManager**:

```

<?xml version="1.0" encoding="UTF-8"?>
<infinispan
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:infinispan:config:${infinispan.core.schema.version}
http://www.infinispan.org/schemas/infinispan-config-10.0.xsd"
  xmlns="urn:infinispan:config:10.0">
  ...
  <cache-container default-cache="default">
    <transport/>
    <local-cache name="default">
      <locking concurrency-level="100" acquire-timeout="1000"/>
    </local-cache>

    <clustered-locks xmlns="urn:infinispan:config:clustered-locks:10.0"
      num-owners = "3"
      reliability="AVAILABLE">
      <clustered-lock name="lock1" />
      <clustered-lock name="lock2" />
    </clustered-locks>
  </cache-container>
  ...
</infinispan>

```

Chapter 14. Multimap Cache

MultimapCache is a type of {brandname} Cache that maps keys to values in which each key can contain multiple values.

14.1. Installation and configuration

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-multimap</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

14.2. MultimapCache API

MultimapCache API exposes several methods to interact with the Multimap Cache. All these methods are non-blocking in most of the cases. See [limitations]

```
public interface MultimapCache<K, V> {

    CompletableFuture<Void> put(K key, V value);

    CompletableFuture<Collection<V>> get(K key);

    CompletableFuture<Boolean> remove(K key);

    CompletableFuture<Boolean> remove(K key, V value);

    CompletableFuture<Void> remove(Predicate<? super V> p);

    CompletableFuture<Boolean> containsKey(K key);

    CompletableFuture<Boolean> containsValue(V value);

    CompletableFuture<Boolean> containsEntry(K key, V value);

    CompletableFuture<Long> size();

    boolean supportsDuplicates();

}
```


14.2.1. `CompletableFuture<Void> put(K key, V value)`

Puts a key-value pair in the multimap cache.

```
MultimapCache<String, String> multimapCache = ...;

multimapCache.put("girlNames", "marie")
    .thenCompose(r1 -> multimapCache.put("girlNames", "oihana"))
    .thenCompose(r3 -> multimapCache.get("girlNames"))
    .thenAccept(names -> {
        if(names.contains("marie"))
            System.out.println("Marie is a girl name");

        if(names.contains("oihana"))
            System.out.println("Oihana is a girl name");
    });
```

The output of this code is :

```
Marie is a girl name
Oihana is a girl name
```

14.2.2. `CompletableFuture<Collection<V>> get(K key)`

Asynchronous that returns a view collection of the values associated with key in this multimap cache, if any. Any changes to the retrieved collection won't change the values in this multimap cache. When this method returns an empty collection, it means the key was not found.

14.2.3. `CompletableFuture<Boolean> remove(K key)`

Asynchronous that removes the entry associated with the key from the multimap cache, if such exists.

14.2.4. `CompletableFuture<Boolean> remove(K key, V value)`

Asynchronous that removes a key-value pair from the multimap cache, if such exists.

14.2.5. `CompletableFuture<Void> remove(Predicate<? super V> p)`

Asynchronous method. Removes every value that match the given predicate.

14.2.6. `CompletableFuture<Boolean> containsKey(K key)`

Asynchronous that returns true if this multimap contains the key.

14.2.7. `CompletableFuture<Boolean> containsValue(V value)`

Asynchronous that returns true if this multimap contains the value in at least one key.

14.2.8. `CompletableFuture<Boolean> containsEntry(K key, V value)`

Asynchronous that returns true if this multimap contains at least one key-value pair with the value.

14.2.9. `CompletableFuture<Long> size()`

Asynchronous that returns the number of key-value pairs in the multimap cache. It doesn't return the distinct number of keys.

14.2.10. `boolean supportsDuplicates()`

Asynchronous that returns true if the multimap cache supports duplicates. This means that the content of the multimap can be 'a' → ['1', '1', '2']. For now this method will always return false, as duplicates are not yet supported. The existence of a given value is determined by 'equals' and 'hashCode' method's contract.

14.3. Creating a Multimap Cache

Currently the MultimapCache is configured as a regular cache. This can be done either by code or XML configuration. See how to configure a regular Cache in the section link to [\[configure a cache\]](#).

14.3.1. Embedded mode

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

// create or obtain a MultimapCacheManager passing the EmbeddedCacheManager
MultimapCacheManager multimapCacheManager = EmbeddedMultimapCacheManagerFactory.from
(cm);

// define the configuration for the multimap cache
multimapCacheManager.defineConfiguration(multimapCacheName, c.build());

// get the multimap cache
multimapCache = multimapCacheManager.get(multimapCacheName);
```

14.3.2. Server mode

TODO

14.4. Limitations

In almost every case the Multimap Cache will behave as a regular Cache, but some limitations exist

in the current version.

14.4.1. Support for duplicates

Duplicates are not supported yet. This means that the multimap won't contain any duplicate key-value pair. Whenever put method is called, if the key-value pair already exist, this key-value pair won't be added. Methods used to check if a key-value pair is already present in the Multimap are the `equals` and `hashCode`.

14.4.2. Eviction

For now, the eviction works per key, and not per key-value pair. This means that whenever a key is evicted, all the values associated with the key will be evicted too. Eviction per key-value could be supported in the future.

14.4.3. Transactions

Implicit transactions are supported through the auto-commit and all the methods are non blocking. Explicit transactions work without blocking in most of the cases. Methods that will block are `size`, `containsEntry` and `remove(Predicate<? super V> p)`

Chapter 15. CDI Support

{brandname} includes integration with [Contexts and Dependency Injection](#) (better known as CDI) via {brandname}'s `infinispan-cdi-embedded` or `infinispan-cdi-remote` module. CDI is part of [Java EE specification](#) and aims for managing beans' lifecycle inside the container. The integration allows to inject Cache interface and bridge Cache and CacheManager events. JCache annotations (JSR-107) are supported by `infinispan-jcache` and `infinispan-jcache-remote` artifacts. For more information have a look at [Chapter 11](#) of the JCACHE specification.

15.1. Maven Dependencies

To include CDI support for {brandname} in your project, use one of the following dependencies:

pom.xml for Embedded mode

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cdi-embedded</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

pom.xml for Remote mode

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cdi-remote</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.



Which version of {brandname} should I use?

We recommend using the latest final version {brandname}.

15.2. Embedded cache integration

15.2.1. Inject an embedded cache

By default you can inject the default {brandname} cache. Let's look at the following example:

Default cache injection

```
...
import javax.inject.Inject;

public class GreetingService {

    @Inject
    private Cache<String, String> cache;

    public String greet(String user) {
        String cachedValue = cache.get(user);
        if (cachedValue == null) {
            cachedValue = "Hello " + user;
            cache.put(user, cachedValue);
        }
        return cachedValue;
    }
}
```

If you want to use a specific cache rather than the default one, you just have to provide your own cache configuration and cache qualifier. See example below:

Qualifier example

```
...
import javax.inject.Qualifier;

@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GreetingCache {
}
```

```
...
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.cdi.ConfigureCache;
import javax.enterprise.inject.Produces;

public class Config {

    @ConfigureCache("greeting-cache") // This is the cache name.
    @GreetingCache // This is the cache qualifier.
    @Produces
    public Configuration greetingCacheConfiguration() {
        return new ConfigurationBuilder()
            .memory()
            .size(1000)
            .build();
    }

    // The same example without providing a custom configuration.
    // In this case the default cache configuration will be used.
    @ConfigureCache("greeting-cache")
    @GreetingCache
    @Produces
    public Configuration greetingCacheConfiguration;
}
```

To use this cache in the GreetingService add the `@GreetingCache` qualifier on your cache injection point.

15.2.2. Override the default embedded cache manager and configuration

You can override the default cache configuration used by the default `EmbeddedCacheManager`. For that, you just have to create a `Configuration` producer with default qualifiers as illustrated in the following snippet:

```
public class Config {  
  
    // By default CDI adds the @Default qualifier if no other qualifier is provided.  
    @Produces  
    public Configuration defaultEmbeddedCacheConfiguration() {  
        return new ConfigurationBuilder()  
            .memory()  
            .size(100)  
            .build();  
    }  
}
```

It's also possible to override the default `EmbeddedCacheManager`. The newly created manager must have default qualifiers and Application scope.

Overriding EmbeddedCacheManager

```
...  
import javax.enterprise.context.ApplicationScoped;  
  
public class Config {  
  
    @Produces  
    @ApplicationScoped  
    public EmbeddedCacheManager defaultEmbeddedCacheManager() {  
        return new DefaultCacheManager(new ConfigurationBuilder()  
            .memory()  
            .size(100)  
            .build());  
    }  
}
```

15.2.3. Configure the transport for clustered use

To use {brandname} in a clustered mode you have to configure the transport with the `GlobalConfiguration`. To achieve that override the default cache manager as explained in the previous section. Look at the following snippet:

```
...
package org.infinispan.configuration.global.GlobalConfigurationBuilder;

@Produces
@ApplicationScoped
public EmbeddedCacheManager defaultClusteredCacheManager() {
    return new DefaultCacheManager(
        new GlobalConfigurationBuilder().transport().defaultTransport().build(),
        new ConfigurationBuilder().memory().size(7).build()
    );
}
```

15.3. Remote cache integration

15.3.1. Inject a remote cache

With the CDI integration it's also possible to use a `RemoteCache` as illustrated in the following snippet:

Injecting RemoteCache

```
public class GreetingService {

    @Inject
    private RemoteCache<String, String> cache;

    public String greet(String user) {
        String cachedValue = cache.get(user);
        if (cachedValue == null) {
            cachedValue = "Hello " + user;
            cache.put(user, cachedValue);
        }
        return cachedValue;
    }
}
```

If you want to use another cache, for example the greeting-cache, add the `@Remote` qualifier on the cache injection point which contains the cache name.


```
public class GreetingService {  
  
    @Inject  
    @Remote("greeting-cache")  
    private RemoteCache<String, String> cache;  
  
    ...  
}
```

Adding the `@Remote` cache qualifier on each injection point might be error prone. That's why the remote cache integration provides another way to achieve the same goal. For that you have to create your own qualifier annotated with `@Remote`:

RemoteCache qualifier

```
@Remote("greeting-cache")  
@Qualifier  
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
public @interface RemoteGreetingCache {  
}
```

To use this cache in the `GreetingService` add the qualifier `@RemoteGreetingCache` qualifier on your cache injection.

15.3.2. Override the default remote cache manager

Like the embedded cache integration, the remote cache integration comes with a default remote cache manager producer. This default `RemoteCacheManager` can be overridden as illustrated in the following snippet:

Overriding default RemoteCacheManager

```
public class Config {  
  
    @Produces  
    @ApplicationScoped  
    public RemoteCacheManager defaultRemoteCacheManager() {  
        return new RemoteCacheManager(localhost, 1544);  
    }  
}
```

15.4. Use a custom remote/embedded cache manager for one or more cache

It's possible to use a custom cache manager for one or more cache. You just need to annotate the cache manager producer with the cache qualifiers. Look at the following example:

```
public class Config {

    @GreetingCache
    @Produces
    @ApplicationScoped
    public EmbeddedCacheManager specificEmbeddedCacheManager() {
        return new DefaultCacheManager(new ConfigurationBuilder()
            .expiration()
            .lifespan(60000L)
            .build());
    }

    @RemoteGreetingCache
    @Produces
    @ApplicationScoped
    public RemoteCacheManager specificRemoteCacheManager() {
        return new RemoteCacheManager("localhost", 1544);
    }
}
```

With the above code the `GreetingCache` or the `RemoteGreetingCache` will be associated with the produced cache manager.



Producer method scope

To work properly the producers must have the scope `@ApplicationScoped`. Otherwise each injection of cache will be associated to a new instance of cache manager.

15.5. Use JCache caching annotations



There is now a separate module for JSR 107 (JCACHE) integration, including API. See [this chapter](#) for details.

When CDI integration and JCache artifacts are present on the classpath, it is possible to use JCache annotations with CDI managed beans. These annotations provide a simple way to handle common use cases. The following caching annotations are defined in this specification:

- `@CacheResult` - caches the result of a method call
- `@CachePut` - caches a method parameter
- `@CacheRemoveEntry` - removes an entry from a cache

- `@CacheRemoveAll` - removes all entries from a cache



Annotations target type

These annotations must only be used on methods.

To use these annotations, proper interceptors need to be declared in `beans.xml` file:

Interceptors for managed environments such as Application Servers

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
  version="1.2" bean-discovery-mode="annotated">

  <interceptors>
    <class>org.infinispan.jcache.annotation.InjectedExceptionInterceptor</class>
    <class>org.infinispan.jcache.annotation.InjectedExceptionPutInterceptor</class>
    <class>
org.infinispan.jcache.annotation.InjectedExceptionRemoveEntryInterceptor</class>
    <class>org.infinispan.jcache.annotation.InjectedExceptionRemoveAllInterceptor</class>
  </interceptors>
</beans>
```

Interceptors for unmanaged environments such as standalone applications

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
  version="1.2" bean-discovery-mode="annotated">

  <interceptors>
    <class>org.infinispan.jcache.annotation.CacheResultInterceptor</class>
    <class>org.infinispan.jcache.annotation.CachePutInterceptor</class>
    <class>org.infinispan.jcache.annotation.CacheRemoveEntryInterceptor</class>
    <class>org.infinispan.jcache.annotation.CacheRemoveAllInterceptor</class>
  </interceptors>
</beans>
```

The following snippet of code illustrates the use of `@CacheResult` annotation. As you can see it simplifies the caching of the `GreetingService#greet` method results.

```
import javax.cache.interceptor.CacheResult;

public class GreetingService {

    @CacheResult
    public String greet(String user) {
        return "Hello" + user;
    }
}
```

The first version of the `GreetingService` and the above version have exactly the same behavior. The only difference is the cache used. By default it's the fully qualified name of the annotated method with its parameter types (e.g. `org.infinispan.example.GreetingService.greet(java.lang.String)`).

Using other cache than default is rather simple. All you need to do is to specify its name with the `cacheName` attribute of the cache annotation. For example:

Specifying cache name for JCache

```
@CacheResult(cacheName = "greeting-cache")
```

15.6. Use Cache events and CDI

It is possible to receive Cache and Cache Manager level events using CDI Events. You can achieve it using `@Observes` annotation as shown in the following snippet:

Event listeners based on CDI

```
import javax.enterprise.event.Observes;
import org.infinispan.notifications.cachemanagerlistener.event.CacheStartedEvent;
import org.infinispan.notifications.cachelistener.event.*;

public class GreetingService {

    // Cache level events
    private void entryRemovedFromCache(@Observes CacheEntryCreatedEvent event) {
        ...
    }

    // Cache Manager level events
    private void cacheStarted(@Observes CacheStartedEvent event) {
        ...
    }
}
```



Check [Listeners and Notifications](#) for more information about event types.

Chapter 16. JCache (JSR-107) provider

{brandname} provides an implementation of JCache 1.0 API ([JSR-107](#)). JCache specifies a standard Java API for caching temporary Java objects in memory. Caching java objects can help get around bottlenecks arising from using data that is expensive to retrieve (i.e. DB or web service), or data that is hard to calculate. Caching these type of objects in memory can help speed up application performance by retrieving the data directly from memory instead of doing an expensive roundtrip or recalculation. This document specifies how to use JCache with {brandname}'s implementation of the specification, and explains key aspects of the API.

16.1. Dependencies

In order to start using {brandname} JCache implementation, a single dependency needs to be added to the Maven pom.xml file:

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-jcache</artifactId>
  <version>${version.infinispan}</version>
  <scope>test</scope>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

16.2. Create a local cache

Creating a local cache, using default configuration options as defined by the JCache API specification, is as simple as doing the following:

```
import javax.cache.*;
import javax.cache.configuration.*;

// Retrieve the system wide cache manager
CacheManager cacheManager = Caching.getCachingProvider().getCacheManager();
// Define a named cache with default JCache configuration
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
```



By default, the JCache API specifies that data should be stored as `storeByValue`, so that object state mutations outside of operations to the cache, won't have an impact in the objects stored in the cache. {brandname} has so far implemented this using serialization/marshalling to make copies to store in the cache, and that way adhere to the spec. Hence, if using default JCache configuration with {brandname}, data stored must be marshallable.

Alternatively, JCache can be configured to store data by reference (just like {brandname} or JDK Collections work). To do that, simply call:

```
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>().setStoreByValue(false));
```

16.3. Create a remote cache

Creating a remote cache (client-server mode), using default configuration options as defined by the JCache API specification, is as simple as doing the following:

```
import javax.cache.*;
import javax.cache.configuration.*;

// Retrieve the system wide cache manager via
org.infinispan.jcache.remote.JCachingProvider
CacheManager cacheManager = Caching.getCachingProvider(
    "org.infinispan.jcache.remote.JCachingProvider").getCacheManager();
// Define a named cache with default JCache configuration
Cache<String, String> cache = cacheManager.createCache("remoteNamedCache",
    new MutableConfiguration<String, String>());
```



In order to use the `org.infinispan.jcache.remote.JCachingProvider`, `infinispan-jcache-remote-<version>.jar` and all its transitive dependencies need to be on your classpath.

16.4. Store and retrieve data

Even though JCache API does not extend neither `java.util.Map` nor `java.util.concurrent.ConcurrentMap`, it provides a key/value API to store and retrieve data:

```
import javax.cache.*;
import javax.cache.configuration.*;

CacheManager cacheManager = Caching.getCacheManager();
Cache<String, String> cache = cacheManager.createCache("namedCache",
    new MutableConfiguration<String, String>());
cache.put("hello", "world"); // Notice that javax.cache.Cache.put(K) returns void!
String value = cache.get("hello"); // Returns "world"
```

Contrary to standard `java.util.Map`, `javax.cache.Cache` comes with two basic put methods called `put` and `getAndPut`. The former returns `void` whereas the latter returns the previous value associated with the key. So, the equivalent of `java.util.Map.put(K)` in JCache is `javax.cache.Cache.getAndPut(K)`.



Even though JCache API only covers standalone caching, it can be plugged with a persistence store, and has been designed with clustering or distribution in mind. The reason why `javax.cache.Cache` offers two put methods is because standard `java.util.Map` put call forces implementors to calculate the previous value. When a persistent store is in use, or the cache is distributed, returning the previous value could be an expensive operation, and often users call standard `java.util.Map.put(K)` without using the return value. Hence, JCache users need to think about whether the return value is relevant to them, in which case they need to call `javax.cache.Cache.getAndPut(K)`, otherwise they can call `java.util.Map.put(K, V)` which avoids returning the potentially expensive operation of returning the previous value.

16.5. Comparing `java.util.concurrent.ConcurrentMap` and `javax.cache.Cache` APIs

Here's a brief comparison of the data manipulation APIs provided by `java.util.concurrent.ConcurrentMap` and `javax.cache.Cache` APIs.

Operation	<code>java.util.concurrent.ConcurrentMap<K, V></code>	<code>javax.cache.Cache<K, V></code>
store and no return	N/A	<code>void put(K key)</code>
store and return previous value	<code>V put(K key)</code>	<code>V getAndPut(K key)</code>
store if not present	<code>V putIfAbsent(K key, V value)</code>	<code>boolean putIfAbsent(K key, V value)</code>
retrieve	<code>V get(Object key)</code>	<code>V get(K key)</code>
delete if present	<code>V remove(Object key)</code>	<code>boolean remove(K key)</code>
delete and return previous value	<code>V remove(Object key)</code>	<code>V getAndRemove(K key)</code>
delete conditional	<code>boolean remove(Object key, Object value)</code>	<code>boolean remove(K key, V oldValue)</code>
replace if present	<code>V replace(K key, V value)</code>	<code>boolean replace(K key, V value)</code>
replace and return previous value	<code>V replace(K key, V value)</code>	<code>V getAndReplace(K key, V value)</code>
replace conditional	<code>boolean replace(K key, V oldValue, V newValue)</code>	<code>boolean replace(K key, V oldValue, V newValue)</code>

Comparing the two APIs, it's obvious to see that, where possible, JCache avoids returning the previous value to avoid operations doing expensive network or IO operations. This is an overriding principle in the design of JCache API. In fact, there's a set of operations that are present in `java.util.concurrent.ConcurrentMap`, but are not present in the `javax.cache.Cache` because they could be expensive to compute in a distributed cache. The only exception is iterating over the contents of the cache:

Operation	java.util.concurrent.Concurrent tMap<K, V>	javax.cache.Cache<K, V>
calculate size of cache	int size()	N/A
return all keys in the cache	Set<K> keySet()	N/A
return all values in the cache	Collection<V> values()	N/A
return all entries in the cache	Set<Map.Entry<K, V>> entrySet()	N/A
iterate over the cache	use iterator() method on keySet, values or entrySet	Iterator<Cache.Entry<K, V>> iterator()

16.6. Clustering JCache instances

{brandname} JCache implementation goes beyond the specification in order to provide the possibility to cluster caches using the standard API. Given a {brandname} configuration file configured to replicate caches like this:

infinispan.xml

```
<infinispan>
  <cache-container default-cache="namedCache">
    <transport cluster="jcache-cluster" />
    <replicated-cache name="namedCache" />
  </cache-container>
</infinispan>
```

You can create a cluster of caches using this code:


```

import javax.cache.*;
import java.net.URI;

// For multiple cache managers to be constructed with the standard JCache API
// and live in the same JVM, either their names, or their classloaders, must
// be different.
// This example shows how to force their classloaders to be different.
// An alternative method would have been to duplicate the XML file and give
// it a different name, but this results in unnecessary file duplication.
ClassLoader tccl = Thread.currentThread().getContextClassLoader();
CacheManager cacheManager1 = Caching.getCachingProvider().getCacheManager(
    URI.create("infinispan-jcache-cluster.xml"), new TestClassLoader(tccl));
CacheManager cacheManager2 = Caching.getCachingProvider().getCacheManager(
    URI.create("infinispan-jcache-cluster.xml"), new TestClassLoader(tccl));

Cache<String, String> cache1 = cacheManager1.getCache("namedCache");
Cache<String, String> cache2 = cacheManager2.getCache("namedCache");

cache1.put("hello", "world");
String value = cache2.get("hello"); // Returns "world" if clustering is working

// --

public static class TestClassLoader extends ClassLoader {
    public TestClassLoader(ClassLoader parent) {
        super(parent);
    }
}

```

Chapter 17. Management Tooling

Management of {brandname} instances is all about exposing as much relevant statistical information that allows administrators to get a view of the state of each {brandname} instance. Taking in account that a single installation could be made up of several tens or hundreds {brandname} instances, providing clear and concise information in an efficient manner is imperative. The following sections dive into the range of management tooling that {brandname} provides.

17.1. JMX

Over the years, [JMX](#) has become the de facto standard for management and administration of middleware and as a result, the {brandname} team has decided to standardize on this technology for the exposure of management and statistical information.

17.1.1. Understanding The Exposed MBeans

By connecting to the VM(s) where {brandname} is running with a standard JMX GUI such as [JConsole](#) or [VisualVM](#) you should find the following MBeans:

- For CacheManager level JMX statistics, without further configuration, you should see an MBean called `org.infinispan:type=CacheManager,name="DefaultCacheManager"` with [properties specified by the CacheManager MBean](#).
- Using the `cacheManagerName` attribute in `globalJmxStatistics` XML element, or using the corresponding `GlobalJmxStatisticsConfigurationBuilder.cacheManagerName(String cacheManagerName)` call, you can name the cache manager in such way that the name is used as part of the JMX object name. So, if the name had been "Hibernate2LC", the JMX name for the cache manager would have been: `org.infinispan:type=CacheManager,name="Hibernate2LC"`. This offers a nice and clean way to manage environments where multiple cache managers are deployed, which follows [JMX best practices](#).
- For Cache level JMX statistics, you should see several different MBeans depending on which configuration options have been enabled. For example, if you have configured a write behind cache store, you should see an MBean exposing properties belonging to the cache store component. All Cache level MBeans follow the same format though which is the following: `org.infinispan:type=Cache,name="{name-of-cache}({cache-mode})",manager="{name-of-cache-manager}",component={component-name}` where:
 - `{name-of-cache}` has been substituted by the actual cache name. If this cache represents the default cache, its name will be `__defaultCache`.
 - `{cache-mode}` has been substituted by the cache mode of the cache. The cache mode is represented by the lower case version of the possible enumeration values shown [here](#).
 - `{name-of-cache-manager}` has been substituted by the name of the cache manager to which this cache belongs. The name is derived from the `cacheManagerName` attribute value in `globalJmxStatistics` element.
 - `{component-name}` has been substituted by one of the JMX component names in the [JMX reference documentation](#).

For example, the cache store JMX component MBean for a default cache configured with synchronous distribution would have the following name: `org.infinispan:type=Cache,name="___defaultcache(dist_sync)",manager="DefaultCacheManager",component=CacheStore`

Please note that cache and cache manager names are quoted to protect against illegal characters being used in these user-defined names.

17.1.2. Enabling JMX Statistics

The MBeans mentioned in the previous section are always created and registered in the MBeanServer allowing you to manage your caches but some of their attributes do not expose meaningful values unless you take the extra step of enabling collection of statistics. Gathering and reporting statistics via JMX can be enabled at 2 different levels:

CacheManager level

The CacheManager is the entity that governs all the cache instances that have been created from it. Enabling CacheManager statistics collections differs depending on the configuration style:

- If configuring the CacheManager via XML, make sure you add the following XML under the `<cache-container />` element:

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

- If configuring the CacheManager programmatically, simply add the following code:

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...
globalConfigurationBuilder.globalJmxStatistics().enable();
```

Cache level

At this level, you will receive management information generated by individual cache instances. Enabling Cache statistics collections differs depending on the configuration style:

- If configuring the Cache via XML, make sure you add the following XML under the one of the top level cache elements, such as `<local-cache />`:

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

- If configuring the Cache programmatically, simply add the following code:

```
ConfigurationBuilder configurationBuilder = ...
configurationBuilder.jmxStatistics().enable();
```

17.1.3. Monitoring cluster health

It is also possible to monitor {brandname} cluster health using JMX. On CacheManager there's an additional object called `CacheContainerHealth`. It contains the following attributes:

- `cacheHealth` - a list of caches and corresponding statuses (HEALTHY, UNHEALTHY or REBALANCING)
- `clusterHealth` - overall cluster health
- `clusterName` - cluster name
- `freeMemoryKb` - Free memory obtained from JVM runtime measured in KB
- `numberOfCpus` - The number of CPUs obtained from JVM runtime
- `numberOfNodes` - The number of nodes in the cluster
- `totalMemoryKb` - Total memory obtained from JVM runtime measured in KB

17.1.4. Multiple JMX Domains

There can be situations where several CacheManager instances are created in a single VM, or Cache names belonging to different CacheManagers under the same VM clash.

Using different JMX domains for multi cache manager environments should be last resort. Instead, it's possible to name a cache manager in such way that it can easily be identified and used by monitoring tools. For example:

- Via XML:

```
<cache-container statistics="true" name="Hibernate2LC"/>
```

- Programmatically:

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...
globalConfigurationBuilder.globalJmxStatistics()
    .enable()
    .cacheManagerName("Hibernate2LC");
```

Using either of these options should result on the CacheManager MBean name being: `org.infinispan:type=CacheManager,name="Hibernate2LC"`

For the time being, you can still set your own `jmxDomain` if you need to and we also allow duplicate domains, or rather duplicate JMX names, but these should be limited to very special cases where different cache managers within the same JVM are named equally.

17.1.5. Registering MBeans In Non-Default MBean Servers

Let's discuss where {brandname} registers all these MBeans. By default, {brandname} registers them in the [standard JVM MBeanServer platform](#). However, users might want to register these

MBeans in a different MBeanServer instance. For example, an application server might work with a different MBeanServer instance to the default platform one. In such cases, users should implement the [MBeanServerLookup interface](#) provided by {brandname} so that the `getMBeanServer()` method returns the MBeanServer under which {brandname} should register the management MBeans. Once you have your implementation ready, simply configure {brandname} with the fully qualified name of this class. For example:

- Via XML:

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

- Programmatically:

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...
globalConfigurationBuilder.globalJmxStatistics()
    .enable()
    .mBeanServerLookup(new com.acme.MyMBeanServerLookup());
```

17.1.6. Available MBeans

For a complete list of available MBeans, refer to the [JMX reference documentation](#)

17.2. Command-Line Interface (CLI)

{brandname} offers a simple Command-Line Interface (CLI) with which it is possible to interact with the data within the caches and with most of the internal components (e.g. transactions, cross-site backups, rolling upgrades).

The CLI is built out of two elements: a server-side module and the client command tool. The server-side module (`infinispan-cli-server-$VERSION.jar`) provides the actual interpreter for the commands and needs to be included alongside your application. {brandname} Server includes CLI support out of the box.

Currently the server (and the client) use the JMX protocol to communicate, but in a future release we plan to support other communication protocols (in particular our own Hot Rod).

The CLI offers both an interactive and a batch mode. To invoke the client, just run the provided `bin/ispn-cli.[sh|bat]` script. The following is a list of command-line switches which affect how the CLI can be started:

```

-c, --connect=URL      connects to a running instance of Infinispan.
                        JMX over RMI
jmx://[username[:password]]@host:port[/container[/cache]]
                        JMX over JBoss remoting
remoting://[username[:password]]@host:port[/container[/cache]]
-f, --file=FILE        reads input from the specified file instead of using
                        interactive mode. If FILE is '-', then commands will be read
                        from stdin
-h, --help              shows this help page
-v, --version           shows version information

```

- JMX over RMI is the traditional way in which JMX clients connect to MBeanServers. Please refer to the [JDK Monitoring and Management](#) documentation for details on how to configure the process to be monitored
- JMX over JBoss Remoting is the protocol of choice when your {brandname} application is running within WildFly or EAP.

The connection to the application can also be initiated from within the CLI using the connect command.

```

[disconnected//]> connect jmx://localhost:12000
[jmx://localhost:12000/MyCacheManager/>

```

The CLI prompt will show the active connection information, including the currently selected CacheManager. Initially no cache is selected so, before performing any cache operations, one must be selected. For this the *cache* command is used. The CLI supports tab-completion for all commands and options and for most parameters where it makes sense to do so. Therefore typing *cache* and pressing TAB will show a list of active caches:

```

[jmx://localhost:12000/MyCacheManager/> cache
__defaultcache  namedCache
[jmx://localhost:12000/MyCacheManager/]> cache __defaultcache
[jmx://localhost:12000/MyCacheManager/__defaultcache]>

```

Pressing TAB at an empty prompt will show the list of all available commands:

alias	cache	container	encoding	get	locate	remove
site	upgrade					
abort	clearcache	create	end	help	put	replace
start	version					
begin	commit	disconnect	evict	info	quit	rollback
stats						

The CLI is based on [Aesh](#) and therefore offers many keyboard shortcuts to navigate and search the

history of commands, to manipulate the cursor at the prompt, including both Emacs and VI modes of operation.

17.2.1. Commands

abort

The *abort* command is used to abort a running batch initiated by the *start* command

```
[jmx://localhost:12000/MyCacheManager/namedCache]> start
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> abort
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
null
```

begin

The *begin* command starts a transaction. In order for this command to work, the cache(s) on which the subsequent operations are invoked must have transactions enabled.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> begin
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> commit
```

cache

The *cache* command selects the cache to use as default for all subsequent operations. If it is invoked without parameters it shows the currently selected cache.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> cache ___defaultcache
[jmx://localhost:12000/MyCacheManager/___defaultcache]> cache
___defaultcache
[jmx://localhost:12000/MyCacheManager/___defaultcache]>
```

clearcache

The *clearcache* command clears a cache from all content.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> clearcache
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
null
```

commit

The *commit* command commits an ongoing transaction

```
[jmx://localhost:12000/MyCacheManager/namedCache]> begin
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b
[jmx://localhost:12000/MyCacheManager/namedCache]> commit
```

container

The *container* command selects the default container (cache manager). Invoked without parameters it lists all available containers

```
[jmx://localhost:12000/MyCacheManager/namedCache]> container
MyCacheManager OtherCacheManager
[jmx://localhost:12000/MyCacheManager/namedCache]> container OtherCacheManager
[jmx://localhost:12000/OtherCacheManager/]>
```

create

The *create* command creates a new cache based on the configuration of an existing cache definition

```
[jmx://localhost:12000/MyCacheManager/namedCache]> create newCache like namedCache
[jmx://localhost:12000/MyCacheManager/namedCache]> cache newCache
[jmx://localhost:12000/MyCacheManager/newCache]>
```

deny

When authorization is enabled and the role mapper has been configured to be the ClusterRoleMapper, principal to role mappings are stored within the cluster registry (a replicated cache available to all nodes). The *deny* command can be used to deny roles previously assigned to a principal:

```
[remoting://localhost:9999]> deny supervisor to user1
```

disconnect

The *disconnect* command disconnects the currently active connection allowing the CLI to connect to another instance.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> disconnect
[disconnected//]
```


encoding

The *encoding* command is used to set a default codec to use when reading/writing entries from/to a cache. When invoked without arguments it shows the currently selected codec. This command is useful since currently remote protocols such as HotRod and Memcached wrap keys and values in specialized structures.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> encoding
none
[jmx://localhost:12000/MyCacheManager/namedCache]> encoding --list
memcached
hotrod
none
rest
[jmx://localhost:12000/MyCacheManager/namedCache]> encoding hotrod
```

end

The *end* command is used to successfully end a running batch initiated by the *start* command

```
[jmx://localhost:12000/MyCacheManager/namedCache]> start
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> end
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
a
```

evict

The *evict* command is used to evict from the cache the entry associated with a specific key.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> evict a
```

get

The *get* command is used to show the value associated to a specified key. For primitive types and Strings, the *get* command will simply print the default representation. For other objects, a JSON representation of the object will be printed.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
a
```

grant

When authorization is enabled and the role mapper has been configured to be the

ClusterRoleMapper, principal to role mappings are stored within the cluster registry (a replicated cache available to all nodes). The *grant* command can be used to grant new roles to a principal:

```
[remoting://localhost:9999]> grant supervisor to user1
```

info

The *info* command is used to show the configuration of the currently selected cache or container.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> info
GlobalConfiguration{asyncListenerExecutor=ExecutorFactoryConfiguration{factory=org.infinispan.executors.DefaultExecutorFactory@98add58},
asyncTransportExecutor=ExecutorFactoryConfiguration{factory=org.infinispan.executors.DefaultExecutorFactory@7bc9c14c},
evictionScheduledExecutor=ScheduledExecutorFactoryConfiguration{factory=org.infinispan.executors.DefaultScheduledExecutorFactory@7ab1a411},
replicationQueueScheduledExecutor=ScheduledExecutorFactoryConfiguration{factory=org.infinispan.executors.DefaultScheduledExecutorFactory@248a9705},
globalJmxStatistics=GlobalJmxStatisticsConfiguration{allowDuplicateDomains=true, enabled=true, jmxDomain='jboss.infinispan',
mBeanServerLookup=org.jboss.as.clustering.infinispan.MBeanServerProvider@6c0dc01,
cacheManagerName='local', properties={}},
transport=TransportConfiguration{clusterName='ISPN', machineId='null', rackId='null', siteId='null', strictPeerToPeer=false, distributedSyncTimeout=240000, transport=null,
nodeName='null', properties={}},
serialization=SerializationConfiguration{advancedExternalizers={1100=org.infinispan.server.core.CacheValue$Externalizer@5fab91d,
1101=org.infinispan.server.memcached.MemcachedValue$Externalizer@720bffd,
1104=org.infinispan.server.hotrod.ServerAddress$Externalizer@771c7eb2},
marshaller=org.infinispan.marshall.VersionAwareMarshaller@6fc21535, version=52,
classResolver=org.jboss.marshalling.ModularClassResolver@2efe83e5},
shutdown=ShutdownConfiguration{hookBehavior=DONT_REGISTER}, modules={},
site=SiteConfiguration{localSite='null'}}
```

locate

The *locate* command shows the physical location of a specified entry in a distributed cluster.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> locate a
[host/node1,host/node2]
```

put

The *put* command inserts an entry in the cache. If the cache previously contained a mapping for the key, the old value is replaced by the specified value. The user can control the type of data that the CLI will use to store the key and value. See the [Data Types](#) section.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> put b 100
[jmx://localhost:12000/MyCacheManager/namedCache]> put c 41391
[jmx://localhost:12000/MyCacheManager/namedCache]> put d true
[jmx://localhost:12000/MyCacheManager/namedCache]> put e { "package.MyClass": {"i": 5,
"x": null, "b": true } }
```

The `put` command can optionally specify a lifespan and a maximum idle time.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a expires 10s
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a expires 10m maxidle 1m
```

replace

The *replace* command replaces an existing entry in the cache. If an old value is specified, then the replacement happens only if the value in the cache coincides.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a
[jmx://localhost:12000/MyCacheManager/namedCache]> replace a b
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
b
[jmx://localhost:12000/MyCacheManager/namedCache]> replace a b c
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
c
[jmx://localhost:12000/MyCacheManager/namedCache]> replace a b d
[jmx://localhost:12000/MyCacheManager/namedCache]> get a
c
```

roles

When authorization is enabled and the role mapper has been configured to be the `ClusterRoleMapper`, principal to role mappings are stored within the cluster registry (a replicated cache available to all nodes). The *roles* command can be used to list the roles associated to a specific user, or to all users if one is not given:

```
[remoting://localhost:9999]> roles user1
[supervisor, reader]
```

rollback

The *rollback* command rolls back an ongoing transaction

```
[jmx://localhost:12000/MyCacheManager/namedCache]> begin  
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a  
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b  
[jmx://localhost:12000/MyCacheManager/namedCache]> rollback
```

site

The *site* command performs operations related to the administration of cross-site replication. It can be used to obtain information related to the status of a site and to change the status (online/offline)

```
[jmx://localhost:12000/MyCacheManager/namedCache]> site --status NYC  
online  
[jmx://localhost:12000/MyCacheManager/namedCache]> site --offline NYC  
ok  
[jmx://localhost:12000/MyCacheManager/namedCache]> site --status NYC  
offline  
[jmx://localhost:12000/MyCacheManager/namedCache]> site --online NYC
```

start

The *start* command initiates a batch of operations.

```
[jmx://localhost:12000/MyCacheManager/namedCache]> start  
[jmx://localhost:12000/MyCacheManager/namedCache]> put a a  
[jmx://localhost:12000/MyCacheManager/namedCache]> put b b  
[jmx://localhost:12000/MyCacheManager/namedCache]> end
```

stats

The *stats* command displays statistics about a cache

```
[jmx://localhost:12000/MyCacheManager/namedCache]> stats
Statistics: {
  averageWriteTime: 143
  evictions: 10
  misses: 5
  hitRatio: 1.0
  readWriteRatio: 10.0
  removeMisses: 0
  timeSinceReset: 2123
  statisticsEnabled: true
  stores: 100
  elapsedTime: 93
  averageReadTime: 14
  removeHits: 0
  numberOfEntries: 100
  hits: 1000
}
LockManager: {
  concurrencyLevel: 1000
  numberOfLocksAvailable: 0
  numberOfLocksHeld: 0
}
```

17.2.2. upgrade

The *upgrade* command performs operations used during the rolling upgrade procedure. For a detailed description of this procedure please see [Rolling Upgrades](#).

```
[jmx://localhost:12000/MyCacheManager/namedCache]> upgrade --synchronize=hotrod --all
[jmx://localhost:12000/MyCacheManager/namedCache]> upgrade --disconnectsource=hotrod
--all
```

17.2.3. version

The *version* command displays version information about both the CLI client and the server

```
[jmx://localhost:12000/MyCacheManager/namedCache]> version
Client Version 5.2.1.Final
Server Version 5.2.1.Final
```

17.2.4. Data Types

The CLI understands the following types:

- string strings can either be quoted between single (') or double (") quotes, or left unquoted. In this case it must not contain spaces, punctuation and cannot begin with a number e.g. 'a string',

key001

- **int** an integer is identified by a sequence of decimal digits, e.g. 256
- **long** a long is identified by a sequence of decimal digits suffixed by 'l', e.g. 1000l
- **double**
 - a double precision number is identified by a floating point number(with optional exponent part) and an optional 'd' suffix, e.g.3.14
- **float**
 - a single precision number is identified by a floating point number(with optional exponent part) and an 'f' suffix, e.g. 10.3f
- **boolean** a boolean is represented either by the keywords true and false
- **UUID** a UUID is represented by its canonical form XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
- **JSON** serialized Java classes can be represented using JSON notation, e.g. {"package.MyClass":{"i":5,"x":null,"b":true}}. Please note that the specified class must be available to the CacheManager's class loader.

17.2.5. Time Values

A time value is an integer number followed by time unit suffix: days (d), hours (h), minutes (m), seconds (s), milliseconds (ms).

17.3. Hawt.io

[Hawt.io](#), a slick, fast, HTML5-based open source management console, also has support for {brandname}. Refer to [Hawt.io's documentation](#) for information regarding this plugin.

17.4. Writing plugins for other management tools

Any management tool that supports JMX already has basic support for {brandname}. However, custom plugins could be written to adapt the JMX information for easier consumption.

Chapter 18. Custom Interceptors

It is possible to add custom interceptors to {brandname}, both declaratively and programmatically. Custom interceptors are a way of extending {brandname} by being able to influence or respond to any modifications to cache. Example of such modifications are: elements are added/removed/updated or transactions are committed. For a detailed list refer to [CommandInterceptor](#) API.

18.1. Adding custom interceptors declaratively

Custom interceptors can be added on a per named cache basis. This is because each named cache have its own interceptor stack. Following xml snippet depicts the ways in which a custom interceptor can be added.

```
<local-cache name="cacheWithCustomInterceptors">
  <!--
    Define custom interceptors. All custom interceptors need to extend
    org.jboss.cache.interceptors.base.CommandInterceptor
  -->
  <custom-interceptors>
    <interceptor position="FIRST" class="com.mycompany.CustomInterceptor1">
      <property name="attributeOne">value1</property>
      <property name="attributeTwo">value2</property>
    </interceptor>
    <interceptor position="LAST" class="com.mycompany.CustomInterceptor2"/>
    <interceptor index="3" class="com.mycompany.CustomInterceptor1"/>
    <interceptor before="org.infinispanpan.interceptors.CallInterceptor" class=
"com.mycompany.CustomInterceptor2"/>
    <interceptor after="org.infinispanpan.interceptors.CallInterceptor" class=
"com.mycompany.CustomInterceptor1"/>
  </custom-interceptors>
</local-cache>
```

18.2. Adding custom interceptors programmatically

In order to do that one needs to obtain a reference to the [AdvancedCache](#) . This can be done as follows:

```
CacheManager cm = getCacheManager();//magic
Cache aCache = cm.getCache("aName");
AdvancedCache advCache = aCache.getAdvancedCache();
```

Then one of the *addInterceptor()* methods should be used to add the actual interceptor. For further documentation refer to [AdvancedCache](#) javadoc.

18.3. Custom interceptor design

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

- + * Custom interceptors must extend [BaseCustomInterceptor](#)
- * Custom interceptors must declare a public, empty constructor to enable construction.
- * Custom interceptors will have setters for any property defined through property tags used in the XML configuration.

Chapter 19. Running on Cloud Services

In order to turn on Cloud support for {brandname} library mode, one needs to add a new dependency to the classpath:

Cloud support in library mode

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cloud</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

The above dependency adds `infinispan-core` to the classpath as well as some default configurations.

19.1. Generic Discovery protocols

The main difference between running {brandname} in a private environment and a cloud provider is that in the latter node discovery becomes a bit trickier because things like multicast don't work. To circumvent this you can use alternate JGroups PING protocols. Before delving into the cloud-specific, let's look at some generic discovery protocols.

19.1.1. TCPing

The TCPing approach contains a static list of the IP address of each member of the cluster in the JGroups configuration file. While this works it doesn't really help when cluster nodes are dynamically added to the cluster.

Sample TCPing configuration

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

See [JGroups TCPPING](#) for more information about TCPing.

19.1.2. GossipRouter

Another approach is to have a central server (Gossip, which each node will be configured to contact. This central server will tell each node in the cluster about each other node.

The address (ip:port) that the Gossip router is listening on can be injected into the JGroups configuration used by {brandname}. To do this pass the gossip routers address as a system property to the JVM e.g. `-DGossipRouterAddress="10.10.2.4[12001]"` and reference this property in the JGroups configuration that {brandname} is using e.g.

Sample TCPGOSSIP configuration

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

More on Gossip Router @ <http://www.jboss.org/community/wiki/JGroupsGossipRouter>

19.2. Amazon Web Services

When running on Amazon Web Service (AWS) platform and similar cloud based environment you can use the S3_PING protocol for discovery.

19.2.1. S3_PING

You can configure your JGroups instances to use a shared storage to exchange the details of the cluster nodes. S3_PING allows Amazon S3 to be used as the shared storage. Be sure that you have signed up for Amazon S3 as well as EC2 to use this method.

Sample S3PING configuration

```
<config>
  <TCP bind_port="7800" />
  <S3_PING
    secret_access_key="replace this with you secret access key"
    access_key="replace this with your access key"
    location="replace this with your S3 bucket location" />
</config>
```

19.2.2. JDBC_PING

A similar approach to S3_PING, but using a JDBC connection to a shared database. On EC2 that is quite easy using Amazon RDS. See the [JDBC_PING Wiki page](#) for details.

19.3. Microsoft Azure

{brandname} can be used on the Azure platform. Aside from using TCP_PING or GossipRouter, there is an Azure-specific discovery protocol:

19.3.1. AZURE_PING

AZURE_PING uses a shared Azure Blob Storage to store discovery information. Configuration is as follows:

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

19.4. Google Compute Engine

{brandname} can be used on the Google Compute Engine (GCE) platform. Aside from using TCP_PING or GossipRouter, there is a GCE-specific discovery protocol:

19.4.1. GOOGLE_PING

GOOGLE_PING uses Google Cloud Storage (GCS) to store information about the cluster members.

```
<protocol type="GOOGLE_PING">
  <property name="location">The name of the bucket</property>
  <property name="access_key">The access key</property>
  <property name="secret_access_key">The secret access key</property>
</protocol>
```

19.5. Kubernetes and OpenShift

Since OpenShift uses Kubernetes underneath both of them can use the same discovery protocol - [Kube_PING](#). The configuration is very straightforward:

Sample KUBE_PING configuration

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

The most important thing is to bind JGroups to `eth0` interface, which is [used by Docker containers](#)

for network communication.

KUBE_PING protocol is configured by environmental variables (which should be available inside a container). The most important thing is to set `KUBERNETES_NAMESPACE` to proper namespace. It might be either hardcoded or populated via [Kubernetes' Downward API](#).

Since KUBE_PING uses Kubernetes API for obtaining available Pods, OpenShift requires adding additional privileges. Assuming that `oc project -q` returns current namespace and `default` is the service account name, one needs to run:

Adding additional OpenShift privileges

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

After performing all above steps, the clustering should be enabled and all Pods should automatically form a cluster within a single namespace.

19.5.1. Using Kubernetes and OpenShift Rolling Updates

Since Pods in Kubernetes and OpenShift are immutable, the only way to alter the configuration is to roll out a new deployment. There are several different strategies to do that but we suggest using [Rolling Updates](#).

An example Deployment Configuration (Kubernetes uses very similar concept called `Deployment`) looks like the following:

DeploymentConfiguration for Rolling Updates

```
- apiVersion: v1
  kind: DeploymentConfig
  metadata:
    name: infinispn-cluster
  spec:
    replicas: 3
    strategy:
      type: Rolling
      rollingParams:
        updatePeriodSeconds: 10
        intervalSeconds: 20
        timeoutSeconds: 600
        maxUnavailable: 1
        maxSurge: 1
    template:
      spec:
        containers:
          - args:
              - -Djboss.default.jgroups.stack=kubernetes
            image: jboss/infinispn-server:latest
            name: infinispn-server
```

```

ports:
- containerPort: 8181
  protocol: TCP
- containerPort: 9990
  protocol: TCP
- containerPort: 11211
  protocol: TCP
- containerPort: 11222
  protocol: TCP
- containerPort: 57600
  protocol: TCP
- containerPort: 7600
  protocol: TCP
- containerPort: 8080
  protocol: TCP
env:
- name: KUBERNETES_NAMESPACE
  valueFrom: {fieldRef: {apiVersion: v1, fieldPath: metadata.namespace}}
terminationMessagePath: /dev/termination-log
terminationGracePeriodSeconds: 90
livenessProbe:
  exec:
    command:
    - /usr/local/bin/is_running.sh
  initialDelaySeconds: 10
  timeoutSeconds: 80
  periodSeconds: 60
  successThreshold: 1
  failureThreshold: 5
readinessProbe:
  exec:
    command:
    - /usr/local/bin/is_healthy.sh
  initialDelaySeconds: 10
  timeoutSeconds: 40
  periodSeconds: 30
  successThreshold: 2
  failureThreshold: 5

```

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

Other hints for tuning configuration parameters are:

- OpenShift should replace running nodes one by one. This can be achieved by adjusting `rollingParams` (`maxUnavailable: 1` and `maxSurge: 1`).
- Depending on the cluster size, one needs to adjust `updatePeriodSeconds` and `intervalSeconds`. The bigger cluster size is, the bigger those values should be used.
- When using Initial State Transfer, the `initialDelaySeconds` value for both probes should be set to higher value.

- During Initial State Transfer nodes might not respond to probes. The best results are achieved with higher values of `failureThreshold` and `successThreshold` values.

19.5.2. Rolling upgrades with Kubernetes and OpenShift

Even though Rolling Upgrades and Rolling Update may sound similarly, they mean different things. The [Rolling Update](#) is a process of replacing old Pods with new ones. In other words it is a process of rolling out new version of an application. A typical example is a configuration change. Since Pods are immutable, Kubernetes/OpenShift needs to replace them one by one in order to use the updated configuration bits. On the other hand the [Rolling Upgrade](#) is a process of migrating data from one {brandname} cluster to another one. A typical example is migrating from one version to another.

For both Kubernetes and OpenShift, the Rolling Upgrade procedure is almost the same. It is based on a standard [Rolling Upgrade procedure](#) with small changes.

Key differences when upgrading using OpenShift/Kubernetes are:

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

Key differences when upgrading using the library mode:

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

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

A detailed instruction might be found in [ISPAN-6673](#) ticket.

Chapter 20. Client/Server

{brandname} offers two alternative access methods: embedded mode and client-server mode.

- In Embedded mode the {brandname} libraries co-exist with the user application in the same JVM as shown in the following diagram

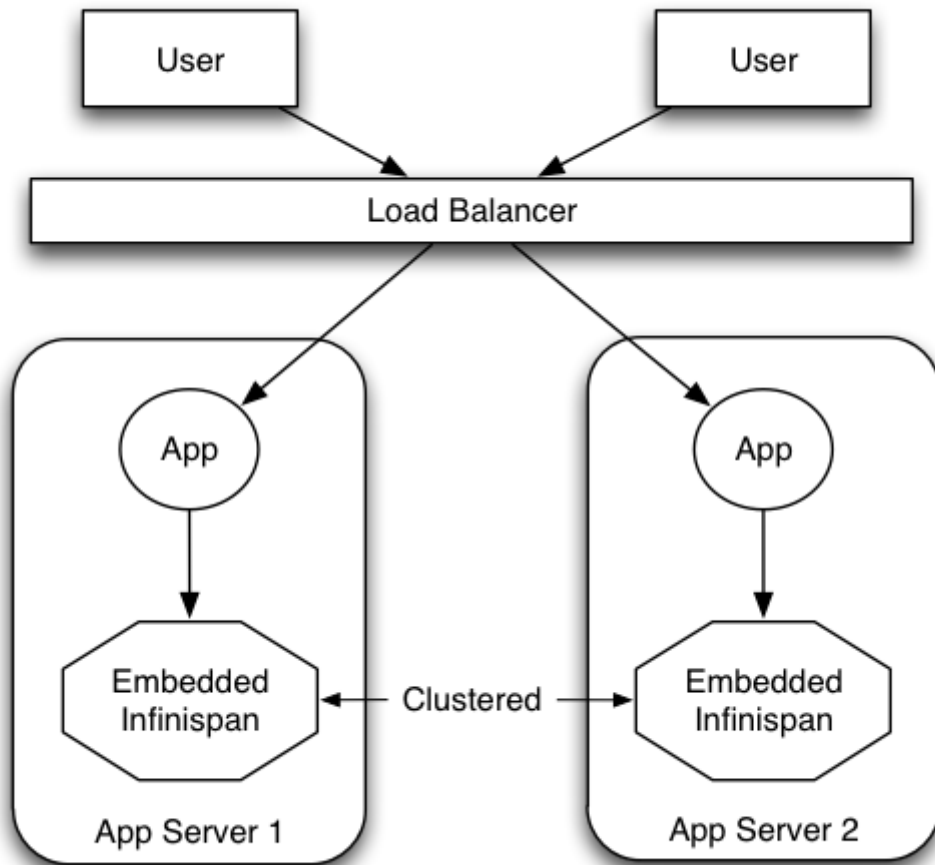


Figure 11. Peer-to-peer access

- Client-server mode is when applications access the data stored in a remote {brandname} server using some kind of network protocol

20.1. Why Client/Server?

There are situations when accessing {brandname} in a client-server mode might make more sense than embedding it within your application, for example, when trying to access {brandname} from a non-JVM environment. Since {brandname} is written in Java, if someone had a C\\ application that wanted to access it, it couldn't just do it in a p2p way. On the other hand, client-server would be perfectly suited here assuming that a language neutral protocol was used and the corresponding client and server implementations were available.

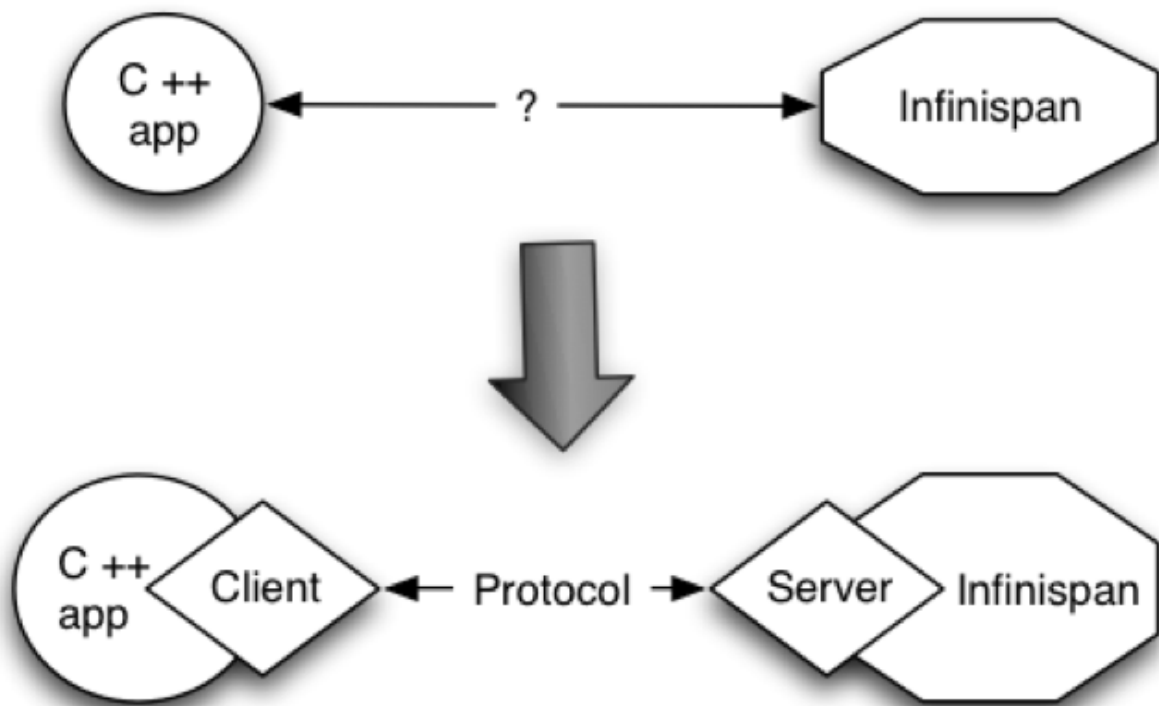


Figure 12. Non-JVM access

In other situations, {brandname} users want to have an elastic application tier where you start/stop business processing servers very regularly. Now, if users deployed {brandname} configured with distribution or state transfer, startup time could be greatly influenced by the shuffling around of data that happens in these situations. So in the following diagram, assuming {brandname} was deployed in p2p mode, the app in the second server could not access {brandname} until state transfer had completed.

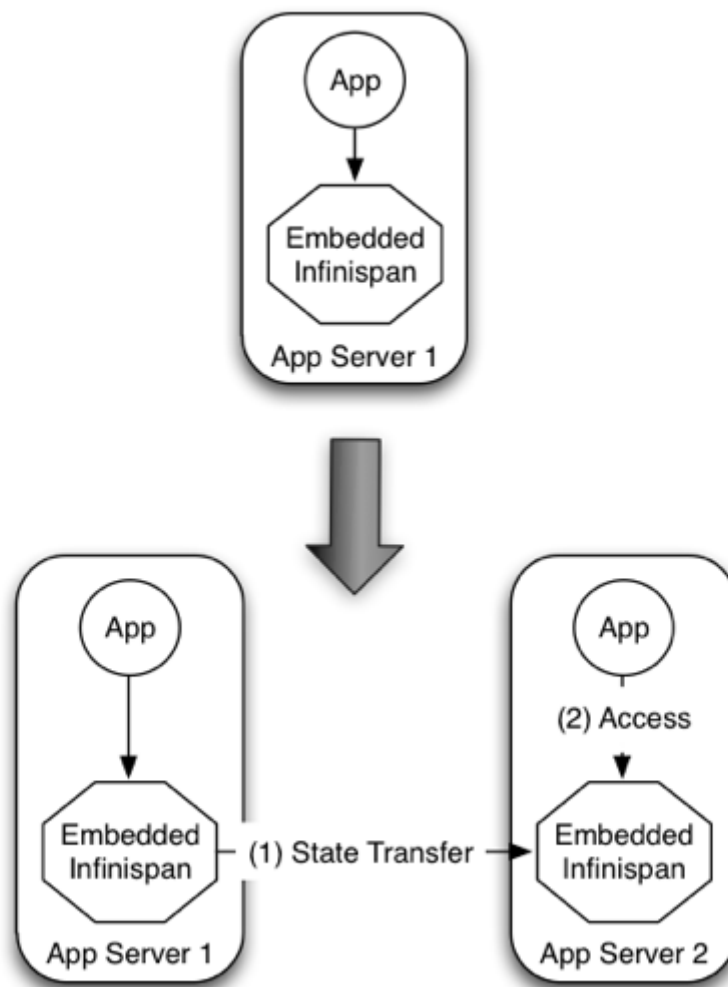


Figure 13. Elasticity issue with P2P

This effectively means that bringing up new application-tier servers is impacted by things like state transfer because applications cannot access {brandname} until these processes have finished and if the state being shifted around is large, this could take some time. This is undesirable in an elastic environment where you want quick application-tier server turnaround and predictable startup times. Problems like this can be solved by accessing {brandname} in a client-server mode because starting a new application-tier server is just a matter of starting a lightweight client that can connect to the backing data grid server. No need for rehashing or state transfer to occur and as a result server startup times can be more predictable which is very important for modern cloud-based deployments where elasticity in your application tier is important.

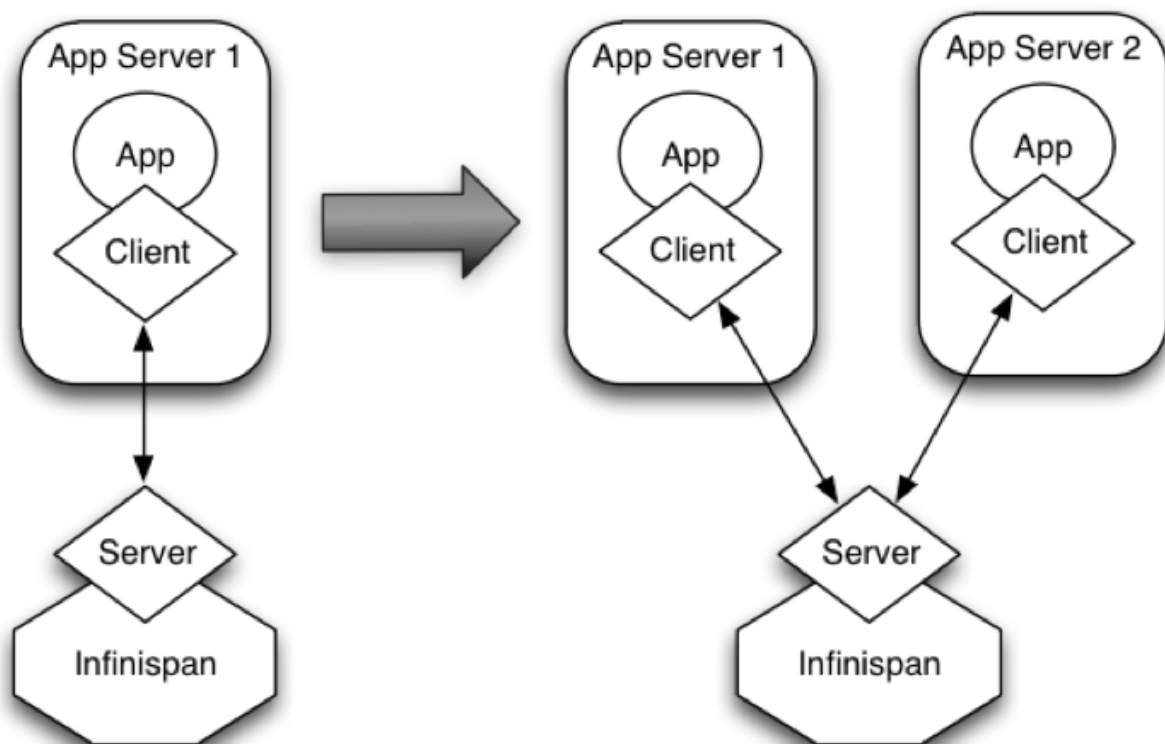


Figure 14. Achieving elasticity

Other times, it's common to find multiple applications needing access to data storage. In this cases, you could in theory deploy an {brandname} instance per each of those applications but this could be wasteful and difficult to maintain. Think about databases here, you don't deploy a database alongside each of your applications, do you? So, alternatively you could deploy {brandname} in client-server mode keeping a pool of {brandname} data grid nodes acting as a shared storage tier for your applications.

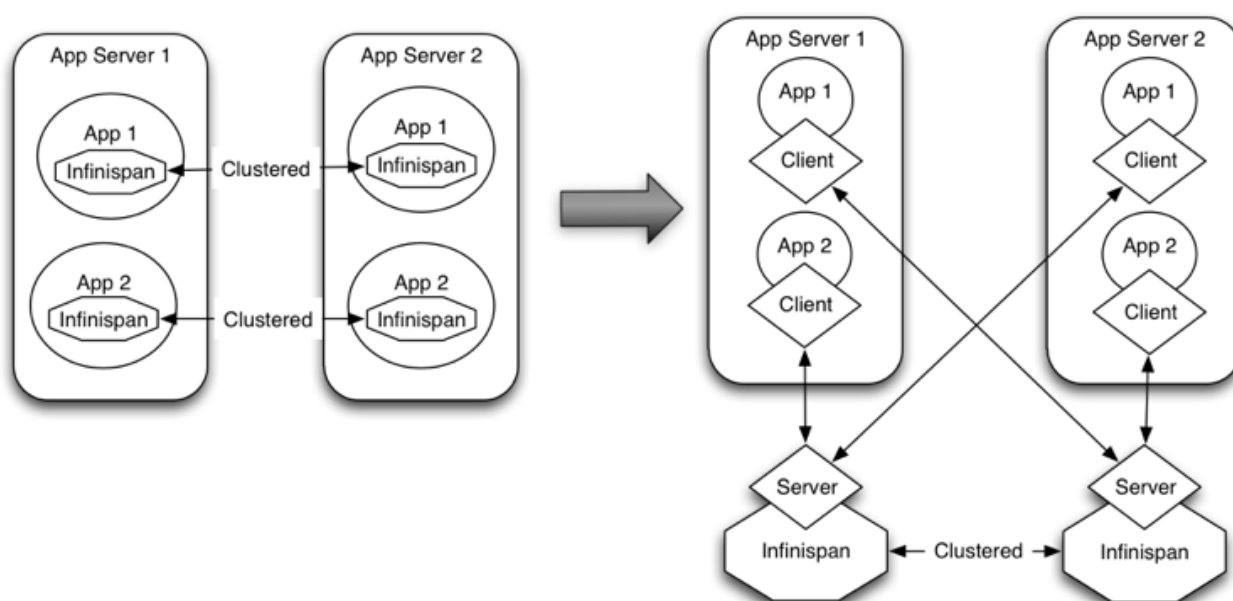


Figure 15. Shared data storage

Deploying {brandname} in this way also allows you to manage each tier independently, for example, you can upgrade you application or app server without bringing down your {brandname} data grid nodes.

20.2. Why use embedded mode?

Before talking about individual {brandname} server modules, it's worth mentioning that in spite of all the benefits, client-server {brandname} still has disadvantages over p2p. Firstly, p2p deployments are simpler than client-server ones because in p2p, all peers are equals to each other and hence this simplifies deployment. So, if this is the first time you're using {brandname}, p2p is likely to be easier for you to get going compared to client-server.

Client-server {brandname} requests are likely to take longer compared to p2p requests, due to the serialization and network cost in remote calls. So, this is an important factor to take in account when designing your application. For example, with replicated {brandname} caches, it might be more performant to have lightweight HTTP clients connecting to a server side application that accesses {brandname} in p2p mode, rather than having more heavyweight client side apps talking to {brandname} in client-server mode, particularly if data size handled is rather large. With distributed caches, the difference might not be so big because even in p2p deployments, you're not guaranteed to have all data available locally.

Environments where application tier elasticity is not so important, or where server side applications access state-transfer-disabled, replicated {brandname} cache instances are amongst scenarios where {brandname} p2p deployments can be more suited than client-server ones.

20.3. Server Modules

So, now that it's clear when it makes sense to deploy {brandname} in client-server mode, what are available solutions? All {brandname} server modules are based on the same pattern where the server backend creates an embedded {brandname} instance and if you start multiple backends, they can form a cluster and share/distribute state if configured to do so. The server types below primarily differ in the type of listener endpoint used to handle incoming connections.

Here's a brief summary of the available server endpoints.

- **Hot Rod Server Module** - This module is an implementation of the [Hot Rod binary protocol](#) backed by {brandname} which allows clients to do dynamic load balancing and failover and smart routing.
 - A [variety of clients](#) exist for this protocol.
 - If you're clients are running Java, this should be your defacto server module choice because it allows for dynamic load balancing and failover. This means that Hot Rod clients can dynamically detect changes in the topology of Hot Rod servers as long as these are clustered, so when new nodes join or leave, clients update their Hot Rod server topology view. On top of that, when Hot Rod servers are configured with distribution, clients can detect where a particular key resides and so they can route requests smartly.
 - Load balancing and failover is dynamically provided by Hot Rod client implementations using information provided by the server.

- **REST Server Module** - The REST server, which is distributed as a WAR file, can be deployed in any servlet container to allow {brandname} to be accessed via a RESTful HTTP interface.
 - To connect to it, you can use any HTTP client out there and there're tons of different client implementations available out there for pretty much any language or system.
 - This module is particularly recommended for those environments where HTTP port is the only access method allowed between clients and servers.
 - Clients wanting to load balance or failover between different {brandname} REST servers can do so using any standard HTTP load balancer such as [mod_cluster](#) . It's worth noting though these load balancers maintain a static view of the servers in the backend and if a new one was to be added, it would require manual update of the load balancer.
- **Memcached Server Module** - This module is an implementation of the [Memcached text protocol](#) backed by {brandname}.
 - To connect to it, you can use any of the [existing Memcached clients](#) which are pretty diverse.
 - As opposed to Memcached servers, {brandname} based Memcached servers can actually be clustered and hence they can replicate or distribute data using consistent hash algorithms around the cluster. So, this module is particularly of interest to those users that want to provide failover capabilities to the data stored in Memcached servers.
 - In terms of load balancing and failover, there're a few clients that can load balance or failover given a static list of server addresses (perl's Cache::Memcached for example) but any server addition or removal would require manual intervention.

20.4. Which protocol should I use?

Choosing the right protocol depends on a number of factors.

	Hot Rod	HTTP / REST	Memcached
Topology-aware	Y	N	N
Hash-aware	Y	N	N
Encryption	Y	Y	N
Authentication	Y	Y	N
Conditional ops	Y	Y	Y
Bulk ops	Y	N	N
Transactions	N	N	N
Listeners	Y	N	N
Query	Y	Y	N
Execution	Y	N	N
Cross-site failover	Y	N	N

20.5. Using Hot Rod Server

The {brandname} Server distribution contains a server module that implements {brandname}'s

custom binary protocol called Hot Rod. The protocol was designed to enable faster client/server interactions compared to other existing text based protocols and to allow clients to make more intelligent decisions with regards to load balancing, failover and even data location operations. Please refer to {brandname} Server's [documentation](#) for instructions on how to configure and run a HotRod server.

To connect to {brandname} over this highly efficient Hot Rod protocol you can either use one of the clients described in this chapter, or use higher level tools such as [Hibernate OGM](#).

20.6. Java Hot Rod client

Hot Rod is a binary, language neutral protocol. This article explains how a Java client can interact with a server via the Hot Rod protocol. A reference implementation of the protocol written in Java can be found in all {brandname} distributions, and this article focuses on the capabilities of this java client.



Looking for more clients? Visit [this website](#) for clients written in a variety of different languages.

20.6.1. Configuration

The Java Hot Rod client can be configured both programmatically and externally, through a configuration file.

The code snippet below illustrates the creation of a client instance using the available Java fluent API:

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
    = new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.tcpNoDelay(true)
  .connectionPool()
    .numTestsPerEvictionRun(3)
    .testOnBorrow(false)
    .testOnReturn(false)
    .testWhileIdle(true)
  .addServer()
    .host("localhost")
    .port(11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

For a complete reference to the available configuration option please refer to the [ConfigurationBuilder](#)'s javadoc.

It is also possible to configure the Java Hot Rod client using a properties file, e.g.:

```

infinispan.client.hotrod.transport_factory =
org.infinispan.client.hotrod.impl.transport.tcp.TcpTransportFactory
infinispan.client.hotrod.server_list = 127.0.0.1:11222
infinispan.client.hotrod.marshaller =
org.infinispan.commons.marshall.jboss.GenericJBossMarshaller
infinispan.client.hotrod.async_executor_factory =
org.infinispan.client.hotrod.impl.async.DefaultAsyncExecutorFactory
infinispan.client.hotrod.default_executor_factory.pool_size = 1
infinispan.client.hotrod.default_executor_factory.queue_size = 10000
infinispan.client.hotrod.tcp_no_delay = true
infinispan.client.hotrod.request_balancing_strategy =
org.infinispan.client.hotrod.impl.transport.tcp.RoundRobinBalancingStrategy
infinispan.client.hotrod.key_size_estimate = 64
infinispan.client.hotrod.value_size_estimate = 512
infinispan.client.hotrod.force_return_values = false
infinispan.client.hotrod.client_intelligence = HASH_DISTRIBUTION_AWARE
infinispan.client.hotrod.batch_size = 10000

## below is connection pooling config
maxActive=-1
maxTotal = -1
maxIdle = -1
whenExhaustedAction = 1
timeBetweenEvictionRunsMillis=120000
minEvictableIdleTimeMillis=300000
testWhileIdle = true
minIdle = 1

```

The properties file is then passed to one of constructors of [RemoteCacheManager](#). You can use property substitution to replace values at runtime with [Java system properties](#):

```

infinispan.client.hotrod.server_list = ${server_list}

```

In the above example the value of the *infinispan.client.hotrod.server_list* property will be expanded to the value of the *server_list* Java system property.

which means that the value should be taken from a system property named *jboss.bind.address.management* and if it is not defined use 127.0.0.1.

For a complete reference of the available configuration options for the properties file please refer to [remote client configuration](#) javadoc.

20.6.2. Authentication

If the server has set up authentication, you need to configure your client accordingly. Depending on the mechs enabled on the server, the client must provide the required information.



This section is about client configuration. If you want to set up the server to require authentication, read the [Hot Rod server authentication](#) section.

DIGEST-MD5

DIGEST-MD5 is the recommended approach for simple username/password authentication scenarios. If you are using the default realm on the server (*"ApplicationRealm"*), all you need to do is provide your credentials as follows:

Hot Rod client configuration with DIGEST-MD5 authentication

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .ssl()
            .username("myuser")
            .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

PLAIN

The PLAIN mechanism is not really recommended unless the connection is also encrypted, as anyone can sniff the clear-text password being sent along the wire.

Hot Rod client configuration with DIGEST-MD5 authentication

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .authentication()
            .saslMechanism("PLAIN")
            .username("myuser")
            .password("qwer1234!");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

EXTERNAL

The EXTERNAL mechanism is special in that it doesn't explicitly provide credentials but uses the client certificate as identity. In order for this to work, in addition to the *TrustStore* (to validate the server certificate) you need to provide a *KeyStore* (to supply the client certificate).

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .ssl()
            // TrustStore is a KeyStore which contains part of the server certificate
            chain (e.g. the CA Root public cert)
            .trustStoreFileName("/path/to/truststore")
            .trustStorePassword("truststorepassword".toCharArray())
            // KeyStore containing this client's own certificate
            .keyStoreFileName("/path/to/keystore")
            .keyStorePassword("keystorepassword".toCharArray())
        .authentication()
            .saslMechanism("EXTERNAL");
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

For more details, read the [Encryption](#) section below.

GSSAPI (Kerberos)

GSSAPI/Kerberos requires a much more complex setup, but it is used heavily in enterprises with centralized authentication servers. To successfully authenticate with Kerberos, you need to create a *LoginContext*. This will obtain a Ticket Granting Ticket (TGT) which will be used as a token to authenticate with the service.

You will need to define a login module in a login configuration file:

gss.conf

```
GssExample {
    com.sun.security.auth.module.Krb5LoginModule required client=TRUE;
};
```

If you are using the IBM JDK, the above becomes:

gss-ibm.conf

```
GssExample {
    com.ibm.security.auth.module.Krb5LoginModule required client=TRUE;
};
```

You will also need to set the following system properties:

`java.security.auth.login.config=gss.conf`


```
java.security.krb5.conf=/etc/krb5.conf
```

The `krb5.conf` file is dependent on your environment and needs to point to your KDC. Ensure that you can authenticate via Kerberos using *kinit*.

Next up, configure your client as follows:

Hot Rod client GSSAPI configuration

```
LoginContext lc = new LoginContext("GssExample", new BasicCallbackHandler("krb_user",
"krb_password".toCharArray()));
lc.login();
Subject clientSubject = lc.getSubject();

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .authentication()
            .enable()
            .serverName("infinispan-server")
            .saslmMechanism("GSSAPI")
            .clientSubject(clientSubject)
            .callbackHandler(new BasicCallbackHandler());
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

For brevity we used the same callback handler both for obtaining the client subject and for handling authentication in the SASL GSSAPI mech, however different callbacks will actually be invoked: `NameCallback` and `PasswordCallback` are needed to construct the client subject, while the `AuthorizeCallback` will be called during the SASL authentication.

Custom CallbackHandlers

In all of the above examples, the Hot Rod client is setting up a default *CallbackHandler* for you that supplies the provided credentials to the SASL mechanism. For advanced scenarios it may be necessary to provide your own custom *CallbackHandler*:

Hot Rod client configuration with authentication via callback

```
public class MyCallbackHandler implements CallbackHandler {
    final private String username;
    final private char[] password;
    final private String realm;

    public MyCallbackHandler(String username, String realm, char[] password) {
        this.username = username;
        this.password = password;
        this.realm = realm;
    }
}
```

```

    }

    @Override
    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            if (callback instanceof NameCallback) {
                NameCallback nameCallback = (NameCallback) callback;
                nameCallback.setName(username);
            } else if (callback instanceof PasswordCallback) {
                PasswordCallback passwordCallback = (PasswordCallback) callback;
                passwordCallback.setPassword(password);
            } else if (callback instanceof AuthorizeCallback) {
                AuthorizeCallback authorizeCallback = (AuthorizeCallback) callback;
                authorizeCallback.setAuthorized(authorizeCallback.getAuthenticationID()
                    .equals(
                        authorizeCallback.getAuthorizationID()));
            } else if (callback instanceof RealmCallback) {
                RealmCallback realmCallback = (RealmCallback) callback;
                realmCallback.setText(realm);
            } else {
                throw new UnsupportedCallbackException(callback);
            }
        }
    }
}

ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
        .security()
            .authentication()
                .enable()
                .serverName("myhotrodserver")
                .saslMechanism("DIGEST-MD5")
                .callbackHandler(new MyCallbackHandler("myuser", "ApplicationRealm",
                    "qwer1234!".toCharArray()));
remoteCacheManager = new RemoteCacheManager(clientBuilder.build());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");

```

The actual type of callbacks that your CallbackHandler will need to be able to handle are mech-specific, so the above is just a simple example.

20.6.3. Encryption



This section is about client configuration. If you want to set up the server to require encryption, read the [Hot Rod server encryption](#) section.

Encryption uses TLS/SSL, so it requires setting up an appropriate server certificate chain. Generally, a certificate chain looks like the following:

The screenshot shows a 'Certificate Hierarchy' window. At the top, there is a tree view with two entries: 'CA' and 'HotRodServer'. 'HotRodServer' is selected and highlighted in blue. Below the tree, the details of the selected certificate are displayed in a form-like layout:

- Version: 3
- Subject: CN=HotRodServer.OU=Infinispan.O=JBoss.L=Red Hat
- Issuer: CN=CA.OU=Infinispan.O=JBoss.L=Red Hat
- Serial Number: 0x41603743
- Valid From: 2/9/2018 4:22:49 PM CET
- Valid Until: 2/9/2019 4:22:49 PM CET
- Public Key: RSA 2048 bits
- Signature Algorithm: SHA256WITHRSA
- Fingerprint: SHA-1, E5:7F:0D:42:D8:46:B2:BD:79:85:9B:1E:BC:03:BB:26:C

Figure 16. Certificate chain

In the above example there is one certificate authority "CA" which has issued a certificate for "HotRodServer". In order for a client to trust the server, it needs at least a portion of the above chain (usually, just the public certificate for "CA"). This certificate needs to be placed in a keystore and used as a *TrustStore* on the client and used as shown below:

Hot Rod client configuration with TLS (server cert)

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .ssl()
            // TrustStore is a KeyStore which contains part of the server certificate
            // chain (e.g. the CA Root public cert)
            .trustStoreFileName("/path/to/truststore")
            .trustStorePassword("truststorepassword".toCharArray());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

SNI

The server may have been configured with TLS/SNI support ([Server Name Indication](#)). This means

that the server is presenting multiple identities (probably bound to separate cache containers). The client can specify which identity to connect to by specifying its name:

Hot Rod client configuration with SNI (server cert)

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .ssl()
            .sniHostName("myservername")
            // TrustStore is a KeyStore which contains part of the server certificate
            // chain (e.g. the CA Root public cert)
            .trustStoreFileName("/path/to/truststore")
            .trustStorePassword("truststorepassword".toCharArray());
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

Client certificates

With the above configurations the client trusts the server. For increased security, a server administrator may have set up the server to require the client to offer a valid certificate for mutual trust. This kind of configuration requires the client to present its own certificate, usually issued by the same certificate authority as the server. This certificate must be stored in a keystore and used as follows:

Hot Rod client configuration with TLS (server and client cert)

```
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder
    .addServer()
        .host("127.0.0.1")
        .port(11222)
    .security()
        .ssl()
            // TrustStore is a KeyStore which contains part of the server certificate
            // chain (e.g. the CA Root public cert)
            .trustStoreFileName("/path/to/truststore")
            .trustStorePassword("truststorepassword".toCharArray())
            // KeyStore containing this client's own certificate
            .keyStoreFileName("/path/to/keystore")
            .keyStorePassword("keystorepassword".toCharArray())
RemoteCache<String, String> cache = remoteCacheManager.getCache("secured");
```

Please read the [KeyTool](#) documentation for more details on KeyStores. Additionally, the [KeyStore Explorer](#) is a great GUI tool for easily managing KeyStores.

20.6.4. Basic API

Below is a sample code snippet on how the client API can be used to store or retrieve information from a Hot Rod server using the Java Hot Rod client. It assumes that a Hot Rod server has been started bound to the default location (localhost:11222)

```
//API entry point, by default it connects to localhost:11222
CacheContainer cacheContainer = new RemoteCacheManager();

//obtain a handle to the remote default cache
Cache<String, String> cache = cacheContainer.getCache();

//now add something to the cache and make sure it is there
cache.put("car", "ferrari");
assert cache.get("car").equals("ferrari");

//remove the data
cache.remove("car");
assert !cache.containsKey("car") : "Value must have been removed!";
```

The client API maps the local API: [RemoteCacheManager](#) corresponds to [DefaultCacheManager](#) (both implement [CacheContainer](#)). This common API facilitates an easy migration from local calls to remote calls through Hot Rod: all one needs to do is switch between [DefaultCacheManager](#) and [RemoteCacheManager](#) - which is further simplified by the common [CacheContainer](#) interface that both inherit.

20.6.5. RemoteCache(.keySet|.entrySet|.values)

The collection methods [keySet](#), [entrySet](#) and [values](#) are backed by the remote cache. That is that every method is called back into the [RemoteCache](#). This is useful as it allows for the various keys, entries or values to be retrieved lazily, and not requiring them all be stored in the client memory at once if the user does not want. These collections adhere to the [Map](#) specification being that [add](#) and [addAll](#) are not supported but all other methods are supported.

One thing to note is the [Iterator.remove](#) and [Set.remove](#) or [Collection.remove](#) methods require more than 1 round trip to the server to operate. You can check out the [RemoteCache](#) Javadoc to see more details about these and the other methods.

Iterator Usage

The iterator method of these collections uses [retrieveEntries](#) internally, which is described below. If you notice [retrieveEntries](#) takes an argument for the batch size. There is no way to provide this to the iterator. As such the batch size can be configured via system property [infinispan.client.hotrod.batch_size](#) or through the [ConfigurationBuilder](#) when configuring the [RemoteCacheManager](#).

Also the [retrieveEntries](#) iterator returned is [Closeable](#) as such the iterators from [keySet](#), [entrySet](#) and [values](#) return an [AutoCloseable](#) variant. Therefore you should always close these `Iterator`s when you are done with them.

```
try (CloseableIterator<Entry<K, V>> iterator = remoteCache.entrySet().iterator) {  
    ...  
}
```

What if I want a deep copy and not a backing collection?

Previous version of `RemoteCache` allowed for the retrieval of a deep copy of the `keySet`. This is still possible with the new backing map, you just have to copy the contents yourself. Also you can do this with `entrySet` and `values`, which we didn't support before.

```
Set<K> keysCopy = remoteCache.keySet().stream().collect(Collectors.toSet());
```

Please use extreme caution with this as a large number of keys can and will cause `OutOfMemoryError` in the client.

```
Set keys = remoteCache.keySet();
```

20.6.6. Remote Iterator

Alternatively, if memory is a concern (different batch size) or you wish to do server side filtering or conversion), use the remote iterator api to retrieve entries from the server. With this method you can limit the entries that are retrieved or even returned a converted value if you don't need all properties of your entry.

```

// Retrieve all entries in batches of 1000
int batchSize = 1000;
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries(
    null, batchSize)) {
    while(iterator.hasNext()) {
        // Do something
    }
}

// Filter by segment
Set<Integer> segments = ...
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries(
    null, segments, batchSize)) {
    while(iterator.hasNext()) {
        // Do something
    }
}

// Filter by custom filter
try (CloseableIterator<Entry<Object, Object>> iterator = remoteCache.retrieveEntries(
    "myFilterConverterFactory", segments, batchSize)) {
    while(iterator.hasNext()) {
        // Do something
    }
}

```

In order to use custom filters, it's necessary to deploy them first in the server. Follow the steps:

- Create a factory for the filter extending [KeyValueFilterConverterFactory](#), annotated with `@NamedFactory` containing the name of the factory, example:

```

import java.io.Serializable;

import org.infinispan.filter.AbstractKeyValueFilterConverter;
import org.infinispan.filter.KeyValueFilterConverter;
import org.infinispan.filter.KeyValueFilterConverterFactory;
import org.infinispan.filter.NamedFactory;
import org.infinispan.metadata.Metadata;

@NamedFactory(name = "myFilterConverterFactory")
public class MyKeyValueFilterConverterFactory implements
KeyValueFilterConverterFactory {

    @Override
    public KeyValueFilterConverter<String, SampleEntity1, SampleEntity2>
getFilterConverter() {
        return new MyKeyValueFilterConverter();
    }
    // Filter implementation. Should be serializable or externalizable for DIST caches
    static class MyKeyValueFilterConverter extends AbstractKeyValueFilterConverter
<String, SampleEntity1, SampleEntity2> implements Serializable {
        @Override
        public SampleEntity2 filterAndConvert(String key, SampleEntity1 entity, Metadata
metadata) {
            // returning null will case the entry to be filtered out
            // return SampleEntity2 will convert from the cache type SampleEntity1
        }

        @Override
        public MediaType format() {
            // returns the MediaType that data should be presented to this converter.
            // When ommitted, the server will use "application/x-java-object".
            // Returning null will cause the filter/converter to be done in the storage
format.
        }
    }
}

```

- Create a jar with a **META-INF/services/org.infinispan.filter.KeyValueFilterConverterFactory** file and within it, write the fully qualified class name of the filter factory class implementation.
- Optional: If the filter uses custom key/value classes, these must be included in the JAR so that the filter can correctly unmarshall key and/or value instances.
- Deploy the JAR file in the {brandname} Server.

20.6.7. Versioned API

A RemoteCacheManager provides instances of [RemoteCache](#) interface that represents a handle to the named or default cache on the remote cluster. API wise, it extends the [Cache](#) interface to which it also adds some new methods, including the so called versioned API. Please find below some

examples of this API link:[#server_hotrod_failover](#)[but to understand the motivation behind it, make sure you read this section.

The code snippet bellow depicts the usage of these versioned methods:

```
// To use the versioned API, remote classes are specifically needed
RemoteCacheManager remoteCacheManager = new RemoteCacheManager();
RemoteCache<String, String> cache = remoteCacheManager.getCache();

remoteCache.put("car", "ferrari");
RemoteCache.VersionedValue valueBinary = remoteCache.getVersioned("car");

// removal only takes place only if the version has not been changed
// in between. (a new version is associated with 'car' key on each change)
assert remoteCache.remove("car", valueBinary.getVersion());
assert !cache.containsKey("car");
```

In a similar way, for replace:

```
remoteCache.put("car", "ferrari");
RemoteCache.VersionedValue valueBinary = remoteCache.getVersioned("car");
assert remoteCache.replace("car", "lamborghini", valueBinary.getVersion());
```

For more details on versioned operations refer to [RemoteCache](#) 's javadoc.

20.6.8. Async API

This is "borrowed" from the {brandname} core and it is largely discussed [here](#).

20.6.9. Streaming API

When sending / receiving large objects, it might make sense to stream them between the client and the server. The Streaming API implements methods similar to the [Hot Rod Basic API](#) and [Hot Rod Versioned API](#) described above but, instead of taking the value as a parameter, they return instances of `InputStream` and `OutputStream`. The following example shows how one would write a potentially large object:

```
RemoteStreamingCache<String> streamingCache = remoteCache.streaming();
OutputStream os = streamingCache.put("a_large_object");
os.write(...);
os.close();
```

Reading such an object through streaming:

```
RemoteStreamingCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
for(int b = is.read(); b >= 0; b = is.read()) {
    ...
}
is.close();
```



The streaming API does **not** apply marshalling/unmarshalling to the values. For this reason you cannot access the same entries using both the streaming and non-streaming API at the same time, unless you provide your own marshaller to detect this situation.

The `InputStream` returned by the `RemoteStreamingCache.get(K key)` method implements the `VersionedMetadata` interface, so you can retrieve version and expiration information:

```
RemoteStreamingCache<String> streamingCache = remoteCache.streaming();
InputStream is = streamingCache.get("a_large_object");
int version = ((VersionedMetadata) is).getVersion();
for(int b = is.read(); b >= 0; b = is.read()) {
    ...
}
is.close();
```



Conditional write methods (`putIfAbsent`, `replace`) only perform the actual condition check once the value has been completely sent to the server (i.e. when the `close()` method has been invoked on the `OutputStream`).

20.6.10. Creating Event Listeners

Java Hot Rod clients can register listeners to receive cache-entry level events. Cache entry created, modified and removed events are supported.

Creating a client listener is very similar to embedded listeners, except that different annotations and event classes are used. Here's an example of a client listener that prints out each event received:

```

import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class EventPrintListener {

    @ClientCacheEntryCreated
    public void handleCreatedEvent(ClientCacheEntryCreatedEvent e) {
        System.out.println(e);
    }

    @ClientCacheEntryModified
    public void handleModifiedEvent(ClientCacheEntryModifiedEvent e) {
        System.out.println(e);
    }

    @ClientCacheEntryRemoved
    public void handleRemovedEvent(ClientCacheEntryRemovedEvent e) {
        System.out.println(e);
    }
}

```

`ClientCacheEntryCreatedEvent` and `ClientCacheEntryModifiedEvent` instances provide information on the affected key, and the version of the entry. This version can be used to invoke conditional operations on the server, such as `replaceWithVersion` or `removeWithVersion`.

`ClientCacheEntryRemovedEvent` events are only sent when the remove operation succeeds. In other words, if a remove operation is invoked but no entry is found or no entry should be removed, no event is generated. Users interested in removed events, even when no entry was removed, can develop event customization logic to generate such events. More information can be found in the [customizing client events section](#).

All `ClientCacheEntryCreatedEvent`, `ClientCacheEntryModifiedEvent` and `ClientCacheEntryRemovedEvent` event instances also provide a `boolean isCommandRetried()` method that will return true if the write command that caused this had to be retried again due to a topology change. This could be a sign that this event has been duplicated or another event was dropped and replaced (eg: `ClientCacheEntryModifiedEvent` replaced `ClientCacheEntryCreatedEvent`).

Once the client listener implementation has been created, it needs to be registered with the server. To do so, execute:

```

RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener());

```

20.6.11. Removing Event Listeners

When an client event listener is not needed any more, it can be removed:

```
EventPrintListener listener = ...
cache.removeClientListener(listener);
```

20.6.12. Filtering Events

In order to avoid inundating clients with events, users can provide filtering functionality to limit the number of events fired by the server for a particular client listener. To enable filtering, a cache event filter factory needs to be created that produces filter instances:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory;
import org.infinispan.filter.NamedFactory;

@NamedFactory(name = "static-filter")
class StaticCacheEventFilterFactory implements CacheEventFilterFactory {
    @Override
    public CacheEventFilterFactory<Integer, String> getFilter(Object[] params) {
        return new StaticCacheEventFilter();
    }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class StaticCacheEventFilter implements CacheEventFilter<Integer, String>,
Serializable {
    @Override
    public boolean accept(Integer key, String oldValue, Metadata oldMetadata,
        String newValue, Metadata newMetadata, EventType eventType) {
        if (key.equals(1)) // static key
            return true;

        return false;
    }
}
```

The cache event filter factory instance defined above creates filter instances which statically filter out all entries except the one whose key is **1**.

To be able to register a listener with this cache event filter factory, the factory has to be given a unique name, and the Hot Rod server needs to be plugged with the name and the cache event filter factory instance. Plugging the {brandname} Server with a custom filter involves the following steps:

1. Create a JAR file with the filter implementation within it.
2. Optional: If the cache uses custom key/value classes, these must be included in the JAR so that the callbacks can be executed with the correctly unmarshalled key and/or value instances. If the

client listener has `useRawData` enabled, this is not necessary since the callback key/value instances will be provided in binary format.

3. Create a META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory file within the JAR file and within it, write the fully qualified class name of the filter class implementation.
4. Deploy the JAR file in the {brandname} Server.

On top of that, the client listener needs to be linked with this cache event filter factory by adding the factory's name to the `@ClientListener` annotation:

```
@ClientListener(filterFactoryName = "static-filter")
public class EventPrintListener { ... }
```

And, register the listener with the server:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener());
```

Dynamic filter instances that filter based on parameters provided when the listener is registered are also possible. Filters use the parameters received by the filter factories to enable this option. For example:

```

import org.infinispan.notifications.cachelistener.filter.CacheEventFilterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventFilter;

class DynamicCacheEventFilterFactory implements CacheEventFilterFactory {
    @Override
    public CacheEventFilter<Integer, String> getFilter(Object[] params) {
        return new DynamicCacheEventFilter(params);
    }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class DynamicCacheEventFilter implements CacheEventFilter<Integer, String>,
Serializable {
    final Object[] params;

    DynamicCacheEventFilter(Object[] params) {
        this.params = params;
    }

    @Override
    public boolean accept(Integer key, String oldValue, Metadata oldMetadata,
        String newValue, Metadata newMetadata, EventType eventType) {
        if (key.equals(params[0])) // dynamic key
            return true;

        return false;
    }
}

```

The dynamic parameters required to do the filtering are provided when the listener is registered:

```

RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener(), new Object[]{1}, null);

```



Filter instances have to be marshallable when they are deployed in a cluster so that the filtering can happen right where the event is generated, even if the event is generated in a different node to where the listener is registered. To make them marshallable, either make them extend `Serializable`, `Externalizable`, or provide a custom `Externalizer` for them.

20.6.13. Customizing Events

The events generated by default contain just enough information to make the event relevant but they avoid cramming too much information in order to reduce the cost of sending them. Optionally, the information shipped in the events can be customised in order to contain more information, such as values, or to contain even less information. This customization is done with

`CacheEventConverter` instances generated by a `CacheEventConverterFactory`:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;
import org.infinispan.filter.NamedFactory;

@NamedFactory(name = "static-converter")
class StaticConverterFactory implements CacheEventConverterFactory {
    final CacheEventConverter<Integer, String, CustomEvent> staticConverter = new
    StaticCacheEventConverter();
    public CacheEventConverter<Integer, String, CustomEvent> getConverter(final
    Object[] params) {
        return staticConverter;
    }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers
// needed when running in a cluster
class StaticCacheEventConverter implements CacheEventConverter<Integer, String,
CustomEvent>, Serializable {
    public CustomEvent convert(Integer key, String oldValue, Metadata oldMetadata,
String newValue, Metadata newMetadata, EventType eventType) {
        return new CustomEvent(key, newValue);
    }
}

// Needs to be Serializable, Externalizable or marshallable with Infinispan
Externalizers
// regardless of cluster or local caches
static class CustomEvent implements Serializable {
    final Integer key;
    final String value;
    CustomEvent(Integer key, String value) {
        this.key = key;
        this.value = value;
    }
}
```

In the example above, the converter generates a new custom event which includes the value as well as the key in the event. This will result in bigger event payloads compared with default events, but if combined with filtering, it can reduce its network bandwidth cost.



The target type of the converter must be either `Serializable` or `Externalizable`. In this particular case of converters, providing an `Externalizer` will not work by default since the default Hot Rod client marshaller does not support them.

Handling custom events requires a slightly different client listener implementation to the one demonstrated previously. To be more precise, it needs to handle `ClientCacheEntryCustomEvent` instances:

```
import org.infinispan.client.hotrod.annotation.*;
import org.infinispan.client.hotrod.event.*;

@ClientListener
public class CustomEventPrintListener {

    @ClientCacheEntryCreated
    @ClientCacheEntryModified
    @ClientCacheEntryRemoved
    public void handleCustomEvent(ClientCacheEntryCustomEvent<CustomEvent> e) {
        System.out.println(e);
    }
}
```

The `ClientCacheEntryCustomEvent` received in the callback exposes the custom event via `getEventData` method, and the `getType` method provides information on whether the event generated was as a result of cache entry creation, modification or removal.

Similar to filtering, to be able to register a listener with this converter factory, the factory has to be given a unique name, and the Hot Rod server needs to be plugged with the name and the cache event converter factory instance. Plugging the {brandname} Server with an event converter involves the following steps:

1. Create a JAR file with the converter implementation within it.
2. Optional: If the cache uses custom key/value classes, these must be included in the JAR so that the callbacks can be executed with the correctly unmarshalled key and/or value instances. If the client listener has `useRawData` enabled, this is not necessary since the callback key/value instances will be provided in binary format.
3. Create a `META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory` file within the JAR file and within it, write the fully qualified class name of the converter class implementation.
4. Deploy the JAR file in the {brandname} Server.

On top of that, the client listener needs to be linked with this converter factory by adding the factory's name to the `@ClientListener` annotation:

```
@ClientListener(converterFactoryName = "static-converter")
public class CustomEventPrintListener { ... }
```

And, register the listener with the server:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new CustomEventPrintListener());
```


Dynamic converter instances that convert based on parameters provided when the listener is registered are also possible. Converters use the parameters received by the converter factories to enable this option. For example:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;

@NamedFactory(name = "dynamic-converter")
class DynamicCacheEventConverterFactory implements CacheEventConverterFactory {
    public CacheEventConverter<Integer, String, CustomEvent> getConverter(final
Object[] params) {
        return new DynamicCacheEventConverter(params);
    }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers needed
when running in a cluster
class DynamicCacheEventConverter implements CacheEventConverter<Integer, String,
CustomEvent>, Serializable {
    final Object[] params;

    DynamicCacheEventConverter(Object[] params) {
        this.params = params;
    }

    public CustomEvent convert(Integer key, String oldValue, Metadata oldMetadata,
String newValue, Metadata newMetadata, EventType eventType) {
        // If the key matches a key given via parameter, only send the key information
        if (params[0].equals(key))
            return new CustomEvent(key, null);

        return new CustomEvent(key, newValue);
    }
}
```

The dynamic parameters required to do the conversion are provided when the listener is registered:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new EventPrintListener(), null, new Object[]{1});
```



Converter instances have to be marshallable when they are deployed in a cluster, so that the conversion can happen right where the event is generated, even if the event is generated in a different node to where the listener is registered. To make them marshallable, either make them extend `Serializable`, `Externalizable`, or provide a custom `Externalizer` for them.

20.6.14. Filter and Custom Events

If you want to do both event filtering and customization, it's easier to implement `org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter` which allows both filter and customization to happen in a single step. For convenience, it's recommended to extend `org.infinispan.notifications.cachelistener.filter.AbstractCacheEventFilterConverter` instead of implementing `org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverter` directly. For example:

```
import org.infinispan.notifications.cachelistener.filter.CacheEventConverterFactory;
import org.infinispan.notifications.cachelistener.filter.CacheEventConverter;

@NamedFactory(name = "dynamic-filter-converter")
class DynamicCacheEventFilterConverterFactory implements
CacheEventConverterFactory {
    public CacheEventFilterConverter<Integer, String, CustomEvent> getFilterConverter
(final Object[] params) {
        return new DynamicCacheEventFilterConverter(params);
    }
}

// Serializable, Externalizable or marshallable with Infinispan Externalizers needed
// when running in a cluster
//
class DynamicCacheEventFilterConverter extends AbstractCacheEventFilterConverter
<Integer, String, CustomEvent>, Serializable {
    final Object[] params;

    DynamicCacheEventFilterConverter(Object[] params) {
        this.params = params;
    }

    public CustomEvent filterAndConvert(Integer key, String oldValue, Metadata
oldMetadata,
        String newValue, Metadata newMetadata, EventType eventType) {
        // If the key matches a key given via parameter, only send the key information
        if (params[0].equals(key))
            return new CustomEvent(key, null);

        return new CustomEvent(key, newValue);
    }
}
```

Similar to filters and converters, to be able to register a listener with this combined filter/converter factory, the factory has to be given a unique name via the `@NamedFactory` annotation, and the Hot Rod server needs to be plugged with the name and the cache event converter factory instance. Plugging the {brandname} Server with an event converter involves the following steps:

1. Create a JAR file with the converter implementation within it.

- Optional: If the cache uses custom key/value classes, these must be included in the JAR so that the callbacks can be executed with the correctly unmarshalled key and/or value instances. If the client listener has `useRawData` enabled, this is not necessary since the callback key/value instances will be provided in binary format.
- Create a `META-INF/services/org.infinispan.notifications.cachelistener.filter.CacheEventFilterConverterFactory` file within the JAR file and within it, write the fully qualified class name of the converter class implementation.
- Deploy the JAR file in the {brandname} Server.

From a client perspective, to be able to use the combined filter and converter class, the client listener must define the same filter factory and converter factory names, e.g.:

```
@ClientListener(filterFactoryName = "dynamic-filter-converter", converterFactoryName =
"dynamic-filter-converter")
public class CustomEventPrintListener { ... }
```

The dynamic parameters required in the example above are provided when the listener is registered via either filter or converter parameters. If filter parameters are non-empty, those are used, otherwise, the converter parameters:

```
RemoteCache<?, ?> cache = ...
cache.addClientListener(new CustomEventPrintListener(), new Object[]{1}, null);
```

20.6.15. Event Marshalling

Hot Rod servers can store data in different formats, but in spite of that, Java Hot Rod client users can still develop `CacheEventConverter` or `CacheEventFilter` instances that work on typed objects. By default, filters and converter will use data as POJO (application/x-java-object) but it is possible to override the desired format by overriding the method `format()` from the filter/converter. If the format returns `null`, the filter/converter will receive data as it's stored.

As indicated in the [Marshalling Data](#) section, Hot Rod Java clients can be configured to use a different `org.infinispan.commons.marshall.Marshaller` instance. If doing this and deploying `CacheEventConverter` or `CacheEventFilter` instances, to be able to present filters/converter with Java Objects rather than marshalled content, the server needs to be able to convert between objects and the binary format produced by the marshaller.

To deploy a Marshaller instance server-side, follow a similar method to the one used to deploy `CacheEventConverter` or `CacheEventFilter` instances:

- Create a JAR file with the converter implementation within it.
- Create a `META-INF/services/org.infinispan.commons.marshall.Marshaller` file within the JAR file and within it, write the fully qualified class name of the marshaller class implementation.
- Deploy the JAR file in the {brandname} Server.

Note that the Marshaller could be deployed in either a separate jar, or in the same jar as the `CacheEventConverter` and/or `CacheEventFilter` instances.

Deploying Protostream Marshallers

If a cache stores protobuf content, as it happens when using protostream marshaller in the Hot Rod client, it's not necessary to deploy a custom marshaller since the format is already supported by the server: there are transcoders from protobuf format to most common formats like JSON and POJO.

When using filters/converters with those caches, and it's desirable to use filter/converters with Java Objects rather than binary protobuf data, it's necessary to deploy the extra protostream marshallers so that the server can unmarshall the data before filtering/converting. To do so, follow these steps:

1. Create a jar and include an implementation of the interface `org.infinispan.query.remote.client.ProtostreamSerializationContextInitializer`, adding extra marshallers and optionally extra protobuf files to the cache manager's Serialization context.
2. Create a `META-INF/services/org.infinispan.query.remote.client.ProtostreamSerializationContextInitializer` file within the JAR file containing the fully qualified class name of the `ProtostreamSerializationContextInitializer` class implementation.
3. Create a `META-INF/MANIFEST.MF` with `Dependencies: org.infinispan.protostream, org.infinispan.remote-query.client`
4. Deploy the JAR file in the {brandname} Server in the `standalone/deployments` folder
5. Configure this deployment in the desired cache manager:

```
<cache-container name="local" default-cache="default">
  <modules>
    <module name="deployment.my-file.jar"/>
  </modules>
  ...
</cache-container>
```



The deployment must be available during the server startup!

20.6.16. Listener State Handling

Client listener annotation has an optional `includeCurrentState` attribute that specifies whether state will be sent to the client when the listener is added or when there's a failover of the listener.

By default, `includeCurrentState` is false, but if set to true and a client listener is added in a cache already containing data, the server iterates over the cache contents and sends an event for each entry to the client as a `ClientCacheEntryCreated` (or custom event if configured). This allows clients to build some local data structures based on the existing content. Once the content has been iterated over, events are received as normal, as cache updates are received. If the cache is clustered, the entire cluster wide contents are iterated over.

`includeCurrentState` also controls whether state is received when the node where the client event

listener is registered fails and it's moved to a different node. The next section discusses this topic in depth.

20.6.17. Listener Failure Handling

When a Hot Rod client registers a client listener, it does so in a single node in a cluster. If that node fails, the Java Hot Rod client detects that transparently and fails over all listeners registered in the node that failed to another node.

During this fail over the client might miss some events. To avoid missing these events, the client listener annotation contains an optional parameter called `includeCurrentState` which if set to true, when the failover happens, the cache contents can iterated over and `ClientCacheEntryCreated` events (or custom events if configured) are generated. By default, `includeCurrentState` is set to false.

Java Hot Rod clients can be made aware of such fail over event by adding a callback to handle it:

```
@ClientCacheFailover
public void handleFailover(ClientCacheFailoverEvent e) {
    ...
}
```

This is very useful in use cases where the client has cached some data, and as a result of the fail over, taking in account that some events could be missed, it could decide to clear any locally cached data when the fail over event is received, with the knowledge that after the fail over event, it will receive events for the contents of the entire cache.

20.6.18. Near Caching

The Java Hot Rod client can be optionally configured with a near cache, which means that the Hot Rod client can keep a local cache that stores recently used data. Enabling near caching can significantly improve the performance of read operations `get` and `getVersioned` since data can potentially be located locally within the Hot Rod client instead of having to go remote.

To enable near caching, the user must set the near cache mode to `INVALIDATED`. By doing that near cache is populated upon retrievals from the server via calls to `get` or `getVersioned` operations. When near cached entries are updated or removed server-side, the cached near cache entries are invalidated. If a key is requested after it's been invalidated, it'll have to be re-fetched from the server.



You should not use `maxIdle` expiration with near caches, as near-cache reads will not propagate the last access change to the server and to the other clients.

When near cache is enabled, its size must be configured by defining the maximum number of entries to keep in the near cache. When the maximum is reached, near cached entries are evicted using a least-recently-used (LRU) algorithm. If providing 0 or a negative value, it is assumed that the near cache is unbounded.



Users should be careful when configuring near cache to be unbounded since it shifts the responsibility to keep the near cache's size within the boundaries of the client JVM to the user.

The Hot Rod client's near cache mode is configured using the `NearCacheMode` enumeration and calling:

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

// Unbounded invalidated near cache
ConfigurationBuilder unbounded = new ConfigurationBuilder();
unbounded.nearCache().mode(NearCacheMode.INVALIDATED).maxEntries(-1);

// Bounded invalidated near cache
ConfigurationBuilder bounded = new ConfigurationBuilder();
bounded.nearCache().mode(NearCacheMode.INVALIDATED).maxEntries(100);
```

Since the configuration is shared by all caches obtained from a single `RemoteCacheManager`, you may not want to enable near-caching for all of them. You can use the `cacheNamePattern` configuration attribute to define a regular expression which matches the names of the caches for which you want near-caching. Caches whose name don't match the regular expression, will not have near-caching enabled.

```
// Bounded invalidated near cache with pattern matching
ConfigurationBuilder bounded = new ConfigurationBuilder();
bounded.nearCache()
    .mode(NearCacheMode.INVALIDATED)
    .maxEntries(100)
    .cacheNamePattern("near.*"); // enable near-cache only for caches whose name starts
with 'near'
```



Near caches work the same way for local caches as they do for clustered caches, but in a clustered cache scenario, if the server node sending the near cache notifications to the Hot Rod client goes down, the Hot Rod client transparently fails over to another node in the cluster, clearing the near cache along the way.

20.6.19. Unsupported methods

Some of the `Cache` methods are not being supported by the `RemoteCache`. Calling one of these methods results in an `UnsupportedOperationException` being thrown. Most of these methods do not make sense on the remote cache (e.g. listener management operations), or correspond to methods that are not supported by local cache as well (e.g. `containsValue`). Another set of unsupported operations are some of the atomic operations inherited from `ConcurrentMap`:

```
boolean remove(Object key, Object value);
boolean replace(Object key, Object value);
boolean replace(Object key, Object oldValue, Object value);
```

[RemoteCache](#) offers alternative versioned methods for these atomic operations, that are also network friendly, by not sending the whole value object over the network, but a version identifier. See the section on versioned API.

Each one of these unsupported operation is documented in the [RemoteCache](#) javadoc.

20.6.20. Return values

There is a set of methods that alter a cached entry and return the previous existing value, e.g.:

```
V remove(Object key);
V put(K key, V value);
```

By default on RemoteCache, these operations return null even if such a previous value exists. This approach reduces the amount of data sent over the network. However, if these return values are needed they can be enforced on a per invocation basis using flags:

```
cache.put("aKey", "initialValue");
assert null == cache.put("aKey", "aValue");
assert "aValue".equals(cache.withFlags(Flag.FORCE_RETURN_VALUE).put("aKey",
    "newValue"));
```

This default behavior can be changed through `force-return-value=true` configuration parameter (see configuration section below).

20.6.21. Hot Rod Transactions

You can configure and use Hot Rod clients in JTA [Transactions](#).

To participate in a transaction, the Hot Rod client requires the [TransactionManager](#) with which it interacts and whether it participates in the transaction through the [Synchronization](#) or [XAResource](#) interface.



Transactions are optimistic in that clients acquire write locks on entries during the prepare phase. To avoid data inconsistency, be sure to read about [Detecting Conflicts with Transactions](#).

Configuring the Server

Caches in the server must also be transactional for clients to participate in JTA [Transactions](#).

The following server configuration is required, otherwise transactions rollback only:

- Isolation level must be **REPEATABLE_READ**.
- Locking mode must be **PESSIMISTIC**. In a future release, **OPTIMISTIC** locking mode will be supported.
- Transaction mode should be **NON_XA** or **NON_DURABLE_XA**. Hot Rod transactions cannot use **FULL_XA** because it degrades performance.

Hot Rod transactions have their own recovery mechanism.

Configuring Hot Rod Clients

When you create the [RemoteCacheManager](#), you can set the default [TransactionManager](#) and [TransactionMode](#) that the [RemoteCache](#) uses.

The [RemoteCacheManager](#) lets you create only one configuration for transactional caches, as in the following example:

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
//other client configuration parameters
cb.transaction().transactionManagerLookup(GenericTransactionManagerLookup.getInstance(
));
cb.transaction().transactionMode(TransactionMode.NON_XA);
cb.transaction().timeout(1, TimeUnit.MINUTES)
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```

The preceding configuration applies to all instances of a remote cache. If you need to apply different configurations to remote cache instances, you can override the [RemoteCache](#) configuration. See [Overriding RemoteCacheManager Configuration](#).

See [ConfigurationBuilder](#) Javadoc for documentation on configuration parameters.

You can also configure the Java Hot Rod client with a properties file, as in the following example:

```
infinispan.client.hotrod.transaction.transaction_manager_lookup =
org.infinispan.client.hotrod.transaction.lookup.GenericTransactionManagerLookup
infinispan.client.hotrod.transaction.transaction_mode = NON_XA
infinispan.client.hotrod.transaction.timeout = 60000
```

TransactionManagerLookup Interface

TransactionManagerLookup provides an entry point to fetch a [TransactionManager](#).

Available implementations of **TransactionManagerLookup**:

GenericTransactionManagerLookup

A lookup class that locates [TransactionManagers](#) running in Java EE application servers. Defaults to the [RemoteTransactionManager](#) if it cannot find a [TransactionManager](#).



In most cases, [GenericTransactionManagerLookup](#) is suitable. However, you can implement the [TransactionManagerLookup](#) interface if you need to integrate a custom [TransactionManager](#).

RemoteTransactionManagerLookup

A basic, and volatile, [TransactionManager](#) if no other implementation is available. Note that this implementation has significant limitations when handling concurrent transactions and recovery.

Transaction Modes

[TransactionMode](#) controls how a [RemoteCache](#) interacts with the [TransactionManager](#).



Configure transaction modes on both the {brandname} server and your client application. If clients attempt to perform transactional operations on non-transactional caches, runtime exceptions can occur.

Transaction modes are the same in both the {brandname} configuration and client settings. Use the following modes with your client, see the {brandname} configuration schema for the server:

NONE

The [RemoteCache](#) does not interact with the [TransactionManager](#). This is the default mode and is non-transactional.

NON_XA

The [RemoteCache](#) interacts with the [TransactionManager](#) via [Synchronization](#).

NON_DURABLE_XA

The [RemoteCache](#) interacts with the [TransactionManager](#) via [XAResource](#). Recovery capabilities are disabled.

FULL_XA

The [RemoteCache](#) interacts with the [TransactionManager](#) via [XAResource](#). Recovery capabilities are enabled. Invoke the [XAResource.recover\(\)](#) method to retrieve transactions to recover.

Overriding Configuration for Cache Instances

Because [RemoteCacheManager](#) does not support different configurations for each cache instance. However, [RemoteCacheManager](#) includes the [getCache\(String\)](#) method that returns the [RemoteCache](#) instances and lets you override some configuration parameters, as follows:

```
getCache(String cacheName, TransactionMode transactionMode)
```

Returns a [RemoteCache](#) and overrides the configured [TransactionMode](#).

```
getCache(String cacheName, boolean forceReturnValue, TransactionMode transactionMode)
```

Same as previous, but can also force return values for write operations.

```
getCache(String cacheName, TransactionManager transactionManager)
```

Returns a [RemoteCache](#) and overrides the configured [TransactionManager](#).

`getCache(String cacheName, boolean forceReturnValue, TransactionManager transactionManager)`

Same as previous, but can also force return values for write operations.

`getCache(String cacheName, TransactionMode transactionMode, TransactionManager transactionManager)`

Returns a `RemoteCache` and overrides the configured `TransactionManager` and `TransactionMode`. Uses the configured values, if `transactionManager` or `transactionMode` is null.

`getCache(String cacheName, boolean forceReturnValue, TransactionMode transactionMode, TransactionManager transactionManager)`

Same as previous, but can also force return values for write operations.



The `getCache(String)` method returns `RemoteCache` instances regardless of whether they are transaction or not. `RemoteCache` includes a `getTransactionManager()` method that returns the `TransactionManager` that the cache uses. If the `RemoteCache` is not transactional, the method returns `null`.

Detecting Conflicts with Transactions

Transactions use the initial values of keys to detect conflicts. For example, "k" has a value of "v" when a transaction begins. During the prepare phase, the transaction fetches "k" from the server to read the value. If the value has changed, the transaction rolls back to avoid a conflict.



Transactions use versions to detect changes instead of checking value equality.

The `forceReturnValue` parameter controls write operations to the `RemoteCache` and helps avoid conflicts. It has the following values:

- If `true`, the `TransactionManager` fetches the most recent value from the server before performing write operations. However, the `forceReturnValue` parameter applies only to write operations that access the key for the first time.
- If `false`, the `TransactionManager` does not fetch the most recent value from the server before performing write operations. Because this setting



This parameter does not affect *conditional* write operations such as `replace` or `putIfAbsent` because they require the most recent value.

The following transactions provide an example where the `forceReturnValue` parameter can prevent conflicting write operations:

Transaction 1 (TX1)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.put("k", "v1");
tm.commit();
```

Transaction 2 (TX2)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.put("k", "v2");
tm.commit();
```

In this example, TX1 and TX2 are executed in parallel. The initial value of "k" is "v".

- If `forceReturnValue = true`, the `cache.put()` operation fetches the value for "k" from the server in both TX1 and TX2. The transaction that acquires the lock for "k" first then commits. The other transaction rolls back during the commit phase because the transaction can detect that "k" has a value other than "v".
- If `forceReturnValue = false`, the `cache.put()` operation does not fetch the value for "k" from the server and returns null. Both TX1 and TX2 can successfully commit, which results in a conflict. This occurs because neither transaction can detect that the initial value of "k" changed.

The following transactions include `cache.get()` operations to read the value for "k" before doing the `cache.put()` operations:

Transaction 1 (TX1)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.get("k");
cache.put("k", "v1");
tm.commit();
```

Transaction 2 (TX2)

```
RemoteCache<String, String> cache = ...
TransactionManager tm = ...

tm.begin();
cache.get("k");
cache.put("k", "v2");
tm.commit();
```

In the preceding examples, TX1 and TX2 both read the key so the `forceReturnValue` parameter does not take effect. One transaction commits, the other rolls back. However, the `cache.get()` operation requires an additional server request. If you do not need the return value for the `cache.put()` operation that server request is inefficient.

Using the Configured Transaction Manager and Transaction Mode

The following example shows how to use the `TransactionManager` and `TransactionMode` that you configure in the `RemoteCacheManager`:

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//The my-cache instance uses the RemoteCacheManager configuration.
RemoteCache<String, String> cache = rcm.getCache("my-cache");

//Return the transaction manager that the cache uses.
TransactionManager tm = cache.getTransactionManager();

//Perform a simple transaction.
tm.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
tm.commit();
```

Overriding the Transaction Manager

The following example shows how to override `TransactionManager` with the `getCache` method:

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//Define a custom TransactionManager.
TransactionManager myCustomTM = ...

//Override the TransactionManager for the my-cache instance. Use the default
configuration if null is returned.
RemoteCache<String, String> cache = rcm.getCache("my-cache", null, myCustomTM);

//Perform a simple transaction.
myCustomTM.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
myCustomTM.commit();
```

Overriding the Transaction Mode

The following example shows how to override `TransactionMode` with the `getCache` method:

```
//Configure the transaction manager and transaction mode.
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb = new org
.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.transaction().transactionManagerLookup(RemoteTransactionManagerLookup.getInstance()
);
cb.transaction().transactionMode(TransactionMode.NON_XA);

RemoteCacheManager rcm = new RemoteCacheManager(cb.build());

//Override the transaction mode for the my-cache instance.
RemoteCache<String, String> cache = rcm.getCache("my-cache", TransactionMode
.NON_DURABLE_XA, null);

//Return the transaction manager that the cache uses.
TransactionManager tm = cache.getTransactionManager();

//Perform a simple transaction.
tm.begin();
cache.put("k1", "v1");
System.out.println("K1 value is " + cache.get("k1"));
tm.commit();
```

20.6.22. Client Intelligence

HotRod defines three level of intelligence for the clients:

1. basic client, interested in neither cluster nor hash information
2. topology-aware client, interested in cluster information
3. hash-distribution-aware client, that is interested in both cluster and hash information

The java client supports all 3 levels of intelligence. It is transparently notified whenever a new server is added/removed from the HotRod cluster. At startup it only needs to know the address of one HotRod server (ip:host). On connection to the server the cluster topology is piggybacked to the client, and all further requests are being dispatched to all available servers. Any further topology change is also piggybacked.

Distribution-aware client

Another aspect of the 3rd level of intelligence is the fact that it is hash-distribution-aware. This means that, for each operation, the client chooses the most appropriate remote server to go to: the data owner. As an example, for a put(k,v) operation, the client calculates k's hash value and knows exactly on which server the data resides on. Then it picks up a tcp connection to that particular server and dispatches the operation to it. This means less burden on the server side which would otherwise need to lookup the value based on the key's hash. It also results in a quicker response from the server, as an additional network roundtrip is skipped. This hash-distribution-aware aspect is only relevant to the distributed HotRod clusters and makes no difference for replicated server deployments.

20.6.23. Request Balancing

Request balancing is only relevant when the server side is configured with replicated {brandname} cluster (on distributed clusters the hash-distribution-aware client logic is used, as discussed in the previous paragraph). Because the client is topology-aware, it knows the list of available servers at all the time. Request balancing has to do with how the client dispatches requests to the available servers.

The default strategy is round-robin: requests are being dispatched to all existing servers in a circular manner. E.g. given a cluster of servers {s1, s2, s3} here is how request will be dispatched:

```
CacheContainer cacheContainer = new RemoteCacheManager();
Cache<String, String> cache = cacheContainer.getCache();

cache.put("key1", "aValue"); //this goes to s1
cache.put("key2", "aValue"); //this goes to s2
String value = cache.get("key1"); //this goes to s3

cache.remove("key2"); //this is dispatched to s1 again, and so on...
```

Custom types of balancing policies can be defined by implementing the [FailoverRequestBalancingStrategy](#) and by specifying it through the `infinispan.client.hotrod.request-`

balancing-strategy configuration property. Please refer to configuration section for more details on this.

20.6.24. Persistent connections

In order to avoid creating a TCP connection on each request (which is a costly operation), the client keeps a pool of persistent connections to all the available servers and it reuses these connections whenever it is possible. The validity of the connections is checked using an async thread that iterates over the connections in the pool and sends a HotRod ping command to the server. By using this connection validation process the client is being proactive: there's a high chance for broken connections to be found while being idle in the pool and not on actual request from the application.

The number of connections per server, total number of connections, how long should a connection be kept idle in the pool before being closed - all these (and more) can be configured. Please refer to the javadoc of [RemoteCacheManager](#) for a list of all possible configuration elements.

20.6.25. Marshalling data

The Hot Rod client allows one to plug in a custom marshaller for transforming user objects into byte arrays and the other way around. This transformation is needed because of Hot Rod's binary nature - it doesn't know about objects.

The marshaller can be plugged through the "marshaller" configuration element (see Configuration section): the value should be the fully qualified name of a class implementing the [Marshaller](#) interface. This is an optional parameter, if not specified it defaults to the [GenericJBossMarshaller](#) - a highly optimized implementation based on the [JBoss Marshalling](#) library.

Since version 6.0, there's a new marshaller available to Java Hot Rod clients based on Protostream which generates portable payloads. You can find more information about it [here](#).

WARNING: If developing your own custom marshaller, take care of potential injection attacks.

To avoid such attacks, make the marshaller verify that any class names read, before instantiating it, is amongst the expected/allowed class names.

The client configuration can be enhanced with a list of regular expressions for classes that are allowed to be read.

WARNING: These checks are opt-in, so if not configured, any class can be read.

In the example below, only classes with fully qualified names containing **Person** or **Employee** would be allowed:

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

...
ConfigurationBuilder configBuilder = ...
configBuilder.addJavaSerialWhitelist(".*Person.*", ".*Employee.*");
```

20.6.26. Reading data in different data formats

By default, every Hot Rod client operation will use the configured marshaller when reading and writing from the server for both keys and values. See [Marshalling Data](#). Using the DataFormat API, it's possible to decorate remote caches so that all operations can happen with a custom data format.

Using different marshallers for Key and Values

Marshallers for Keys and Values can be overridden at run time. For example, to bypass all serialization in the Hot Rod client and read the byte[] as they are stored in the server:

```
// Existing Remote cache instance
RemoteCache<String, Pojo> remoteCache = ...

// IdentityMarshaller is a no-op marshaller
DataFormat rawKeyAndValues = DataFormat.builder().keyMarshaller(IdentityMarshaller
.INSTANCE).valueMarshaller(IdentityMarshaller.INSTANCE).build();

// Will create a new instance of RemoteCache with the supplied DataFormat
RemoteCache<byte[], byte[]> rawResultsCache = remoteCache.withDataFormat
(rawKeyAndValues);
```

Reading data in different formats

Apart from defining custom key and value marshallers, it's also possible to request/send data in different formats specified by a [org.infinispan.commons.dataconversion.MediaType](#):

```
// Existing remote cache using ProtostreamMarshaller
RemoteCache<String, Pojo> protobufCache = ...

// Request values returned as JSON, using the UTF8StringMarshaller that converts
between UTF-8 to String:
DataFormat jsonString = DataFormat.builder().valueType(MediaType.APPLICATION_JSON)
.valueMarshaller(new UTF8StringMarshaller().build());

RemoteCache<String, String> jsonStrCache = remoteCache.withDataFormat(jsonString);

// Alternatively, it's possible to request JSON values but marshalled/unmarshalled
with a custom value marshaller that returns `org.codehaus.jackson.JsonNode` objects:
DataFormat jsonNode = DataFormat.builder().valueType(MediaType.APPLICATION_JSON)
.valueMarshaller(new CustomJacksonMarshaller().build());

RemoteCache<String, JsonNode> jsonNodeCache = remoteCache.withDataFormat(jsonNode);
```



The data conversions happen in the server, and if it doesn't support converting from the storage format to the request format and vice versa, an error will be returned. For more details on server data format configuration and supported conversions, see [here](#).



Using differentmarshallers and formats for the key, with `.keyMarshaller()` and `.keyType()` may interfere with the client intelligence routing mechanism, causing it to contact the server that is not the owner of the key during Hot Rod operations. This will not result in errors but can result in extra hops inside the cluster to execute the operation. If performance is critical, make sure to use the keys in the format stored by the server.

20.6.27. Statistics

Various server usage statistics can be obtained through the `RemoteCache.stats()` method. This returns a `ServerStatistics` object - please refer to javadoc for details on the available statistics.

20.6.28. Multi-Get Operations

The Java Hot Rod client does not provide multi-get functionality out of the box but clients can build it themselves with the given APIs.

20.6.29. Failover capabilities

Hot Rod clients' capabilities to keep up with topology changes helps with request balancing and more importantly, with the ability to failover operations if one or several of the servers fail.

Some of the conditional operations mentioned above, including `putIfAbsent`, `replace` with and without version, and conditional `remove` have strict method return guarantees, as well as those operations where returning the previous value is forced.

In spite of failures, these methods return values need to be guaranteed, and in order to do so, it's necessary that these methods are not applied partially in the cluster in the event of failure. For example, imagine a `replace()` operation called in a server for key=k1 with `Flag.FORCE_RETURN_VALUE`, whose current value is `A` and the replace wants to set it to `B`. If the replace fails, it could happen that some servers contain `B` and others contain `A`, and during the failover, the original `replace()` could end up returning `B`, if the replace failovers to a node where `B` is set, or could end up returning `A`.

To avoid this kind of situations, whenever Java Hot Rod client users want to use conditional operations, or operations whose previous value is required, it's important that the cache is configured to be transactional in order to avoid incorrect conditional operations or return values.

20.6.30. Site Cluster Failover

On top of the in-cluster failover, Hot Rod clients are also able to failover to different clusters, which could be represented as an independent site.

The way site cluster failover works is that if all the main cluster nodes are not available, the client checks to see if any other clusters have been defined in which cases it tries to failover to the alternative cluster. If the failover succeeds, the client will remain connected to the alternative cluster until this becomes unavailable, in which case it'll try any other clusters defined, and ultimately, it'll try the original server settings.

To configure a cluster in the Hot Rod client, one host/port pair details must be provided for each of

the clusters configured. For example:

```
org.infinispan.client.hotrod.configuration.ConfigurationBuilder cb
    = new org.infinispan.client.hotrod.configuration.ConfigurationBuilder();
cb.addCluster().addClusterNode("remote-cluster-host", 11222);
RemoteCacheManager rmc = new RemoteCacheManager(cb.build());
```



Remember that regardless of the cluster definitions, the initial server(s) configuration must be provided unless the initial servers can be resolved using the default server host and port details.

20.6.31. Manual Site Cluster Switch

As well as supporting automatic site cluster failover, Java Hot Rod clients can also switch between site clusters manually by calling `RemoteCacheManager`'s `switchToCluster(clusterName)` and `switchToDefaultCluster()`.

Using `switchToCluster(clusterName)`, users can force a client to switch to one of the clusters pre-defined in the Hot Rod client configuration. To switch to the initial servers defined in the client configuration, call `switchToDefaultCluster()`.

20.6.32. Monitoring the Hot Rod client

The Hot Rod client can be monitored and managed via JMX similarly to what is described in the [Management](#) chapter. By enabling statistics, an MBean will be registered for the `RemoteCacheManager` as well as for each `RemoteCache` obtained through it. Through these MBeans it is possible to obtain statistics about remote and near-cache hits/misses and connection pool usage.

20.6.33. Concurrent Updates

Data structures, such as {brandname} `Cache`, that are accessed and modified concurrently can suffer from data consistency issues unless there're mechanisms to guarantee data correctness. {brandname} `Cache`, since it implements `ConcurrentMap`, provides operations such as `conditional replace`, `putIfAbsent`, and `conditional remove` to its clients in order to guarantee data correctness. It even allows clients to operate against cache instances within JTA transactions, hence providing the necessary data consistency guarantees.

However, when it comes to [Hot Rod protocol](#) backed servers, clients do not yet have the ability to start remote transactions but they can call instead versioned operations to mimic the conditional methods provided by the embedded {brandname} cache instance API. Let's look at a real example to understand how it works.

Data Consistency Problem

Imagine you have two ATMs that connect using Hot Rod to a bank where an account's balance is stored. Two closely followed operations to retrieve the latest balance could return 500 CHF (swiss francs) as shown below:

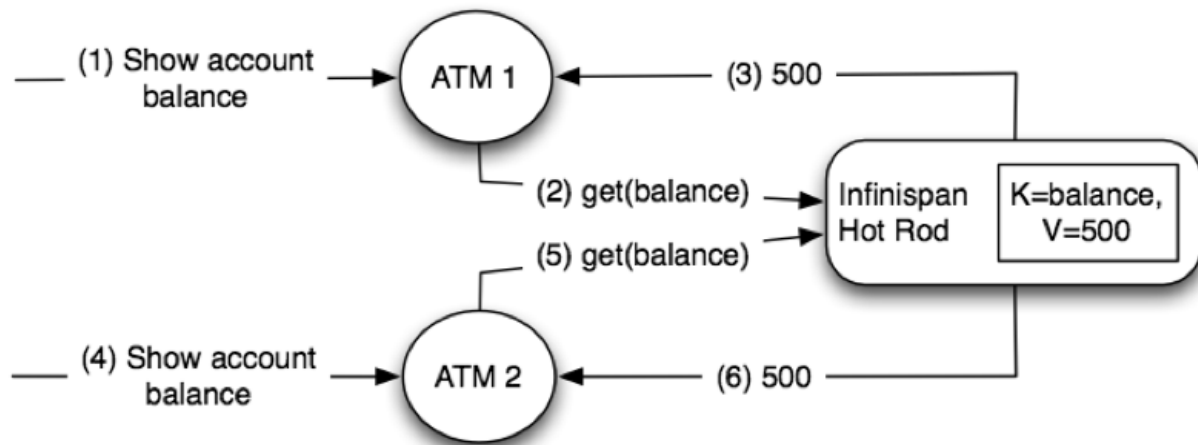
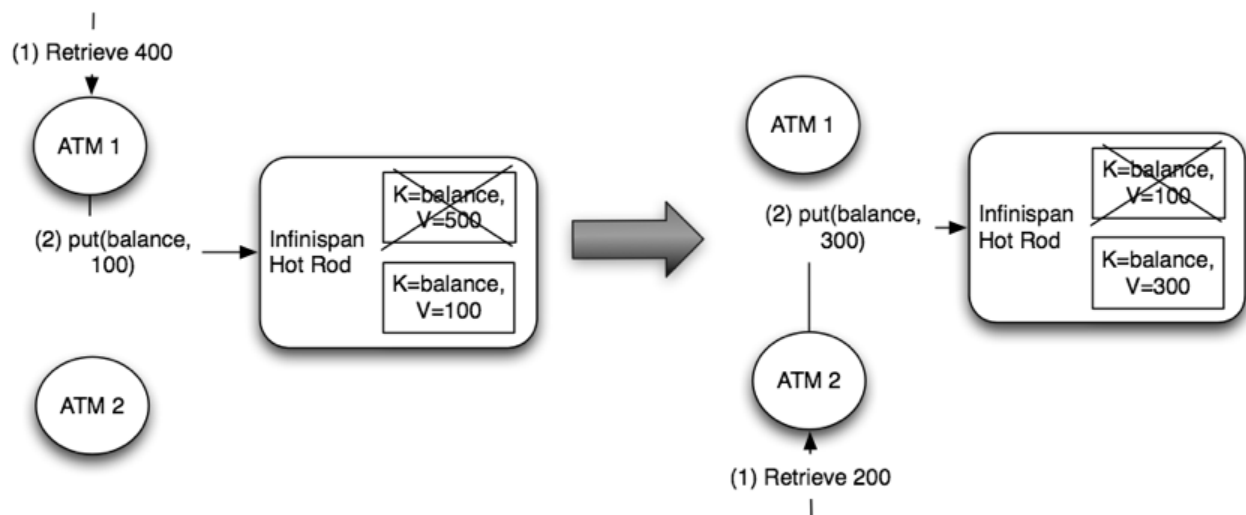


Figure 17. Concurrent readers

Next a customer connects to the first ATM and requests 400 CHF to be retrieved. Based on the last value read, the ATM could calculate what the new balance is, which is 100 CHF, and request a put with this new value. Let's imagine now that around the same time another customer connects to the ATM and requests 200 CHF to be retrieved. Let's assume that the ATM thinks it has the latest balance and based on its calculations it sets the new balance to 300 CHF:



Obviously, this would be wrong. Two concurrent updates have resulted in an incorrect account balance. The second update should not have been allowed since the balance the second ATM had was incorrect. Even if the ATM would have retrieved the balance before calculating the new balance, someone could have updated between the new balance being retrieved and the update. Before finding out how to solve this issue in a client-server scenario with Hot Rod, let's look at how this is solved when {brandname} clients run in peer-to-peer mode where clients and {brandname} live within the same JVM.

Embedded-mode Solution

If the ATM and the {brandname} instance storing the bank account lived in the same JVM, the ATM could use the [conditional replace API](#) referred at the beginning of this article. So, it could send the previous known value to verify whether it has changed since it was last read. By doing so, the first

operation could double check that the balance is still 500 CHF when it was to update to 100 CHF. Now, when the second operation comes, the current balance would not be 500 CHF any more and hence the conditional replace call would fail, hence avoiding data consistency issues:

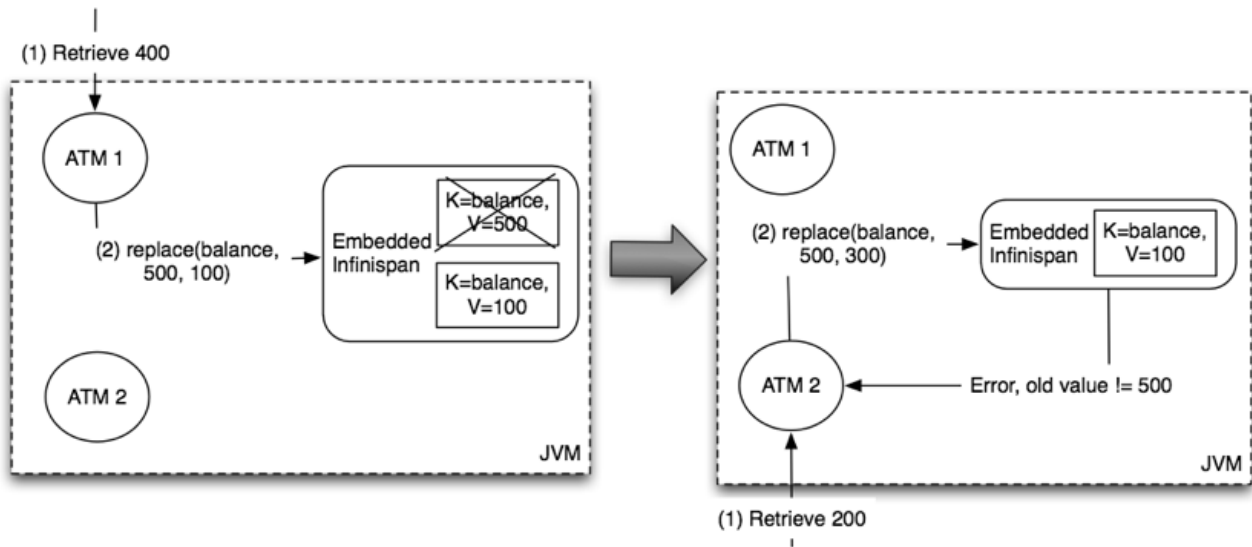


Figure 18. P2P solution

Client-Server Solution

In theory, Hot Rod could use the same p2p solution but sending the previous value would be not practical. In this example, the previous value is just an integer but the value could be a lot bigger and hence forcing clients to send it to the server would be rather wasteful. Instead, Hot Rod offers versioned operations to deal with this situation.

Basically, together with each key/value pair, Hot Rod stores a version number which uniquely identifies each modification. So, using an operation called [getVersioned](#) or [getWithVersion](#), clients can retrieve not only the value associated with a key, but also the current version. So, if we look at the previous example once again, the ATMs could call [getVersioned](#) and get the balance's version:

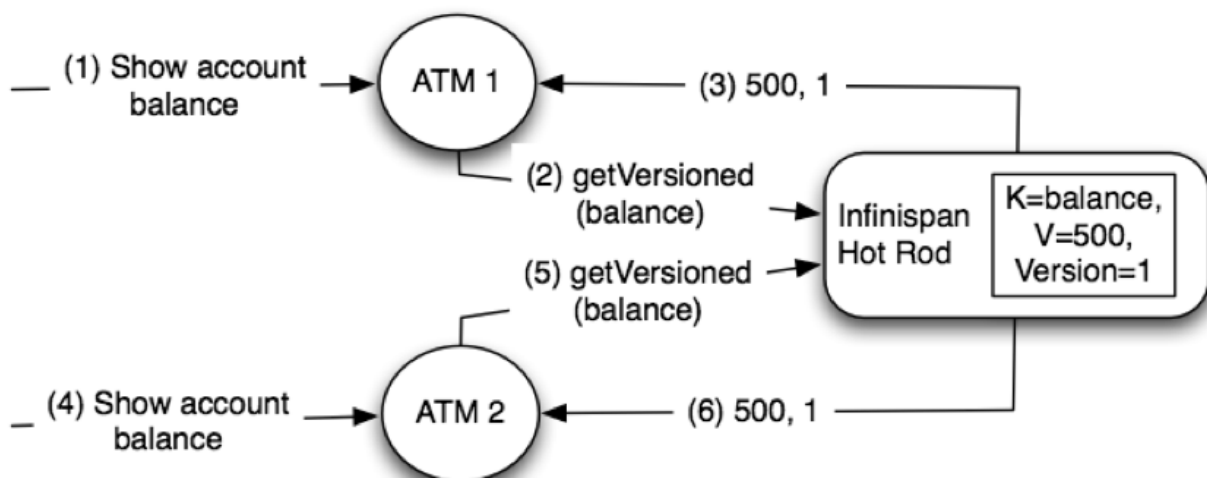


Figure 19. Get versioned

When the ATMs wanted to modify the balance, instead of just calling put, they could call

[replaceIfUnmodified](#) operation passing the latest version number of which the clients are aware of. The operation will only succeed if the version passed matches the version in the server. So, the first modification by the ATM would be allowed since the client passes 1 as version and the server side version for the balance is also 1. On the other hand, the second ATM would not be able to make the modification because after the first ATM's modification the version would have been incremented to 2, and now the passed version (1) and the server side version (2) would not match:

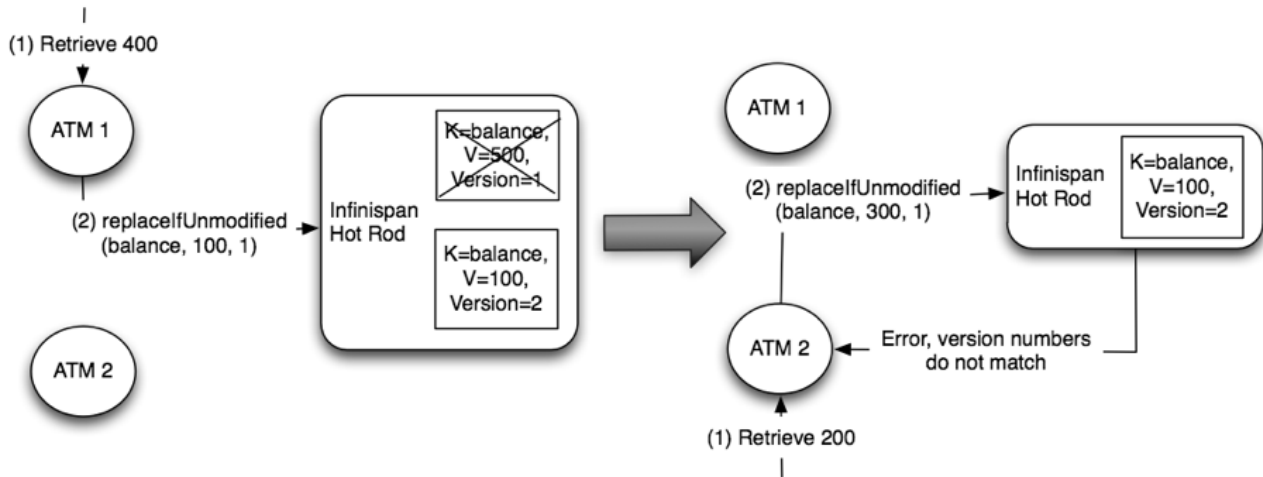


Figure 20. Replace if versions match

20.6.34. Javadocs

It is highly recommended to read the following Javadocs (this is pretty much all the public API of the client):

- [RemoteCacheManager](#)
- [RemoteCache](#)

20.7. REST Server

The {brandname} Server distribution contains a module that implements [RESTful](#) HTTP access to the {brandname} data grid, built on [Netty](#).

20.7.1. Running the REST server

The REST server endpoint is part of the {brandname} Server and by default listens on port 8080. To run the server locally, [download](#) the zip distribution and execute in the extracted directory:

```
bin/standalone.sh -b 0.0.0.0
```

or alternatively, run via docker:

```
docker run -it -p 8080:8080 -e "APP_USER=user" -e "APP_PASS=changeme"
jboss/infinispan-server
```

Security

The REST server is protected by authentication, so before usage it is necessary to create an application login. When running via docker, this is achieved by the APP_USER and APP_PASS command line arguments, but when running locally, this can be done with:

```
bin/add-user.sh -u user -p changeme -a
```

20.7.2. Supported protocols

The REST Server supports HTTP/1.1 as well as HTTP/2 protocols. It is possible to switch to HTTP/2 by either performing a [HTTP/1.1 Upgrade procedure](#) or by negotiating communication protocol using [TLS/ALPN extension](#).

Note: TLS/ALPN with JDK8 requires additional steps from the client perspective. Please refer to your client documentation but it is very likely that you will need Jetty ALPN Agent or OpenSSL bindings.

20.7.3. REST API

HTTP PUT and POST methods are used to place data in the cache, with URLs to address the cache name and key(s) - the data being the body of the request (the data can be anything you like). Other headers are used to control the cache settings and behaviour.

Data formats

Configuration

Each cache exposed via REST stores data in a configurable data format defined by a [MediaType](#). More details in the configuration [here](#).

An example of storage configuration is as follows:

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

When no MediaType is configured, {brandname} assumes "application/octet-stream" for both keys and values, with the following exceptions:

- If the cache is indexed, it assumes "application/x-protostream"
- If the cache is configured with compatibility mode, it assumes "application/x-java-object"

Supported formats

Data can be written and read in different formats than the storage format; {brandname} can convert between those formats when required.

The following "standard" formats can be converted interchangeably:

- *application/x-java-object*
- *application/octet-stream*
- *application/x-www-form-urlencoded*
- *text/plain*

The following formats can be converted to/from the formats above:

- *application/xml*
- *application/json*
- *application/x-jboss-marshalling*
- *application/x-protostream*
- *application/x-java-serialized*

Finally, the following conversion is also supported:

- Between *application/x-protostream* and *application/json*

All the REST API calls can provide headers describing the content written or the required format of the content when reading. {brandname} supports the standard HTTP/1.1 headers "Content-Type" and "Accept" that are applied for values, plus the "Key-Content-Type" with similar effect for keys.

Accept header

The REST server is compliant with the [RFC-2616](#) Accept header, and will negotiate the correct MediaType based on the conversions supported. Example, sending the following header when reading data:

```
Accept: text/plain;q=0.7, application/json;q=0.8, */*;q=0.6
```

will cause {brandname} to try first to return content in JSON format (higher priority 0.8). If it's not possible to convert the storage format to JSON, next format tried will be *text/plain* (second highest priority 0.7), and finally it falls back to **/**, that will pick a format suitable for displaying automatically based on the cache configuration.

Key-Content-Type header

Most REST API calls have the Key included in the URL. {brandname} will assume the Key is a *java.lang.String* when handling those calls, but it's possible to use a specific header *Key-Content-Type* for keys in different formats.

Examples:

- Specifying a byte[] Key as a Base64 string:

API call:

```
`PUT /my-cache/AQIDBDM=`
```

Headers:

Key-Content-Type: application/octet-stream

- Specifying a byte[] Key as a hexadecimal string:

API call:

GET /my-cache/0x01CA03042F

Headers:

```
Key-Content-Type: application/octet-stream; encoding=hex
```

- Specifying a double Key:

API call:

POST /my-cache/3.141456

Headers:

```
Key-Content-Type: application/x-java-object;type=java.lang.Double
```

The *type* parameter for *application/x-java-object* is restricted to:

- Primitive wrapper types
- java.lang.String
- Bytes, making *application/x-java-object;type=Bytes* equivalent to *application/octet-stream;encoding=hex*

Putting data in

PUT /{cacheName}/{cacheKey}

A PUT request of the above URL form will place the payload (body) in the given cache, with the given key (the named cache must exist on the server). For example <http://someserver/hr/payRoll-3> (in which case *hr* is the cache name, and *payRoll-3* is the key). Any existing data will be replaced, and Time-To-Live and Last-Modified values etc will updated (if applicable).

POST /{cacheName}/{cacheKey}

Exactly the same as PUT, only if a value in a cache/key already exists, it will return a Http CONFLICT status (and the content will not be updated).

Headers

- **Key-Content-Type**: OPTIONAL The content type for the Key present in the URL.
- **Content-Type** : OPTIONAL The **MediaType** of the Value being sent.
- **performAsync** : OPTIONAL true/false (if true, this will return immediately, and then replicate data to the cluster on its own. Can help with bulk data inserts/large clusters.)
- **timeToLiveSeconds** : OPTIONAL number (the number of seconds before this entry will automatically be deleted). If no parameter is sent, {brandname} assumes configuration default value. Passing any negative value will create an entry which will live forever.
- **maxIdleTimeSeconds** : OPTIONAL number (the number of seconds after last usage of this entry when it will automatically be deleted). If no parameter is sent, {brandname} configuration default value. Passing any negative value will create an entry which will live forever.

Passing 0 as parameter for timeToLiveSeconds and/or maxIdleTimeSeconds

- If both **timeToLiveSeconds** and **maxIdleTimeSeconds** are 0, the cache will use the default **lifespan** and **maxIdle** values configured in XML/programmatically
- If only **maxIdleTimeSeconds** is 0, it uses the **timeToLiveSeconds** value passed as parameter (or -1 if not present), and default **maxIdle** configured in XML/programmatically
- If only **timeToLiveSeconds** is 0, it uses default **lifespan** configured in XML/programmatically, and **maxIdle** is set to whatever came as parameter (or -1 if not present)

JSON/Protostream conversion

When caches are indexed, or specifically configured to store *application/x-protostream*, it's possible to send and receive JSON documents that are automatically converted to/from protostream. In order for the conversion to work, a protobuf schema must be registered.

The registration can be done via REST, by doing a POST/PUT in the `__protobuf_metadata` cache. Example using cURL:

```
curl -u user:password -X POST --data-binary @./schema.proto
http://127.0.0.1:8080/rest/___protobuf_metadata/schema.proto
```

When writing a JSON document, a special field **_type** must be present in the document to identify the protobuf *Message* corresponding to the document.

For example, consider the following schema:

```
message Person {
  required string name = 1;
  required int32 age = 2;
}
```

A conformant JSON document would be:

```
{
  "_type": "Person",
  "name": "user1",
  "age": 32
}
```

Getting data back out

HTTP GET and HEAD are used to retrieve data from entries.

GET */cacheName/cacheKey*

This will return the data found in the given cacheName, under the given key - as the body of the response. A Content-Type header will be present in the response according to the Media Type negotiation. Browsers can use the cache directly of course (eg as a CDN). An **ETag** will be returned unique for each entry, as will the Last-Modified and Expires headers field indicating the state of the data at the given URL. ETags allow browsers (and other clients) to ask for data only in the case where it has changed (to save on bandwidth) - this is standard HTTP and is honoured by {brandname}.

Headers

- **Key-Content-Type**: OPTIONAL The content type for the Key present in the URL. When omitted, *application/x-java-object; type=java.lang.String* is assumed
- **Accept**: OPTIONAL The required format to return the content

It is possible to obtain additional information by appending the "extended" parameter on the query string, as follows:

GET */cacheName/cacheKey?extended*

This will return the following custom headers:

- **Cluster-Primary-Owner**: the node name of the primary owner for this key
- **Cluster-Node-Name**: the JGroups node name of the server that has handled the request
- **Cluster-Physical-Address**: the physical JGroups address of the server that has handled the request.

HEAD */cacheName/cacheKey*

The same as GET, only no content is returned (only the header fields). You will receive the same content that you stored. E.g., if you stored a String, this is what you get back. If you stored some XML or JSON, this is what you will receive. If you stored a binary (base 64 encoded) blob, perhaps a serialized; Java; object - you will need to; deserialize this yourself.

Similarly to the GET method, the HEAD method also supports returning extended information via headers. See above.

Headers

- **Key-Content-Type**: OPTIONAL The content type for the Key present in the URL. When omitted, *application/x-java-object; type=java.lang.String* is assumed

Listing keys

GET /{cacheName}

This will return a list of keys present in the given cacheName as the body of the response. The format of the response can be controlled via the Accept header as follows:

- *application/xml* - the list of keys will be returned in XML format.
- *application/json* - the list of keys will be return in JSON format.
- *text/plain* - the list of keys will be returned in plain text format, one key per line

If the cache identified by cacheName is distributed, only the keys owned by the node handling the request will be returned. To return all keys, append the "global" parameter to the query, as follows:

GET /cacheName?global

Removing data

Data can be removed at the cache key/element level, or via a whole cache name using the HTTP delete method.

DELETE /{cacheName}/{cacheKey}

Removes the given key name from the cache.

Headers

- **Key-Content-Type**: OPTIONAL The content type for the Key present in the URL. When omitted, *application/x-java-object; type=java.lang.String* is assumed

DELETE /{cacheName}

Removes ALL the entries in the given cache name (i.e., everything from that path down). If the operation is successful, it returns 200 code.



Make it quicker!

Set the header `performAsync` to true to return immediately and let the removal happen in the background.

Querying

The REST server supports Ickle Queries in JSON format. It's important that the cache is configured with *application/x-protostream* for both Keys and Values. If the cache is indexed, no configuration is needed.

GET /{cacheName}?action=search&query={ickle query}

Will execute an Ickle query in the given cache name.

Request parameters

- *query*: REQUIRED the query string
- *max_results*: OPTIONAL the number of results to return, default is 10
- *offset*: OPTIONAL the index of the first result to return, default is 0
- *query_mode*: OPTIONAL the [execution mode](#) of the query once it's received by server. Valid values are *FETCH* and *BROADCAST*. Default is *FETCH*.

Query Result

Results are JSON documents containing one or more hits. Example:

```
{
  "total_results" : 150,
  "hits" : [ {
    "hit" : {
      "name" : "user1",
      "age" : 35
    }
  }, {
    "hit" : {
      "name" : "user2",
      "age" : 42
    }
  }, {
    "hit" : {
      "name" : "user3",
      "age" : 12
    }
  } ]
}
```

- *total_results*: NUMBER, the total number of results from the query.
- *hits*: ARRAY, list of matches from the query
- *hit*: OBJECT, each result from the query. Can contain all fields or just a subset of fields in case a *Select* clause is used.

POST /{cacheName}?action=search

Similar to que query using GET, but the body of the request is used instead to specify the query parameters.

Example:

```
{
  "query":"from Entity where name:\"user1\"",
  "max_results":20,
  "offset":10
}
```

20.7.4. CORS

The REST server supports [CORS](#) including preflight and rules based on the request origin.

Example:

```
<rest-connector name="rest1" socket-binding="rest" cache-container="default">
  <cors-rules>
    <cors-rule name="restrict host1" allow-credentials="false">
      <allowed-origins>http://host1,https://host1</allowed-origins>
      <allowed-methods>GET</allowed-methods>
    </cors-rule>
    <cors-rule name="allow ALL" allow-credentials="true" max-age-seconds="2000">
      <allowed-origins>*</allowed-origins>
      <allowed-methods>GET,OPTIONS,POST,PUT,DELETE</allowed-methods>
      <allowed-headers>Key-Content-Type</allowed-headers>
    </cors-rule>
  </cors-rules>
</rest-connector>
```

The rules are evaluated sequentially based on the "Origin" header set by the browser; in the example above if the origin is either "http://host1" or "https://host1" the rule "restrict host1" will apply, otherwise the next rule will be tested. Since the rule "allow ALL" permits all origins, any script coming from a different origin will be able to perform the methods specified and use the headers supplied.

The `<cors-rule>` element can be configured as follows:

Config	Description	Mandatory
name	The name of the rule	yes
allow-credentials	Enable CORS requests to use credentials	no
allowed-origins	A comma separated list used to set the CORS 'Access-Control-Allow-Origin' header to indicate the response can be shared with the origins	yes

Config	Description	Mandatory
allowed-methods	A comma separated list used to set the CORS 'Access-Control-Allow-Methods' header in the preflight response to specify the methods allowed for the configured origin(s)	yes
max-age-seconds	The amount of time CORS preflight request headers can be cached	no
expose-headers	A comma separated list used to set the CORS 'Access-Control-Expose-Headers' in the preflight response to specify which headers can be exposed to the configured origin(s)	no

20.7.5. Client side code

Part of the point of a RESTful service is that you don't need to have tightly coupled client libraries/bindings. All you need is a HTTP client library. For Java, Apache HTTP Commons Client works just fine (and is used in the integration tests), or you can use java.net API.

Ruby example

```
# Shows how to interact with the REST api from ruby.
# No special libraries, just standard net/http
#
# Author: Michael Neale
#
require 'net/http'

uri = URI.parse('http://localhost:8080/rest/default/MyKey')
http = Net::HTTP.new(uri.host, uri.port)

#Create new entry

post = Net::HTTP::Post.new(uri.path, {"Content-Type" => "text/plain"})
post.basic_auth('user', 'pass')
post.body = "DATA HERE"

resp = http.request(post)

puts "POST response code : " + resp.code

#get it back

get = Net::HTTP::Get.new(uri.path)
get.basic_auth('user', 'pass')
```

```

resp = http.request(get)

puts "GET response code: " + resp.code
puts "GET Body: " + resp.body

#use PUT to overwrite

put = Net::HTTP::Put.new(uri.path, {"Content-Type" => "text/plain"})
put.basic_auth('user', 'pass')
put.body = "ANOTHER DATA HERE"

resp = http.request(put)

puts "PUT response code : " + resp.code

#and remove...
delete = Net::HTTP::Delete.new(uri.path)
delete.basic_auth('user', 'pass')

resp = http.request(delete)

puts "DELETE response code : " + resp.code

#Create binary data like this... just the same...

uri = URI.parse('http://localhost:8080/rest/default/MyLogo')
put = Net::HTTP::Put.new(uri.path, {"Content-Type" => "application/octet-stream"})
put.basic_auth('user', 'pass')
put.body = File.read('./logo.png')

resp = http.request(put)

puts "PUT response code : " + resp.code

#and if you want to do json...
require 'rubygems'
require 'json'

#now for fun, lets do some JSON !
uri = URI.parse('http://localhost:8080/rest/jsonCache/user')
put = Net::HTTP::Put.new(uri.path, {"Content-Type" => "application/json"})
put.basic_auth('user', 'pass')

data = {:name => "michael", :age => 42 }
put.body = data.to_json

resp = http.request(put)

puts "PUT response code : " + resp.code

get = Net::HTTP::Get.new(uri.path)

```

```
get.basic_auth('user', 'pass')
resp = http.request(get)

puts "GET Body: " + resp.body
```

Python 3 example

```
import urllib.request

# Setup basic auth
base_uri = 'http://localhost:8080/rest/default'
auth_handler = urllib.request.HTTPBasicAuthHandler()
auth_handler.add_password(user='user', passwd='pass', realm='ApplicationRealm', uri=base_uri)
opener = urllib.request.build_opener(auth_handler)
urllib.request.install_opener(opener)

# putting data in
data = "SOME DATA HERE \!"

req = urllib.request.Request(url=base_uri + '/Key', data=data.encode("UTF-8"), method='PUT',
                             headers={"Content-Type": "text/plain"})
with urllib.request.urlopen(req) as f:
    pass

print(f.status)
print(f.reason)

# getting data out
resp = urllib.request.urlopen(base_uri + '/Key')
print(resp.read().decode('utf-8'))
```

Java example

```
package org.infinispan;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Base64;

/**
 * Rest example accessing a cache.
 */
```



```

* @author Samuel Tauil (samuel@redhat.com)
*/
public class RestExample {

    /**
     * Method that puts a String value in cache.
     *
     * @param urlServerAddress URL containing the cache and the key to insert
     * @param value            Text to insert
     * @param user             Used for basic auth
     * @param password         Used for basic auth
     */
    public void putMethod(String urlServerAddress, String value, String user, String
password) throws IOException {
        System.out.println("-----");
        System.out.println("Executing PUT");
        System.out.println("-----");
        URL address = new URL(urlServerAddress);
        System.out.println("executing request " + urlServerAddress);
        HttpURLConnection connection = (HttpURLConnection) address.openConnection();
        System.out.println("Executing put method of value: " + value);
        connection.setRequestMethod("PUT");
        connection.setRequestProperty("Content-Type", "text/plain");
        addAuthorization(connection, user, password);
        connection.setDoOutput(true);

        OutputStreamWriter outputStreamWriter = new OutputStreamWriter(connection
.getOutputStream());
        outputStreamWriter.write(value);

        connection.connect();
        outputStreamWriter.flush();
        System.out.println("-----");
        System.out.println(connection.getResponseCode() + " " + connection
.getResponseMessage());
        System.out.println("-----");
        connection.disconnect();
    }

    /**
     * Method that gets a value by a key in url as param value.
     *
     * @param urlServerAddress URL containing the cache and the key to read
     * @param user             Used for basic auth
     * @param password         Used for basic auth
     * @return String value
     */
    public String getMethod(String urlServerAddress, String user, String password)
throws IOException {
        String line;
        StringBuilder stringBuilder = new StringBuilder();

```

```

System.out.println("-----");
System.out.println("Executing GET");
System.out.println("-----");

URL address = new URL(urlServerAddress);
System.out.println("executing request " + urlServerAddress);

URLConnection connection = (URLConnection) address.openConnection();
connection.setRequestMethod("GET");
connection.setRequestProperty("Content-Type", "text/plain");
addAuthorization(connection, user, password);
connection.setDoOutput(true);

BufferedReader bufferedReader = new BufferedReader(new InputStreamReader
(connection.getInputStream()));

connection.connect();

while ((line = bufferedReader.readLine()) != null) {
    stringBuilder.append(line).append('\n');
}

System.out.println("Executing get method of value: " + stringBuilder.toString
());

System.out.println("-----");
System.out.println(connection.getResponseCode() + " " + connection
.getResponseMessage());
System.out.println("-----");

connection.disconnect();

return stringBuilder.toString();
}

private void addAuthorization(URLConnection connection, String user, String
pass) {
    String credentials = user + ":" + pass;
    String basic = Base64.getEncoder().encodeToString(credentials.getBytes());
    connection.setRequestProperty("Authorization", "Basic " + basic);
}

/**
 * Main method example.
 */
public static void main(String[] args) throws IOException {
    RestExample restExample = new RestExample();
    String user = "user";
    String pass = "pass";
    restExample.putMethod("http://localhost:8080/rest/default/1", "Infinispan REST

```

```
Test", user, pass);
    restExample.getMethod("http://localhost:8080/rest/default/1", user, pass);
}
}
```

20.8. Memcached Server

The {brandname} Server distribution contains a server module that implements the [Memcached text protocol](#). This allows Memcached clients to talk to one or several {brandname} backed Memcached servers. These servers could either be working standalone just like Memcached does where each server acts independently and does not communicate with the rest, or they could be clustered where servers replicate or distribute their contents to other {brandname} backed Memcached servers, thus providing clients with failover capabilities. Please refer to {brandname} Server's [memcached server guide](#) for instructions on how to configure and run a Memcached server.

20.8.1. Client Encoding

The memcached text protocol assumes data values read and written by clients are raw bytes. The support for type negotiation will come with [the memcached binary protocol](#) implementation, as part of [ISPN-8726](#).

Although it's not possible for a memcached client to negotiate the data type to obtain data from the server or send data in different formats, the server can optionally be configured to handle values encoded with a certain Media Type. By setting the `client-encoding` attribute in the `memcached-connector` element, the server will return content in this configured format, and clients also send data in this format.

The `client-encoding` is useful when a single cache is accessed from multiple remote endpoints (Rest, HotRod, Memcached) and it allows to tailor the responses/requests to memcached text clients. For more information on interoperability between endpoints, consult [Endpoint Interop guide](#).

20.8.2. Command Clarifications

Flush All

Even in a clustered environment, `flush_all` command leads to the clearing of the {brandname} Memcached server where the call lands. There's no attempt to propagate this flush to other nodes in the cluster. This is done so that `flush_all` with delay use case can be reproduced with the {brandname} Memcached server. The aim of passing a delay to `flush_all` is so that different Memcached servers in a full can be flushed at different times, and hence avoid overloading the database with requests as a result of all Memcached servers being empty. For more info, check the [Memcached text protocol section on flush_all](#).

20.8.3. Unsupported Features

This section explains those parts of the memcached text protocol that for one reason or the other, are not currently supported by the {brandname} based memcached implementation.

Individual Stats

Due to difference in nature between the original memcached implementation which is C/C\ based and the {brandname} implementation which is Java based, there're some general purpose stats that are not supported. For these unsupported stats, {brandname} memcached server always returns 0.

Unsupported statistics

- pid
- pointer_size
- rusage_user
- rusage_system
- bytes
- curr_connections
- total_connections
- connection_structures
- auth_cmds
- auth_errors
- limit_maxbytes
- threads
- conn_yields
- reclaimed

Statistic Settings

The settings statistics section of the text protocol has not been implemented due to its volatility.

Settings with Arguments Parameter

Since the arguments that can be send to the Memcached server are not documented, {brandname} Memcached server does not support passing any arguments to stats command. If any parameters are passed, the {brandname} Memcached server will respond with a CLIENT_ERROR .

Delete Hold Time Parameter

Memcached does no longer honor the optional hold time parameter to delete command and so the {brandname} based memcached server does not implement such feature either.

Verbosity Command

Verbosity command is not supported since {brandname} logging cannot be simplified to defining the logging level alone.

20.8.4. Talking To {brandname} Memcached Servers From Non-Java Clients

This section shows how to talk to {brandname} memcached server via non-java client, such as a python script.

Multi Clustered Server Tutorial

The example showcases the distribution capabilities of {brandname} memcached servers that are not available in the original memcached implementation.

- Start two clustered nodes: This configuration is the same one used for the GUI demo:

```
$ ./bin/standalone.sh -c clustered.xml -Djboss.node.name=nodeA
$ ./bin/standalone.sh -c clustered.xml -Djboss.node.name=nodeB
-Djboss.socket.binding.port-offset=100
```

Alternatively use

```
$ ./bin/domain.sh
```

Which automatically starts two nodes.

- Execute [test_memcached_write.py](#) script which basically executes several write operations against the {brandname} memcached server bound to port 11211. If the script is executed successfully, you should see an output similar to this:

```
Connecting to 127.0.0.1:11211
Testing set ['Simple_Key': Simple value] ... OK
Testing set ['Expiring_Key' : 999 : 3] ... OK
Testing increment 3 times ['Incr_Key' : starting at 1 ]
Initialise at 1 ... OK
Increment by one ... OK
Increment again ... OK
Increment yet again ... OK
Testing decrement 1 time ['Decr_Key' : starting at 4 ]
Initialise at 4 ... OK
Decrement by one ... OK
Testing decrement 2 times in one call ['Multi-Decr_Key' : 3 ]
Initialise at 3 ... OK
Decrement by 2 ... OK
```

- Execute [test_memcached_read.py](#) script which connects to server bound to 127.0.0.1:11311 and verifies that it can read the data that was written by the writer script to the first server. If the script is executed successfully, you should see an output similar to this:

```
Connecting to 127.0.0.1:11311
Testing get ['Simple_Key'] should return Simple value ... OK
Testing get ['Expiring_Key'] should return nothing... OK
Testing get ['Incr_Key'] should return 4 ... OK
Testing get ['Decr_Key'] should return 3 ... OK
Testing get ['Multi-Decr_Key'] should return 1 ... OK
```

20.9. Executing code in the Remote Grid

In an earlier section we described executing code in the grid. Unfortunately these methods are designed to be used in an embedded scenario with direct access to the grid. This section will detail how you can perform similar functions but while using a remote client connected to the grid.

20.10. Scripting

Scripting is a feature of {brandname} Server which allows invoking server-side scripts from remote clients. Scripting leverages the JDK's `javax.script` `ScriptEngines`, therefore allowing the use of any JVM languages which offer one. By default, the JDK comes with Nashorn, a `ScriptEngine` capable of running JavaScript.

20.10.1. Installing scripts

Scripts are stored in a special script cache, named `'__script_cache'`. Adding a script is therefore as simple as put+ting it into the cache itself. If the name of the script contains a filename extension, e.g. `+myscript.js`, then that extension determines the engine that will be used to execute it. Alternatively the script engine can be selected using script metadata (see below). Be aware that, when security is enabled, access to the script cache via the remote protocols requires that the user belongs to the `'__script_manager'` role.

20.10.2. Script metadata

Script metadata is additional information about the script that the user can provide to the server to affect how a script is executed. It is contained in a specially-formatted comment on the first lines of the script.

Properties are specified as key=value pairs, separated by commas. You can use several different comment styles: The `//`, `;;`, `#` depending on the scripting language you use. You can split metadata over multiple lines if necessary, and you can use single (') or double (") quotes to delimit your values.

The following are examples of valid metadata comments:

```
// name=test, language=javascript
// mode=local, parameters=[a,b,c]
```

Metadata properties

The following metadata property keys are available

- **mode**: defines the mode of execution of a script. Can be one of the following values:
 - **local**: the script will be executed only by the node handling the request. The script itself however can invoke clustered operations
 - **distributed**: runs the script using the Distributed Executor Service
- **language**: defines the script engine that will be used to execute the script, e.g. Javascript
- **extension**: an alternative method of specifying the script engine that will be used to execute the script, e.g. js
- **role**: a specific role which is required to execute the script
- **parameters**: an array of valid parameter names for this script. Invocations which specify parameter names not included in this list will cause an exception.
- **datatype**: optional property providing information, in the form of Media Types (also known as MIME) about the type of the data stored in the caches, as well as parameter and return values. Currently it only accepts a single value which is `text/plain; charset=utf-8`, indicating that data is String UTF-8 format. This metadata parameter is designed for remote clients that only support a particular type of data, making it easy for them to retrieve, store and work with parameters.

Since the execution mode is a characteristic of the script, nothing special needs to be done on the client to invoke scripts in different modes.

20.10.3. Script bindings

The script engine within {brandname} exposes several internal objects as bindings in the scope of the script execution. These are:

- **cache**: the cache against which the script is being executed
- **marshaller**: the marshaller to use for marshalling/unmarshalling data to the cache
- **cacheManager**: the cacheManager for the cache
- **scriptingManager**: the instance of the script manager which is being used to run the script. This can be used to run other scripts from a script.

20.10.4. Script parameters

Aside from the standard bindings described above, when a script is executed it can be passed a set of named parameters which also appear as bindings. Parameters are passed as name,value pairs where name is a string and value can be any value that is understood by the marshaller in use.

The following is an example of a JavaScript script which takes two parameters, multiplicand and multiplier and multiplies them. Because the last operation is an expression evaluation, its result is returned to the invoker.

```
// mode=local,language=javascript
multiplicand * multiplier
```

To store the script in the script cache, use the following Hot Rod code:

```
RemoteCache<String, String> scriptCache = cacheManager.getCache("___script_cache");
scriptCache.put("multiplication.js",
    "// mode=local,language=javascript\n" +
    "multiplicand * multiplier\n");
```

20.10.5. Running Scripts using the Hot Rod Java client

The following example shows how to invoke the above script by passing two named parameters.

```
RemoteCache<String, Integer> cache = cacheManager.getCache();
// Create the parameters for script execution
Map<String, Object> params = new HashMap<>();
params.put("multiplicand", 10);
params.put("multiplier", 20);
// Run the script on the server, passing in the parameters
Object result = cache.execute("multiplication.js", params);
```

20.10.6. Distributed execution

The following is a script which runs on all nodes. Each node will return its address, and the results from all nodes will be collected in a List and returned to the client.

```
// mode:distributed,language=javascript
cacheManager.getAddress().toString();
```

20.11. Server Tasks

Server tasks are server-side scripts defined in Java language. To develop a server task, you should define a class that extends `org.infinispan.tasks.ServerTask` interface, defined in `infinispan-tasks-api` module.

A typical server task implementation would implement these methods:

- `setTaskContext` allows server tasks implementors to access execution context information. This includes task parameters, cache reference on which the task is executed...etc. Normally, implementors would store this information locally and use it when the task is actually executed.
- `getName` should return a unique name for the task. The client will use this name to invoke the task.
- `getExecutionMode` is used to decide whether to invoke the task in 1 node in a cluster of N nodes

or invoke it in N nodes. For example, server tasks that invoke stream processing are only required to be executed in 1 node in the cluster. This is because stream processing itself makes sure processing is distributed to all nodes in cluster.

- `call` is the method that's invoked when the user invokes the server task.

Here's an example of a hello greet task that takes as parameter the name of the person to greet.

```
package example;

import org.infinispan.tasks.ServerTask;
import org.infinispan.tasks.TaskContext;

public class HelloTask implements ServerTask<String> {

    private TaskContext ctx;

    @Override
    public void setTaskContext(TaskContext ctx) {
        this.ctx = ctx;
    }

    @Override
    public String call() throws Exception {
        String name = (String) ctx.getParameters().get().get("name");
        return "Hello " + name;
    }

    @Override
    public String getName() {
        return "hello-task";
    }

}
```

Once the task has been implemented, it needs to be wrapped inside a jar. The jar is then deployed to the {brandname} Server and from then on it can be invoked. The {brandname} Server uses [service loader pattern](#) to load the task, so implementations need to adhere to these requirements. For example, server task implementations must have a zero-argument constructor.

Moreover, the jar must contain a `META-INF/services/org.infinispan.tasks.ServerTask` file containing the fully qualified name(s) of the server tasks included in the jar. For example:

```
example.HelloTask
```

With jar packaged, the next step is to push the jar to the {brandname} Server. The server is powered by WildFly Application Server, so if using Maven [Wildfly's Maven plugin](#) can be used for this:

```
<plugin>
  <groupId>org.wildfly.plugins</groupId>
  <artifactId>wildfly-maven-plugin</artifactId>
  <version>1.2.0.Final</version>
</plugin>
```

Then call the following from command line:

```
$ mvn package wildfly:deploy
```

Alternative ways of deployment jar files to Wildfly Application Server are explained [here](#).

Executing the task can be done using the following code:

```
// Create a configuration for a locally-running server
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.addServer().host("127.0.0.1").port(11222);

// Connect to the server
RemoteCacheManager cacheManager = new RemoteCacheManager(builder.build());

// Obtain the remote cache
RemoteCache<String, String> cache = cacheManager.getCache();

// Create task parameters
Map<String, String> parameters = new HashMap<>();
parameters.put("name", "developer");

// Execute task
String greet = cache.execute("hello-task", parameters);
System.out.println(greet);
```

Chapter 21. Compatibility Mode



Compatibility mode is deprecated and will be removed from {brandname}. To achieve interoperability between remote endpoints, you should use protocol interoperability capabilities. See [Protocol Interoperability](#).

Compatibility mode configures {brandname} caches so that you can access {brandname} in multiple ways. Achieving such compatibility requires extra work from {brandname} in order to make sure that contents are converted back and forth between the different formats of each endpoint and this is the reason why compatibility mode is disabled by default.

21.1. Enable Compatibility Mode

For compatibility mode to work as expected, all endpoints need to be configured with the same cache manager, and need to talk to the same cache. If you're using the brand new [{brandname} Server distribution](#), this is all done for you. If you're in the mood to experiment with this in a standalone unit test, [this class](#) shows you how you can start multiple endpoints from a single class.

So, to get started using {brandname}'s compatibility mode, it needs to be enabled, either via XML:

infinispan.xml

```
<local-cache>
  <compatibility/>
</local-cache>
```

Or programmatically:

```
ConfigurationBuilder builder = ...
builder.compatibility().enable();
```

The key thing to remember about {brandname}'s compatibility mode is that where possible, it tries to store data unmarshalling or deserializing it. It does so because the most common use case is for it to store Java objects and having Java objects stored in deserialized form means that they're very easy to use from an embedded cache. With this in mind, it makes some assumptions. For example, if something is stored via Hot Rod, it's most likely coming from the reference Hot Rod client, which is written in Java, and which uses a marshaller that keeps binary payloads very compact. So, when the Hot Rod operation reaches the compatibility layer, it will try to unmarshall it, by default using the same default marshaller used by the Java Hot Rod client, hence providing good out-of-the-box support for the majority of cases.

21.1.1. Optional: Configuring Compatibility Marshaller

It could happen though the client might be using a Hot Rod client written for another language other than Java, say [Ruby](#) or [Python](#). In this case, some kind of custom marshaller needs to be configured that either translates that serialized payload into a Java object to be stored in the cache,

or keeps it in serialized form. Both options are valid, but of course it will have an impact on what kind of objects are retrieved from {brandname} if using the embedded cache. The marshaller is expected to implement [this interface](#). Configuring the compatibility marshaller is optional and can be done via XML:

infinispan.xml

```
<local-cache>
  <compatibility marshaller="com.acme.CustomMarshaller"/>
</local-cache>
```

Or programmatically:

```
ConfigurationBuilder builder = ...
builder.compatibility().enable().marshaller(new com.acme.CustomMarshaller());
```

One concrete example of this marshaller logic can be found in the [SpyMemcachedCompatibleMarshaller](#). [Spy Memcached](#) uses their own transcoders in order to marshall objects, so the compatibility marshaller created is in charge of marshalling/unmarshalling data stored via Spy Memcached client. If you want to retrieve data stored via Spy Memcached via say Hot Rod, you can configure the Java Hot Rod client to use this same marshaller, and this is precisely what the test where the Spy Memcached marshaller is located is demonstrating.

21.2. Code examples

The best code examples available showing compatibility in action can be found in the [{brandname} Compatibility Mode testsuite](#), but more will be developed in the near future.

Chapter 22. Protocol Interoperability

Clients exchange data with {brandname} through endpoints such as REST or Hot Rod.

Each endpoint uses a different protocol so that clients can read and write data in a suitable format. Because {brandname} can interoperate with multiple clients at the same time, it must convert data between client formats and the storage formats.

To configure {brandname} endpoint interoperability, you should define the [MediaType](#) that sets the format for data stored in the cache.

22.1. Considerations with Media Types and Endpoint Interoperability

Configuring {brandname} to store data with a specific media type affects client interoperability.

Although REST clients do support sending and receiving [encoded binary data](#), they are better at handling text formats such as JSON, XML, or plain text.

Memcached text clients can handle String-based keys and byte[] values but cannot negotiate data types with the server. These clients do not offer much flexibility when handling data formats because of the protocol definition.

Java Hot Rod clients are suitable for handling Java objects that represent entities that reside in the cache. Java Hot Rod clients use marshalling operations to serialize and deserialize those objects into byte arrays.

Similarly, non-Java Hot Rod clients, such as the C++, C#, and Javascript clients, are suitable for handling objects in the respective languages. However, non-Java Hot Rod clients can interoperate with Java Hot Rod clients using platform independent data formats.

22.2. REST, Hot Rod, and Memcached Interoperability with Text-Based Storage

You can configure key and values with a text-based storage format.

For example, specify [text/plain; charset=UTF-8](#), or any other character set, to set plain text as the media type. You can also specify a media type for other text-based formats such as JSON ([application/json](#)) or XML ([application/xml](#)) with an optional character set.

The following example configures the cache to store entries with the [text/plain; charset=UTF-8](#) media type:

```
<cache>
  <encoding>
    <key media-type="text/plain; charset=UTF-8"/>
    <value media-type="text/plain; charset=UTF-8"/>
  </encoding>
</cache>
```

To handle the exchange of data in a text-based format, you must configure Hot Rod clients with the `org.infinispan.commons.marshall.StringMarshaller` marshaller.

REST clients must also send the correct headers when writing and reading from the cache, as follows:

- Write: `Content-Type: text/plain; charset=UTF-8`
- Read: `Accept: text/plain; charset=UTF-8`

Memcached clients do not require any configuration to handle text-based formats.

This configuration is compatible with...	
REST clients	Yes
Java Hot Rod clients	Yes
Memcached clients	Yes
Non-Java Hot Rod clients	No
Querying and Indexing	No
Custom Java objects	No

22.3. REST, Hot Rod, and Memcached Interoperability with Custom Java Objects

If you store entries in the cache as marshalled, custom Java objects, you should configure the cache with the `MediaType` of the marshalled storage.

Java Hot Rod clients use the JBoss marshalling storage format as the default to store entries in the cache as custom Java objects.

The following example configures the cache to store entries with the `application/x-jboss-marshalling` media type:

```
<distributed-cache name="my-cache">
  <encoding>
    <key media-type="application/x-jboss-marshalling"/>
    <value media-type="application/x-jboss-marshalling"/>
  </encoding>
</distributed-cache>
```

If you use the Protostream marshaller, configure the MediaType as `application/x-protostream`. For UTF8Marshaller, configure the MediaType as `text/plain`.



If only Hot Rod clients interact with the cache, you do not need to configure the MediaType.

Because REST clients are most suitable for handling text formats, you should use primitives such as `java.lang.String` for keys. Otherwise, REST clients must handle keys as `bytes[]` using a [supported binary encoding](#).

REST clients can read values for cache entries in XML or JSON format. However, the classes must be available in the server.

To read and write data from Memcached clients, you must use `java.lang.String` for keys. Values are stored and returned as marshalled objects.

Some Java Memcached clients allow data transformers that marshall and unmarshall objects. You can also configure the Memcached server module to encode responses in different formats, such as 'JSON' which is language neutral. This allows non-Java clients to interact with the data even if the storage format for the cache is Java-specific. See [Client Encoding](#) details for the {brandname} Memcached server module.



Storing Java objects in the cache requires you to deploy entity classes to {ProductName}. See [Deploying Entity Classes](#).

This configuration is compatible with...

REST clients	Yes
Java Hot Rod clients	Yes
Memcached clients	Yes
Non-Java Hot Rod clients	No
Querying and Indexing	No
Custom Java objects	Yes

22.4. Java and Non-Java Client Interoperability with Protobuf

Storing data in the cache as Protobuf encoded entries provides a platform independent configuration that enables Java and Non-Java clients to access and query the cache from any endpoint.

If indexing is configured for the cache, {brandname} automatically stores keys and values with the `application/x-protostream` media type.

If indexing is not configured for the cache, you can configure it to store entries with the `application/x-protostream` media type as follows:

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

{brandname} converts between `application/x-protostream` and `application/json`, which allows REST clients to read and write JSON formatted data. However REST clients must send the correct headers, as follows:

Read Header

Read: Accept: application/json

Write Header

Write: Content-Type: application/json



The `application/x-protostream` media type uses Protobuf encoding, which requires you to register a Protocol Buffers schema definition that describes the entities and marshallers that the clients use. See [Storing Protobuf Entities](#).

This configuration is compatible with...

REST clients	Yes
Java Hot Rod clients	Yes
Non-Java Hot Rod clients	Yes
Querying and Indexing	Yes
Custom Java objects	Yes

22.5. Custom Code Interoperability

You can deploy custom code with {brandname}. For example, you can deploy scripts, tasks, listeners, converters, and merge policies. Because your custom code can access data directly in the cache, it must interoperate with clients that access data in the cache through different endpoints.

For example, you might create a remote task to handle custom objects stored in the cache while other clients store data in binary format.

To handle interoperability with custom code you can either convert data on demand or store data as Plain Old Java Objects (POJOs).

22.5.1. Converting Data On Demand

If the cache is configured to store data in a binary format such as `application/x-protostream` or

`application/x-jboss-marshalling`, you can configure your deployed code to perform cache operations using Java objects as the media type. See [Overriding the MediaType Programmatically](#).

This approach allows remote clients to use a binary format for storing cache entries, which is optimal. However, you must make entity classes available to the server so that it can convert between binary format and Java objects.

Additionally, if the cache uses Protobuf (`application/x-protostream`) as the binary format, you must deploy protostream marshallers so that {ProductName} can unmarshall data from your custom code. See [Deploying Protostream Marshallers](#).

22.5.2. Storing Data as POJOs

Storing unmarshalled Java objects in the server is not recommended. Doing so requires {brandname} to serialize data when remote clients read from the cache and then deserialize data when remote clients write to the cache.

The following example configures the cache to store entries with the `application/x-java-object` media type:

```
<distributed-cache name="my-cache">
  <encoding>
    <key media-type="application/x-java-object"/>
    <value media-type="application/x-java-object"/>
  </encoding>
</distributed-cache>
```

Hot Rod clients must use a supported marshaller when data is stored as POJOs in the cache, either the JBoss marshaller or the default Java serialization mechanism. You must also deploy the classes must be deployed in the server.

REST clients must use a storage format that {brandname} can convert to and from Java objects, currently JSON or XML.



Storing Java objects in the cache requires you to deploy entity classes to {brandname}. See [Deploying Entity Classes](#).

Memcached clients must send and receive a serialized version of the stored POJO, which is a JBoss marshalled payload by default. However if you configure the [Client Encoding](#) in the appropriate Memcached connector, you change the storage format so that Memcached clients use a platform neutral format such as `JSON`.

This configuration is compatible with...

REST clients	Yes
Java Hot Rod clients	Yes
Non-Java Hot Rod clients	No

This configuration is compatible with...	
Querying and Indexing	Yes. However, querying and indexing works with POJOs only if the entities are annotated .
Custom Java objects	Yes

22.6. Deploying Entity Classes

If you plan to store entries in the cache as custom Java objects or POJOs, you must deploy entity classes to {brandname}. Clients always exchange objects as `bytes[]`. The entity classes represent those custom objects so that {brandname} can serialize and deserialize them.

To make entity classes available to the server, do the following:

- Create a **JAR** file that contains the entities and dependencies.
- Stop {brandname} if it is running.

{brandname} loads entity classes during boot. You cannot make entity classes available to {brandname} if the server is running.

- Copy the **JAR** file to the `$INFINISPAN_HOME/standalone/deployments/` directory.
- Specify the **JAR** file as a module in the cache manager configuration, as in the following example:

```
<cache-container name="local" default-cache="default">
  <modules>
    <module name="deployment.my-entities.jar"/>
  </modules>
  ...
</cache-container>
```

22.7. Trying the Interoperability Demo

Try the demo for protocol interoperability using the {brandname} Docker image at: <https://github.com/infinispan-demos/endpoint-interop>

Chapter 23. Security

Security within {brandname} is implemented at several layers:

- within the core library, to provide coarse-grained access control to CacheManagers, Caches and data
- over remote protocols, to obtain credentials from remote clients and to secure the transport using encryption
- between nodes in a cluster, so that only authorized nodes can join and to secure the transport using encryption

In order to maximize compatibility and integration, {brandname} uses widespread security standards where possible and appropriate, such as X.509 certificates, SSL/TLS encryption and Kerberos/GSSAPI. Also, to avoid pulling in any external dependencies and to increase the ease of integration with third party libraries and containers, the implementation makes use of any facilities provided by the standard Java security libraries (JAAS, JSSE, JCA, JCE, SASL, etc). For this reason, the {brandname} core library only provides interfaces and a set of basic implementations.

23.1. Embedded Security

Applications interact with {brandname} using its API within the same JVM. The two main components which are exposed by the {brandname} API are CacheManagers and Caches. If an application wants to interact with a secured CacheManager and Cache, it should provide an identity which {brandname}'s security layer will validate against a set of required roles and permissions. If the identity provided by the user application has sufficient permissions, then access will be granted, otherwise an exception indicating a security violation will be thrown. The identity is represented by the `javax.security.auth.Subject` class which is a wrapper around multiple Principals, e.g. a user and all the groups it belongs to. Since the Principal name is dependent on the owning system (e.g. a Distinguished Name in LDAP), {brandname} needs to be able to map Principal names to roles. Roles, in turn, represent one or more permissions. The following diagram shows the relationship between the various elements:

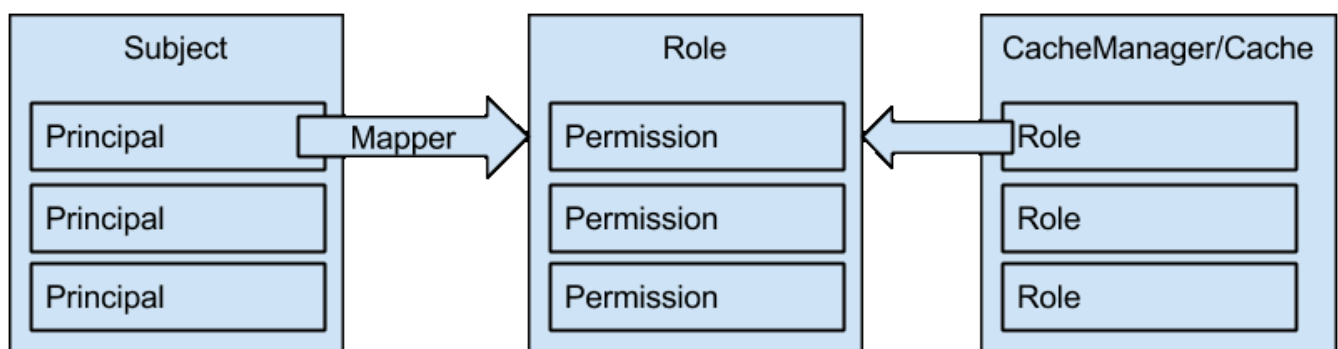


Figure 21. Roles/Permissions mapping

23.1.1. Embedded Permissions

Access to a cache manager or a cache is controlled by using a list of required permissions. Permissions are concerned with the type of action that is performed on one of the above entities

and not with the type of data being manipulated. Some of these permissions can be narrowed to specifically named entities, where applicable (e.g. a named cache). Depending on the type of entity, there are different types of permission available:

Cache Manager permissions

- **CONFIGURATION** (defineConfiguration): whether a new cache configuration can be defined
- **LISTEN** (addListener): whether listeners can be registered against a cache manager
- **LIFECYCLE** (stop): whether the cache manager can be stopped
- **ALL**: a convenience permission which includes all of the above

Cache permissions

- **READ** (get, contains): whether entries can be retrieved from the cache
- **WRITE** (put, putIfAbsent, replace, remove, evict): whether data can be written/replaced/removed/evicted from the cache
- **EXEC** (distexec, streams): whether code execution can be run against the cache
- **LISTEN** (addListener): whether listeners can be registered against a cache
- **BULK_READ** (keySet, values, entrySet, query): whether bulk retrieve operations can be executed
- **BULK_WRITE** (clear, putAll): whether bulk write operations can be executed
- **LIFECYCLE** (start, stop): whether a cache can be started / stopped
- **ADMIN** (getVersion, addInterceptor*, removeInterceptor, getInterceptorChain, getEvictionManager, getComponentRegistry, getDistributionManager, getAuthorizationManager, evict, getRpcManager, getCacheConfiguration, getCacheManager, getInvocationContextContainer, setAvailability, getDataContainer, getStats, getXAResource): whether access to the underlying components/internal structures is allowed
- **ALL**: a convenience permission which includes all of the above
- **ALL_READ**: combines **READ** and **BULK_READ**
- **ALL_WRITE**: combines **WRITE** and **BULK_WRITE**

Some permissions might need to be combined with others in order to be useful. For example, suppose you want to allow only "supervisors" to be able to run stream operations, while "standard" users can only perform puts and gets, you would define the following mappings:

```
<role name="standard" permission="READ WRITE" />
<role name="supervisors" permission="READ WRITE EXEC BULK"/>
```

23.1.2. Embedded API

When a `DefaultCacheManager` has been constructed with security enabled using either the programmatic or declarative configuration, it returns a `SecureCache` which will check the security context before invoking any operations on the underlying caches. A `SecureCache` also makes sure that applications cannot retrieve lower-level insecure objects (such as `DataContainer`). In Java,

executing code with a specific identity usually means wrapping the code to be executed within a `PrivilegedAction`:

```
import org.infinispan.security.Security;

Security.doAs(subject, new PrivilegedExceptionAction<Void>() {
    public Void run() throws Exception {
        cache.put("key", "value");
    }
});
```

If you are using Java 8, the above call can be simplified to:

```
Security.doAs(mySubject, PrivilegedAction<String>() -> cache.put("key", "value"));
```

Notice the use of `Security.doAs()` in place of the typical `Subject.doAs()`. While in `{brandname}` you can use either, unless you really need to modify the `AccessControlContext` for reasons specific to your application's security model, using `Security.doAs()` provides much better performance. If you need the current `Subject`, use the following:

```
Security.getSubject();
```

which will automatically retrieve the `Subject` either from the `{brandname}`'s context or from the `AccessControlContext`.

`{brandname}` also fully supports running under a full-blown `SecurityManager`. The `{brandname}` distribution contains an example `security.policy` file which you should customize with the appropriate paths before supplying it to your JVM.

23.1.3. Embedded Configuration

There are two levels of configuration: global and per-cache. The global configuration defines the set of roles/permissions mappings while each cache can decide whether to enable authorization checks and the required roles.

Programmatic

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
global
    .security()
        .authorization()
            .principalRoleMapper(new IdentityRoleMapper())
            .role("admin")
                .permission(AuthorizationPermission.ALL)
            .role("supervisor")
                .permission(AuthorizationPermission.EXEC)
                .permission(AuthorizationPermission.READ)
                .permission(AuthorizationPermission.WRITE)
            .role("reader")
                .permission(AuthorizationPermission.READ);
ConfigurationBuilder config = new ConfigurationBuilder();
config
    .security()
        .authorization()
            .enable()
            .role("admin")
            .role("supervisor")
            .role("reader");
```

Declarative

```
<infinispan>
  <cache-container default-cache="secured">
    <security>
      <authorization enabled="true">
        <identity-role-mapper />
        <role name="admin" permissions="ALL" />
        <role name="reader" permissions="READ" />
        <role name="writer" permissions="WRITE" />
        <role name="supervisor" permissions="READ WRITE EXEC BULK"/>
      </authorization>
    </security>
    <local-cache name="secured">
      <security>
        <authorization roles="admin reader writer supervisor" />
      </security>
    </local-cache>
  </cache-container>
</infinispan>
```

Role Mappers

In order to convert the Principals in a Subject into a set of roles to be used when authorizing, a suitable `PrincipalRoleMapper` must be specified in the global configuration. {brandname} comes

with 3 mappers and also allows you to provide a custom one:

- IdentityRoleMapper (Java: `org.infinispan.security.impl.IdentityRoleMapper`, XML: `<identity-role-mapper />`): this mapper just uses the Principal name as the role name
- CommonNameRoleMapper (Java: `org.infinispan.security.impl.CommonRoleMapper`, XML: `<common-name-role-mapper />`): if the Principal name is a Distinguished Name (DN), this mapper extracts the Common Name (CN) and uses it as a role name. For example the DN `cn=managers,ou=people,dc=example,dc=com` will be mapped to the role managers
- ClusterRoleMapper (Java: `org.infinispan.security.impl.ClusterRoleMapper` XML: `<cluster-role-mapper />`): a mapper which uses the ClusterRegistry to store principal to role mappings. This allows the use of the CLI's GRANT and DENY commands to add/remove roles to a principal.
- Custom role mappers (XML: `<custom-role-mapper class="a.b.c" />`): just supply the fully-qualified class name of an implementation of `org.infinispan.security.PrincipalRoleMapper`

23.2. Security Audit

{brandname} offers a pluggable audit logger which tracks whether a cache or a cache manager operation was allowed or denied. The audit logger is configured at the cache container authorization level:

Programmatic

```
GlobalConfigurationBuilder global = new GlobalConfigurationBuilder();
global
    .authorization()
        .auditLogger(new LoggingAuditLogger());
```

Declarative

```
<infinispan>
  <cache-container default-cache="secured">
    <security>
      <authorization audit-logger="org.infinispan.security.impl.LoggingAuditLogger"
">
      ...
    </authorization>
  </security>
  ...
</cache-container>
</infinispan>
```

In embedded mode the default audit logger is `org.infinispan.security.impl.NullAuditLogger` which does nothing. {brandname} also comes with the `org.infinispan.security.impl.LoggingAuditLogger` which outputs audit logs through the available logging framework (e.g. Log4J) at level TRACE and category AUDIT. These logs look like:

```
[ALLOW|DENY] user READ cache[defaultCache]
```

Using an appropriate logging appender it is possible to send the AUDIT category either to a log file, a JMS queue, a database, etc. The user which is included in the log above is the name of the first non-java.security.acl.Group principal in the Subject.

23.3. Cluster security

JGroups can be configured so that nodes need to authenticate each other when joining / merging. The authentication uses SASL and is setup by adding the SASL protocol to your JGroups XML configuration above the GMS protocol, as follows:

```
<SASL mech="DIGEST-MD5"
  client_name="node_user"
  client_password="node_password"
  server_callback_handler_class=
"org.example.infinispan.security.JGroupsSaslServerCallbackHandler"
  client_callback_handler_class=
"org.example.infinispan.security.JGroupsSaslClientCallbackHandler"
  sasl_props="com.sun.security.sasl.digest.realm=test_realm" />
```

In the above example, the SASL mech will be DIGEST-MD5. Each node will need to declare the user and password it will use when joining the cluster. The behaviour of a node differs depending on whether it is the coordinator or any other node. The coordinator acts as the SASL server, whereas joining/merging nodes act as SASL clients. Therefore two different CallbackHandlers are required, the `server_callback_handler_class` will be used by the coordinator, and the `client_callback_handler_class` will be used by the other nodes. The SASL protocol in JGroups is only concerned with the authentication process. If you wish to implement node authorization, you can do so within the server callback handler, by throwing an Exception. The following example shows how this can be done:


```

public class AuthorizingServerCallbackHandler implements CallbackHandler {

    @Override
    public void handle(Callback[] callbacks) throws IOException,
        UnsupportedCallbackException {
        for (Callback callback : callbacks) {
            ...
            if (callback instanceof AuthorizeCallback) {
                AuthorizeCallback acb = (AuthorizeCallback) callback;
                UserProfile user = UserManager.loadUser(acb.getAuthenticationID());
                if (!user.hasRole("myclusterrole")) {
                    throw new SecurityException("Unauthorized node " +user);
                }
            }
            ...
        }
    }
}

```

Chapter 24. Integrations

{brandname} can be integrated with a number of other projects, as detailed below.

24.1. Apache Spark

{brandname} provides an [Apache Spark](#) connector capable of exposing caches as an RDD, allowing batch and stream jobs to be run against data stored in {brandname}. For further details, see the [{brandname} Spark connector documentation](#). Also check the [Docker based Twitter demo](#).

24.2. Apache Hadoop

The {brandname} Hadoop connector can be used to expose {brandname} as a Hadoop compliant data source and sink that implements [InputFormat/OutputFormat](#). For further details, refer to the full [documentation](#).

24.3. Apache Lucene

{brandname} includes a highly scalable distributed [Apache Lucene Directory](#) implementation.

This directory closely mimics the same semantics of the traditional filesystem and RAM-based directories, being able to work as a drop-in replacement for existing applications using Lucene and providing reliable index sharing and other features of {brandname} like node auto-discovery, automatic failover and rebalancing, optionally transactions, and can be backed by traditional storage solutions as filesystem, databases or cloud store engines.

The implementation extends Lucene's *org.apache.lucene.store.Directory* so it can be used to *store* the index in a cluster-wide shared memory, making it easy to distribute the index. Compared to rsync-based replication this solution is suited for use cases in which your application makes frequent changes to the index and you need them to be quickly distributed to all nodes. Consistency levels, synchronicity and guarantees, total elasticity and auto-discovery are all configurable; also changes applied to the index can optionally participate in a JTA transaction, optionally supporting XA transactions with recovery.

Two different *LockFactory* implementations are provided to guarantee only one *IndexWriter* at a time will make changes to the index, again implementing the same semantics as when opening an index on a local filesystem. As with other Lucene Directories, you can override the *LockFactory* if you prefer to use an alternative implementation.

24.3.1. Lucene compatibility

Apache Lucene versions 5.5.x

24.3.2. Maven dependencies

All you need is the following:

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-lucene-directory</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

24.3.3. How to use it

See the below example of using the {brandname} Lucene Directory in order to index and query a single Document:

```

import java.io.IOException;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.document.StringField;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.index.Term;
import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.TermQuery;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.infinispan.lucene.directory.DirectoryBuilder;
import org.infinispan.manager.DefaultCacheManager;

// Create caches that will store the index. Here the programmatic configuration is
used
DefaultCacheManager defaultManager = new DefaultCacheManager();
Cache metadataCache = defaultManager.getCache("metadataCache");
Cache dataCache = defaultManager.getCache("dataCache");
Cache lockCache = defaultManager.getCache("lockCache");

// Create the directory
Directory directory = DirectoryBuilder.newDirectoryInstance(metadataCache, dataCache,
lockCache, indexName).create();

// Use the directory in Lucene
IndexWriterConfig indexWriterConfig = new IndexWriterConfig(new StandardAnalyzer())
.setOpenMode(IndexWriterConfig.OpenMode.CREATE_OR_APPEND);

IndexWriter indexWriter = new IndexWriter(directory, indexWriterConfig);

// Index a single document
Document doc = new Document();
doc.add(new StringField("field", "value", Field.Store.NO));
indexWriter.addDocument(doc);
indexWriter.close();

// Querying the inserted document
DirectoryReader directoryReader = DirectoryReader.open(directory);
IndexSearcher searcher = new IndexSearcher(directoryReader);
TermQuery query = new TermQuery(new Term("field", "value"));
TopDocs topDocs = searcher.search(query, 10);
System.out.println(topDocs.totalHits);

```

The *indexName* in the *DirectoryBuilder* is a unique key to identify your index. It takes the same role as the path did on filesystem based indexes: you can create several different indexes giving them

different names. When you use the same *indexName* in another instance connected to the same network (or instantiated on the same machine, useful for testing) they will join, form a cluster and share all content. Using a different *indexName* allows you to store different indexes in the same set of Caches.

The *metadataCache*, *dataCache* and *lockCache* are the caches that will store the indexes. More details provided below.

New nodes can be added or removed dynamically, making the service administration very easy and also suited for cloud environments: it's simple to react to load spikes, as adding more memory and CPU power to the search system is done by just starting more nodes.

24.3.4. Configuration

{brandname} can be configured as LOCAL clustering mode, in which case it will disable clustering features and serve as a cache for the index, or any clustering mode. A transaction manager is not mandatory, but when enabled the changes to the index can participate in transactions.

Batching was required in previous versions, it's not strictly needed anymore.

As pointed out in the javadocs of [DirectoryBuilder](#), it's possible for it to use more than a single cache, using specific configurations for different purposes. Each cache is explained below:

Lock Cache

The lock cache is used to store a single entry per index that will function as the directory lock. Given the small storage requirement this cache is usually configured as REPL_SYNC. Example of declarative configuration:

```
<replicated-cache name="LuceneIndexesLocking" mode="SYNC" remote-timeout="25000">
  <transaction mode="NONE"/>
  <indexing index="NONE" />
  <memory>
    <object size="-1"/>
  </memory>
</replicated-cache>
```

Metadata Cache

The metadata cache is used to store information about the files of the directory, such as buffer sizes and number of chunks. It uses more space than the Lock Cache, but not as much as the Data Cache, so using a REPL_SYNC cache should be fine for most cases. Example of configuration:

```
<replicated-cache name="LuceneIndexesMetadaData" mode="SYNC" remote-timeout="25000">
  <transaction mode="NONE"/>
  <indexing index="NONE" />
  <memory>
    <object size="-1"/>
  </memory>
</replicated-cache>
```

Data Cache

The {brandname} Lucene directory splits large (bigger than the chunkSize configuration) files into chunks and stores them in the Data cache. This is the largest of the 3 index caches, and both DIST_SYNC/REPL_SYNC cache modes can be used. Usage of REPL_SYNC offers lower latencies for queries since each node holds the whole index locally; DIST_SYNC, on the other hand, will affect query latency due to remote calls to fetch for chunks, but offers better scalability.

Example of configuration:

```
<distributed-cache name="LuceneIndexesData" mode="SYNC" remote-timeout="25000">
  <transaction mode="NONE"/>
  <indexing index="NONE" />
  <memory>
    <object size="-1"/>
  </memory>
</distributed-cache>
```

24.3.5. Using a CacheLoader

Using a CacheLoader you can have the index content backed up to a permanent storage; you can use a shared store for all nodes or one per node, see [cache passivation](#) for more details.

When using a CacheLoader to store a Lucene index, to get best write performance you would need to configure the CacheLoader with *async=true* .

24.3.6. Storing the index in a database

It might be useful to store the Lucene index in a relational database; this would be very slow but {brandname} can act as a cache between the application and the JDBC interface, making this configuration useful in both clustered and non-clustered configurations. When storing indexes in a JDBC database, it's suggested to use the *JdbcStringBasedCacheStore* , which will need the *key-to-string-mapper* attribute to be set to *org.infinispan.lucene.LuceneKey2StringMapper*:

```
<jdbc:string-keyed-jdbc-store preload="true" key-to-string-mapper=
  "org.infinispan.lucene.LuceneKey2StringMapper">
```

24.3.7. Loading an existing Lucene Index

The `org.infinispan.lucene.cachestore.LuceneCacheLoader` is an {brandname} `CacheLoader` able to have {brandname} directly load data from an existing Lucene index into the grid. Currently this supports reading only.

Property	Description	Default
<i>location</i>	The path where the indexes are stored. Subdirectories (of first level only) should contain the indexes to be loaded, each directory matching the index name attribute of the {brandname} <code>Directory</code> constructor.	none (mandatory)
<i>autoChunkSize</i>	A threshold in bytes: if any segment is larger than this, it will be transparently chunked in smaller cache entries up to this size.	32MB

It's worth noting that the IO operations are delegated to Lucene's standard `org.apache.lucene.store.FSDirectory`, which will select an optimal approach for the running platform.

Implementing write-through should not be hard: you're welcome to try implementing it.

24.3.8. Architectural limitations

This `Directory` implementation makes it possible to have almost real-time reads across multiple nodes. A fundamental limitation of the Lucene design is that only a single `IndexWriter` is allowed to make changes on the index: a pessimistic lock is acquired by the writer; this is generally ok as a single `IndexWriter` *instance* is very fast and accepts update requests from multiple threads. When sharing the `Directory` across {brandname} nodes the `IndexWriter` limitation is not lifted: since you can have only one instance, that reflects in your application as having to apply all changes on the same node. There are several strategies to write from multiple nodes on the same index:

Index write strategies

- One node writes, the other delegate to it sending messages
- Each node writes on turns
- Your application makes sure it will only ever apply index writes on one node

The {brandname} *Lucene Directory* protects its content by implementing a distributed locking strategy, though this is designed as a last line of defense and is not to be considered an efficient mechanism to coordinate multiple writes: if you don't apply one of the above suggestions and get high write contention from multiple nodes you will likely get timeout exception.

24.3.9. Suggestions for optimal performance

JGroups and networking stack

JGroups manages all network IO and as such it is a critical component to tune for your specific environment. Make sure to read the [JGroups reference documentation](#), and play with the performance tests included in JGroups to make sure your network stack is setup appropriately. Don't forget to check also operating system level parameters, for example buffer sizes dedicated for networking. JGroups will log warning when it detects something wrong, but there is much more you can look into.

Using a CacheStore

Currently all CacheStore implementations provided by {brandname} have a significant slowdown; we hope to resolve that soon but for the time being if you need high performance on writes with the Lucene Directory the best option is to disable any CacheStore; the second best option is to configure the CacheStore as *async* . If you only need to load a Lucene index from read-only storage, see the above description for *org.infinispan.lucene.cachestore.LuceneCacheLoader* .

Apply standard Lucene tuning

All known options of Lucene apply to the {brandname} Lucene Directory as well; of course the effect might be less significant in some cases, but you should definitely read the [Apache Lucene documentation](#) .

Disable batching and transactions

Early versions required {brandname} to have batching or transactions enabled. This is no longer a requirement, and in fact disabling them should provide little improvement in performance.

Set the right chunk size

The chunk size can be specified using the [DirectoryBuilder](#) fluent API. To correctly set this variable you need to estimate what the expected size of your segments is; generally this is trivial by looking at the file size of the index segments generated by your application when it's using the standard FSDirectory. You then have to consider:

- The chunk size affects the size of internally created buffers, and large chunk sizes will cause memory usage to grow. Also consider that during index writing such arrays are frequently allocated.
- If a segment doesn't fit in the chunk size, it's going to be fragmented. When searching on a fragmented segment performance can't peak.

Using the *org.apache.lucene.index.IndexWriterConfig* you can tune your index writing to *approximately* keep your segment size to a reasonable level, from there then tune the chunksize, after having defined the chunksize you might want to revisit your network configuration settings.

24.3.10. Demo

There is a simple command-line demo of its capabilities distributed with {brandname} under demos/lucene-directory; make sure you grab the "Binaries, server and demos" package from download page, which contains all demos.

Start several instances, then try adding text in one instance and searching for it on the other. The configuration is not tuned at all, but should work out-of-the box without any changes. If your network interface has multicast enabled, it will cluster across the local network with other instances of the demo.

24.3.11. Additional Links

- Issue tracker: <https://jira.jboss.org/browse/ISPN/component/12312732>
- Source code: <https://github.com/infinispan/infinispan/tree/master/lucene/lucene-directory/src/main/java/org/infinispan/lucene>

24.4. Directory Provider for Hibernate Search

Hibernate Search applications can use {brandname} as a directory provider, taking advantage of {brandname}'s distribution and low latency capabilities to store the Lucene indexes.

24.4.1. Maven dependencies

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-directory-provider</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

24.4.2. How to use it

The directory provider alias is "infinispan", and to enable it for an index, the following property should be in the [Hibernate Search configuration](#):

```
hibernate.search.MyIndex.directory_provider = infinispan
```

to enable it by default for all indexes:

```
hibernate.search.default.directory_provider = infinispan
```

The {brandname} cluster will start with a [default configuration](#), see below how to override it.

24.4.3. Configuration

Optional properties allow for a custom {brandname} configuration or to use an existent *EmbeddedCacheManager*:

Property	Description	Example value
<code>hibernate.search.infinispan.configuration_resourcename</code>	Custom configuration for {brandname}	config/infinispan.xml
<code>hibernate.search.infinispan.configuration.transport_override_resourcename</code>	Overrides the JGroups stack in the {brandname} configuration file	jgroups-ec2.xml
<code>hibernate.search.infinispan.cache_manager_jndiname</code>	Specifies the JNDI name under which the <i>EmbeddedCacheManager</i> to use is bound. Will cause the properties above to be ignored when present	<code>java:jboss/infinispan/container/hibernate-search</code>

24.4.4. Architecture considerations

The same limitations presented in the Lucene Directory apply here, meaning the index will be shared across several nodes and only one *IndexWriter* can have the lock.

One common strategy is to use Hibernate Search's JMS Master/Slave or JGroups backend together with the {brandname} directory provider: instead of sending updates directly to the index, they are sent to a JMS queue or JGroups channel and a single node applies all the changes on behalf of all other nodes.

Refer to the [Hibernate Search documentation](#) for instructions on how to setup JMS or JGroups backends.

24.5. JPA/Hibernate 2L Cache

Hibernate manages a second-level cache where it moves data into and out as a result of operations performed by *Session* or *EntityManager* (JPA). The second-level cache is pluggable via an SPI which {brandname} implements. This enables {brandname} to be used as second-level cache for Hibernate.

[Hibernate documentation](#) contains a lot of information about second-level cache, types of caches... etc. This chapter focuses on what you need to know to use {brandname} as second-level cache provider with Hibernate.

Applications running in environments where {brandname} is not default cache provider for Hibernate will need to depend on the correct cache provider version.

The {brandname} cache provider version suitable for your application depends on the Hibernate version in use:

Hibernate 5.3

Use the following Maven coordinates:

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-hibernate-cache-v53</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

Hibernate 5.2



Hibernate 5.2 is supported in {brandname} 9.2.x only.

Use the following Maven coordinates:

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-hibernate-cache</artifactId>
  <version>${version.infinispan}</version> <!-- 9.2.x.Final -->
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname} 9.2.x.

Hibernate 5.1

Use the following Maven coordinates:

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-hibernate-cache-v51</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.



Hibernate version 5.0 and earlier: the {brandname} cache provider is shipped by Hibernate. Documentation and Maven coordinates are located in the [Hibernate documentation](#).

Apart from {brandname} specific configuration, it's worth noting that enabling second cache requires some changes to the descriptor file (`persistence.xml` for JPA or `application.properties` for Spring). To use second level cache, you first need to enable the second level cache so that entities and/or collections can be cached:

Table 7. Enable second-level cache

JPA	<code><property name="hibernate.cache.use_second_level_cache" value="true"/></code>
Spring	<code>spring.jpa.properties.hibernate.cache.use_second_level_cache=true</code>

To select which entities/collections to cache, first annotate them with `javax.persistence.Cacheable`. Then make sure shared cache mode is set to `ENABLE_SELECTIVE`:

Table 8. Enable selective shared cached mode

JPA	<code><shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode></code>
Spring	<code>spring.jpa.properties.javax.persistence.sharedCache.mode=ENABLE_SELECTIVE</code>



This is the most common way of selecting which entities/collections to cache. However, there are alternative ways to which are explained in the [Hibernate documentation](#).

Optionally, queries can also be cached but for that query cache needs to be enabled:

Table 9. Enable query cache

JPA	<code><property name="hibernate.cache.use_query_cache" value="true"/></code>
Spring	<code>spring.jpa.properties.hibernate.cache.use_query_cache=true</code>



As well as enabling query cache, forcing a query to be cached requires the query to be made cacheable. For example, for JPA queries: `query.setHint("org.hibernate.cacheable", Boolean.TRUE)`.

The best way to find out whether second level cache is working or not is to inspect the statistics. By inspecting the statistics you can verify if the cache is being hit, if any new data is stored in cache... etc. Statistics are disabled by default, so it is recommended that you enable statistics:

Table 10. Enable statistics

JPA	<code><property name="hibernate.generate_statistics" value="true" /></code>
Spring	<code>spring.jpa.properties.hibernate.generate_statistics=true</code>

24.5.1. Deployment Scenarios

How to configure {brandname} to be the second level cache provider varies slightly depending on the deployment scenario:

Single-Node Standalone Hibernate Application

In standalone library mode, a JPA/Hibernate application runs inside a Java SE application or inside containers that don't offer {brandname} integration.

Enabling {brandname} second level cache provider inside a JPA/Hibernate application that runs in single node is very straightforward. First, make sure the Hibernate {brandname} cache provider is available in the classpath. Then, modify the persistence.xml to include these properties:

```

<!-- Use Infinispan second level cache provider -->
<property name="hibernate.cache.region.factory_class" value="infinispan"/>

<!--
    Force using local configuration when only using a single node.
    Otherwise a clustered configuration is loaded.
-->
<property name="hibernate.cache.infinispan.cfg"
    value="org/infinispan/hibernate/cache/commons/builder/infinispan-configs-
local.xml"/>

```

By default when running standalone, the {brandname} second-level cache provider uses an {brandname} configuration that's designed for clustered environments. However, {brandname} also provides a configuration designed for local, single node, environments. To enable that configuration, set `hibernate.cache.infinispan.cfg` to `org/infinispan/hibernate/cache/commons/builder/infinispan-configs-local.xml` value. You can find more about the configuration check the [Default Local Configuration](#) section.

A simple tutorial showing how to use {brandname} as Hibernate cache provider in a standalone application can be found [here](#).

Single-Node Standalone Spring Application

Using Hibernate within Spring applications is a very common use case. In this section you will learn what you need to do configure Hibernate within Spring to use {brandname} as second-level cache provider.

As in the previous case, start by making sure that Hibernate {brandname} Cache provider is available in the classpath. Then, modify `application.properties` file to contain:

```

# Use Infinispan second level cache provider
spring.jpa.properties.hibernate.cache.region.factory_class=infinispan
#
# Force using local configuration when only using a single node.
# Otherwise a clustered configuration is loaded.
spring.jpa.properties.hibernate.cache.infinispan.cfg=org/infinispan/hibernate/cache/co
mmons/builder/infinispan-configs-local.xml

```

By default when running standalone, the {brandname} second-level cache provider uses an {brandname} configuration that's designed for clustered environments. However, {brandname} also provides a configuration designed for local, single node, environments. To enable that configuration, set `spring.jpa.properties.hibernate.cache.infinispan.cfg` to `org/infinispan/hibernate/cache/commons/builder/infinispan-configs-local.xml` value. You can find more about the configuration check the [Default Local Configuration](#) section.

A simple tutorial showing how to use {brandname} as Hibernate cache provider in a Spring application can be found [here](#).

Single-Node WildFly Application

In WildFly, {brandname} is the default second level cache provider for JPA/Hibernate. This means that when using JPA in WildFly, region factory is already set to `infinispan`. {brandname}'s configuration is located in WildFly's `standalone.xml` file. It follows the same settings explained in [Default Local Configuration](#) section.



When running in Wildfly, do not set `hibernate.cache.infinispan.cfg`. The configuration of the caches comes from WildFly's configuration file.

Several aspects of the {brandname} second level cache provider can be configured directly in `persistence.xml`. This means that some of those tweaks do not require changing WildFly's `standalone.xml` file. You can find out more about these changes in the [Configuration Properties](#) section.

So, to enable Hibernate to use {brandname} as second-level cache, all you need to do is enable second-level cache. This is explained in detail in the introduction of this chapter.

A simple tutorial showing how to use {brandname} as Hibernate cache provider in a WildFly application can be found [here](#).

Multi-Node Standalone Hibernate Application

When running a JPA/Hibernate in a multi-node environment and enabling {brandname} second-level cache, it is necessary to cluster the second-level cache so that cache consistency can be guaranteed. Clustering the {brandname} second-level cache provider is as simple as adding the following property to the `persistence.xml` file:

```
<!-- Use Infinispan second level cache provider -->
<property name="hibernate.cache.region.factory_class" value="infinispan"/>
```

The default {brandname} configuration used by the second-level cache provider is already configured to work in a cluster environment, so no need to add any extra properties. You can find more about the configuration check the [Default Cluster Configuration](#) section.

Multi-Node Standalone Spring Application

If interested in running a Spring application that uses Hibernate and {brandname} as second level cache, the cache needs to be clustered. Clustering the {brandname} second-level cache provider is as simple as adding the following property to the `application.properties` file:

```
# Use Infinispan second level cache provider
spring.jpa.properties.hibernate.cache.region.factory_class=infinispan
```

The default {brandname} configuration used by the second-level cache provider is already configured to work in a cluster environment, so no need to add any extra properties. You can find more about the configuration check the [Default Cluster Configuration](#) section.

Multi-Node Wildfly Application

As mentioned in the single node Wildfly case, {brandname} is the default second level cache provider for JPA/Hibernate when running inside Wildfly. This means that when using JPA in WildFly, region factory is already set to `infinispan`.

When running Wildfly multi-node clusters, it is recommended that you start off by using `standalone-ha.xml` configuration file. Within this file you can find Hibernate {brandname} caches configured with the correct settings to work in a clustered environment. You can find more about the configuration check the [Default Cluster Configuration](#) section.

Several aspects of the {brandname} second level cache provider can be configured directly in `persistence.xml`. This means that some of those tweaks do not require changing WildFly's `standalone-ha.xml` file. You can find out more about these changes in the [Configuration Properties](#) section.

So, to enable Hibernate to use {brandname} as second-level cache, all you need to do is enable second-level cache. Enabling second-level cache is explained in detail in the introduction of this chapter.

24.5.2. Configuration Reference

This section is dedicated at explaining configuration in detail as well as some extra configuration options.

Default Local Configuration

{brandname} second-level cache provider comes with a configuration designed for local, single node, environments. These are the characteristics of such configuration:

Entities, collections, queries and timestamps are stored in non-transactional local caches.

Entities and collections query caches are configured with the following eviction settings:

- Eviction wake up interval is 5 seconds.
- Max number of entries are 10,000.
- Max idle time before expiration is 100 seconds.
- Default eviction algorithm is LRU, least recently used.

You can change these settings on a per entity or collection basis or per individual entity or collection type. More information in the [Configuration Properties](#) section below.

No eviction/expiration is configured for timestamp caches, nor it's allowed.

Default Cluster Configuration

{brandname} second-level cache provider default configuration is designed for multi-node clustered environments. The aim of this section is to explain the default settings for each of the different global data type caches (entity, collection, query and timestamps), why these were chosen and what are the available alternatives. These are the characteristics of such configuration:

By default all *entities and collections* are configured to use a *synchronous invalidation* as clustering mode. Whenever a new *entity or collection* is read from database and needs to be cached, *it's only cached locally* in order to reduce intra-cluster traffic. This option can be changed so that entities/collections are cached cluster wide, by switching the entity/collection cache to be replicated or distributed. How to change this option is explained in the [Configuration Properties](#) section.



When data read from the database is put in the cache, with replicated or distributed caches, the data is propagated to other nodes using asynchronous communication. In the presence of concurrent database loads, one operation will succeed while others might fail (silently). This is fine because they'd all be trying to put the same data loaded from the database. This has the side effect that under these circumstances, the cache might not be up to date right after making the JPA call that leads to the database load. However, the cache will eventually contain the data loaded, even if it happens after a short delay.

All *entities and collections* are configured to use a *synchronous invalidation* as clustering mode. This means that when an entity is updated, the updated cache will send a message to the other members of the cluster telling them that the entity has been modified. Upon receipt of this message, the other nodes will remove this data from their local cache, if it was stored there. This option can be changed so that both local and remote nodes contain the updates by configuring entities or collections to use a replicated or distributed cache. With replicated caches all nodes would contain the update, whereas with distributed caches only a subset of the nodes. How to change this option is explained in the [Configuration Properties](#) section.

All *entities and collections* have *initial state transfer disabled* since there's no need for it.

Entities and collections are configured with the following eviction settings. You can change these settings on a per entity or collection basis or per individual entity or collection type. More information in the [Configuration Properties](#) section below.

- Eviction wake up interval is 5 seconds.
- Max number of entries are 10,000.
- Max idle time before expiration is 100 seconds.
- Default eviction algorithm is LRU, least recently used.

Queries

Assuming that query caching has been enabled for the persistence unit (see chapter introduction), the query cache is configured so that *queries are only cached locally*. Alternatively, you can configure query caching to use replication by selecting the **replicated-query** as query cache name. However, replication for query cache only makes sense if, and only if, all of this conditions are true:

- Performing the query is quite expensive.
- The same query is very likely to be repeatedly executed on different cluster nodes.
- The query is unlikely to be invalidated out of the cache



Hibernate must aggressively invalidate query results from the cache any time any instance of one of the entity types targeted by the query. All such query results are invalidated, even if the change made to the specific entity instance would not have affected the query result. For example: the cached result of `SELECT id FROM cars where color = 'red'` is thrown away when you call `INSERT INTO cars VALUES ..., color = 'blue'`. Also, the result of an update within a transaction is not visible to the result obtained from the query cache.

query cache uses the *same eviction/expiration settings as for entities/collections*.

query cache has *initial state transfer disabled*. It is not recommended that this is enabled.

Up to Hibernate 5.2 both transactional and non-transactional query caches have been supported, though non-transactional variant is recommended. Hibernate 5.3 drops support for transactional caches, only non-transactional variant is supported. If the cache is configured with transactions this setting is ignored and warning is logged.

Timestamps

The *timestamps cache* is configured with *asynchronous replication* as clustering mode. Local or invalidated cluster modes are not allowed, since all cluster nodes must store all timestamps. As a result, *no eviction/expiration is allowed for timestamp caches either*.



Asynchronous replication was selected as default for timestamps cache for performance reasons. A side effect of this choice is that when an entity/collection is updated, for a very brief period of time stale queries might be returned. It's important to note that due to how {brandname} deals with asynchronous replication, stale queries might be found even query is done right after an entity/collection update on same node.



Hibernate must aggressively invalidate query results from the cache any time any instance of one of the entity types is modified. All cached query results referencing given entity type are invalidated, even if the change made to the specific entity instance would not have affected the query result. The timestamps cache plays here an important role - it contains last modification timestamp for each entity type. After a cached query results is loaded, its timestamp is compared to all timestamps of the entity types that are referenced in the query. If any of these is higher, the cached query result is discarded and the query is executed against DB. This requires synchronization of the wall clock across the cluster to work as expected.

Configuration Properties

As explained above, {brandname} second-level cache provider comes with default configuration in `infinispan-config.xml` that is suited for clustered use. If there's only single JVM accessing the DB, you can use more performant `infinispan-config-local.xml` by setting the `hibernate.cache.infinispan.cfg` property. If you require further tuning of the cache, you can provide your own configuration. Caches that are not specified in the provided configuration will default to `infinispan-config.xml` (if the provided configuration uses clustering) or `infinispan-`

config-local.xml.



It is not possible to specify the configuration this way in WildFly. Cache configuration changes in Wildfly should be done either modifying the cache configurations inside the application server configuration, or creating new caches with the desired tweaks and plugging them accordingly. See examples below on how entity/collection specific configurations can be applied.

Example 1. Use custom {brandname} configuration

```
<property
  name="hibernate.cache.infinispan.cfg"
  value="my-infinispan-configuration.xml" />
```



If the cache is configured as transactional, {brandname} cache provider automatically sets transaction manager so that the TM used by {brandname} is the same as TM used by Hibernate.

Cache configuration can differ for each type of data stored in the cache. In order to override the cache configuration template, use property `hibernate.cache.infinispan.data-type.cfg` where `data-type` can be one of:

- **entity**: Entities indexed by `@Id` or `@EmbeddedId` attribute.
- **immutable-entity**: Entities tagged with `@Immutable` annotation or set as `mutable=false` in mapping file.
- **naturalid**: Entities indexed by their `@NaturalId` attribute.
- **collection**: All collections.
- **timestamps**: Mapping *entity type* → *last modification timestamp*. Used for query caching.
- **query**: Mapping *query* → *query result*.
- **pending-puts**: Auxiliary caches for regions using invalidation mode caches.

For specifying cache template for specific region, use region name instead of the `data-type`:

Example 2. Use custom cache template

```
<property
  name="hibernate.cache.infinispan.entities.cfg"
  value="custom-entities" />
<property
  name="hibernate.cache.infinispan.query.cfg"
  value="custom-query-cache" />
<property
  name="hibernate.cache.infinispan.com.example.MyEntity.cfg"
  value="my-entities" />
<property
  name="hibernate.cache.infinispan.com.example.MyEntity.someCollection.cfg"
  value="my-entities-some-collection" />
```

Use custom cache template in Wildfly

When applying entity/collection level changes inside JPA applications deployed in Wildfly, it is necessary to specify deployment name and persistence unit name (separated by # character):

```
<property
  name=
    "hibernate.cache.infinispan._war_or_ear_name_#_unit_name_.com.example.MyEntity.cfg"
  value="my-entities" />
<property
  name=
    "hibernate.cache.infinispan._war_or_ear_name_#_unit_name_.com.example.MyEntity.someCollection.cfg"
  value="my-entities-some-collection" />
```



Cache configurations are used only as a template for the cache created for given region. Usually each entity hierarchy or collection has its own region



Except for eviction/expiration settings, it is highly recommended not to deviate from the template configuration settings.

Some options in the cache configuration can also be overridden directly through properties. These are:

- `hibernate.cache.infinispan.something.eviction.strategy`: Available options are `NONE`, `LRU` and `LIRS`.
- `hibernate.cache.infinispan.something.eviction.max_entries`: Maximum number of entries in the cache.

- `hibernate.cache.infinispan.something.expiration.lifespan`: Lifespan of entry from insert into cache (in milliseconds).
- `hibernate.cache.infinispan.something.expiration.max_idle`: Lifespan of entry from last read/modification (in milliseconds).
- `hibernate.cache.infinispan.something.expiration.wake_up_interval`: Period of thread checking expired entries.
- `hibernate.cache.infinispan.statistics`: Globally enables/disable {brandname} statistics collection, and their exposure via JMX.

Example:

```
<property name="hibernate.cache.infinispan.entity.eviction.strategy"
  value= "LRU"/>
<property name="hibernate.cache.infinispan.entity.eviction.wake_up_interval"
  value= "2000"/>
<property name="hibernate.cache.infinispan.entity.eviction.max_entries"
  value= "5000"/>
<property name="hibernate.cache.infinispan.entity.expiration.lifespan"
  value= "60000"/>
<property name="hibernate.cache.infinispan.entity.expiration.max_idle"
  value= "30000"/>
```

With the above configuration, you're overriding whatever eviction/expiration settings were defined for the default entity cache name in the {brandname} cache configuration used. This happens regardless of whether it's the default one or user defined. More specifically, we're defining the following:

- All entities to use LRU eviction strategy
- The eviction thread to wake up every 2 seconds (2000 milliseconds)
- The maximum number of entities for each entity type to be 5000 entries
- The lifespan of each entity instance to be 1 minute (60000 milliseconds).
- The maximum idle time for each entity instance to be 30 seconds (30000 milliseconds).

You can also override eviction/expiration settings on a per entity/collection type basis. This allows overrides that only affects a particular entity (i.e. `com.acme.Person`) or collection type (i.e. `com.acme.Person.addresses`). Example:

```
<property name="hibernate.cache.infinispan.com.acme.Person.eviction.strategy"
  value= "LIRS"/>
```

Inside of Wildfly, same as with the entity/collection configuration override, eviction/expiration settings would also require deployment name and persistence unit information (a working example can be found [here](#)):

```

<property name=
"hibernate.cache.infinispan._war_or_ear_name_#_unit_name_.com.acme.Person.eviction.str
ategy"
    value= "LIRS"/>
<property name=
"hibernate.cache.infinispan._war_or_ear_name_#_unit_name_.com.acme.Person.expiration.l
ifespans"
    value= "65000"/>

```

24.5.3. Cache Strategies

{brandname} cache provider supports all Hibernate cache strategies: `transactional`, `read-write`, `nonstrict-read-write` and `read-only`.

Integrations with Hibernate 4.x required *transactional invalidation caches* and in integrations with Hibernate ≤ 5.2 *transactional invalidation caches* are supported (in JTA environment). However for all 5.x versions *non-transactional caches* are preferred. With Hibernate 5.3 the support for transactional caches has been dropped completely, and both `read-write` and `transactional` use the same implementation. {brandname} provides the same consistency guarantees for both `transactional` and `read-write` strategies, use of transactions is considered an implementation detail.

In integrations with Hibernate 5.2 or lower the actual setting of cache concurrency mode (`read-write` vs. `transactional`) is not honored on invalidation caches, the appropriate strategy is selected based on the cache configuration (*non-transactional* vs. *transactional*).

Support for *replicated/distributed* caches for `read-write` and `read-only` strategies has been added during 5.x development and this requires exclusively *non-transactional configuration*. Also eviction should not be used in this configuration as it can lead to consistency issues. Expiration (with reasonably long max-idle times) can be used.

`Nonstrict-read-write` strategy is supported on *non-transactional distributed/replicated* caches, but the eviction should be turned off as well. In addition to that, the entities must use versioning. This means that this strategy cannot be used for caching natural IDs (which are never versioned). This mode mildly relaxes the consistency - between DB commit and end of transaction commit a stale read may occur in another transaction. However this strategy uses less RPCs and can be more performant than the other ones.

`Read-only` mode is supported in all configurations mentioned above but use of this mode currently does not bring any performance gains.

The available combinations are summarized in table below:

Table 11. Cache concurrency strategy/cache mode compatibility table

Concurrency strategy	Cache transactions	Cache mode	Eviction
transactional	≤ 5.2 transactional	invalidation	yes

Concurrency strategy	Cache transactions	Cache mode	Eviction
transactional	≥ 5.3 non-transactional	invalidation	yes
read-write	non-transactional	invalidation	yes
read-write	non-transactional	distributed/replicated	no
nonstrict-read-write	non-transactional	distributed/replicated	no

Changing caches to behave different to the default behaviour explained in previous section is explained in the [Configuration Properties](#) section.



Use of transactional caches is possible only in JTA environment. Hibernate supports JDBC-only transactions but {brandname} transactional caches do not integrate with these. Therefore, in non-JTA environment the only option is to use **read-write**, **nonstrict-read-write** or **read-only** on non-transactional cache. Configuring the cache as transactional in non-JTA can lead to undefined behaviour.

*Example 3. Stale read with **nonstrict-read-write** strategy*

```
A=0 (non-cached), B=0 (cached in 2LC)
TX1: write A = 1, write B = 1
TX1: start commit
TX1: commit A, B in DB
TX2: read A = 1 (from DB), read B = 0 (from 2LC) // breaks transactional atomicity
TX1: update A, B in 2LC
TX1: end commit
Tx3: read A = 1, B = 1 // reads after TX1 commit completes are consistent again
```

24.5.4. Using minimal puts

Hibernate offers a configuration option **hibernate.cache.use_minimal_puts** which is off by default in {brandname} implementation. This option checks if the cache contains given key before updating the value from database (put-from-load) and omits the update if the cached value is already present. When using invalidation caches it makes sense to keep this off as the put-from-load is local node-only and silently fails if the entry is locked. With replicated/distributed caches the update is applied to remote nodes, even if the local node already contains the entry, and this has higher performance impact, so it might make sense to turn this option on and avoid updating the cache.

24.6. JPA / Hibernate OGM

Hibernate can perform CRUD operations directly on an {brandname} cluster.

Hibernate OGM is an extension of the popular Hibernate ORM project which makes the Hibernate API suited to interact with NoSQL databases such as {brandname}.

When some of your object graphs need high scalability and elasticity, you can use Hibernate OGM to store these specific entities into {brandname} instead of your traditional RDBMS. The drawback is that {brandname} - not being a relational database - can not run complex relational queries.

Hibernate OGM allows you to get started with {brandname} in minutes, as:

- the JPA API and its annotations are simple and well known
- you don't need to learn Protobuf or Externalizer encoding formats
- no need to learn the {brandname} API
- the Hot Rod client is also setup and managed for you

It will still be beneficial to eventually learn how to configure {brandname} for top performance and learn about all capabilities it has, but you can get a proof of concept application done quickly with the example configuration.

Hibernate OGM also gives you several more benefits; being designed and implemented in collaboration with the {brandname} team it incorporates experience and deep understanding of how to best perform some common operations.

For example a common mistake for people new to {brandname} is to "serialize" Java POJOs for long term storage of important information; the {brandname} API allows this as it's useful for short lived caching of metadata, but you wouldn't be able to de-serialize your data when you make any changes to your model. You wouldn't want to wipe your database after any and each update of your application, would you?

In the best of cases such an encoding wouldn't be very efficient; in some worse scenarios your team might not have thought such details though and you get stuck into a complex migration on your live data.

Just like when using Hibernate ORM with a relational database, data stored over Hibernate OGM is easy to recover even using other tools as it's encoded using a well defined Protobuf schema.

Being able to "map" new domain objects by simply adding a couple of annotations is going to make you more productive than re-inventing such error-prone encoding techniques, or figuring out how to best store object graphs and relations into {brandname}.

Finally, using Hibernate OGM allows you to use all existing framework integration points, such as injecting an `EntityManager` as usual: it's not yet another tool but it's the real Hibernate, so inheriting all well known integrations: this will work in Java EE, Spring, Grails, Jhipster, ... and all other technologies integrating with Hibernate.

It's booted like any Hibernate instance: compared to using it with an RDBMS you just have to change some configuration properties, and of course omit the `DataSource` as {brandname} won't use one.

For more details, check the [Hibernate OGM project](#) and the [Hibernate OGM / {brandname}](#) section of the documentation.

24.7. Using {brandname} with Spring

{brandname} integrates with the Spring Framework to make it easy to add caching capabilities to your applications.

24.7.1. Spring Boot Starter

Check out the {brandname} [Spring Boot Starter on GitHub](#) to quickly get up and running.

24.7.2. Setting Up {brandname} as a Spring Cache Provider

{brandname} implements the Spring SPI to offer high-performance, in-memory caching capabilities.

Adding Spring Cache Support

The [Spring Framework](#) offers a [cache abstraction](#) with two simple annotations:

- `@Cacheable` adds entries to the cache.
- `@CacheEvict` removes entries from the cache.

To add caching support to your application, do the following:

1. Enable cache annotations in your application context either declaratively or programmatically.
 - **Declaratively:** Add `<cache:annotation-driven/>` to your application context.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:cache="http://www.springframework.org/schema/cache"
  xmlns:p="http://www.springframework.org/schema/p"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/cache
    http://www.springframework.org/schema/cache/spring-cache.xsd">

  <cache:annotation-driven />

</beans>
```

- **Programmatically:** Enable cache support as follows:

```
@EnableCaching @Configuration
public class Config {
}
```

2. Add {brandname} and the Spring integration module to your `pom.xml`.

- Embedded mode: `infinispan-spring5-embedded`
- Remote client-server mode: `infinispan-spring5-remote`

The following is an example with embedded mode:

```
<dependencies>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-spring5-embedded</artifactId>
    <version>${version.infinispan}</version>
  </dependency>
  <!-- Tip: Use the Spring Boot starter
  instead of the spring-boot artifact. -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${version.spring}</version>
  </dependency>
</dependencies>
```

Replace:

- `${version.infinispan}` with the appropriate version of {brandname}.
- `${version.spring}` with the appropriate version of Spring.

Configuring {brandname} as the Spring Cache Provider

The Spring cache provider SPI has two interfaces through which it interacts with {brandname}: `org.springframework.cache.CacheManager` and `org.springframework.cache.Cache`. The `CacheManager` interface acts as a factory for named `Cache` instances.

At runtime Spring looks for a `CacheManager` implementation that has a bean named `cacheManager` in the application context.

You can configure your application context either declaratively or programmatically.

- **Declaratively:**

```
<!-- Infinispan cache manager -->
<bean id="cacheManager"
  class=
  "org.infinispan.spring.embedded.provider.SpringEmbeddedCacheManagerFactoryBean"
  p:configurationFileLocation=
  "classpath:/org/infinispan/spring/embedded/provider/sample/books-infinispan-
  config.xml" />
```

- **Programmatically:**

```

@EnableCaching
@Configuration
public class Config {

    @Bean
    public CacheManager cacheManager() {
        return new SpringEmbeddedCacheManager(infinispanCacheManager());
    }

    private EmbeddedCacheManager infinispanCacheManager() {
        return new DefaultCacheManager();
    }

}

```

24.7.3. Adding Caching to Your Application

Add the `@Cacheable` and `@CacheEvict` annotations to your application code.

Adding Cache Entries

The `@Cacheable` annotation adds returned values to a defined cache.

For instance, you have a data access object (DAO) for books. You want book instances to be cached after they have been loaded from the underlying database using `BookDao#findBook(Integer bookId)`.

Annotate the `findBook(Integer bookId)` method with `@Cacheable` as follows:

```

@Transactional
@Cacheable(value = "books", key = "#bookId")
public Book findBook(Integer bookId) {...}

```

Any `Book` instances returned from `findBook(Integer bookId)` are stored in a cache named `books`, using `bookId` as the key.

Note that `"#bookId"` is an expression in the [Spring Expression Language](#) that evaluates the `bookId` argument.



If your application needs to reference entries in the cache directly, you should include the `key` attribute. Without this attribute, Spring generates a hash from the supplied method arguments to use as the cache key.

Deleting Cache Entries

The `@CacheEvict` annotation deletes entries from a defined cache.

Annotate the `deleteBook(Integer bookId)` method with `@CacheEvict` as follows:

```
// Evict all entries in the "books" cache
@Transactional
@CacheEvict (value="books", key = "#bookId", allEntries = true)
public void deleteBookAllEntries() {...}

// Evict entries in the "books" cache that match #bookId
@Transactional
@CacheEvict (value="books", key = "#bookId")
public void deleteBook(Integer bookId) {...}}
```

24.7.4. Configuring Timeouts for Cache Operations

The {brandname} Spring cache provider defaults to blocking behaviour when performing read and write operations. By default operations are synchronous and do not time out. However, you might want to set a maximum time to wait for operations before timing out in some situations. For example, timeouts are useful if you need to ensure that an operation completes within a certain time and you can ignore the cached value.

`infinispan.spring.operation.read.timeout`

Specifies the time, in milliseconds, to wait for read operations to complete. The default is 0 which means unlimited wait time.

`infinispan.spring.operation.write.timeout`

Specifies the time, in milliseconds, to wait for write operations to complete. The default is 0 which means unlimited wait time.

To configure timeouts for cache operations, set the properties in the context XML for your application on either `SpringEmbeddedCacheManagerFactoryBean` or `SpringRemoteCacheManagerFactoryBean`.



In remote client-server mode, you can also add these properties to `hotrod-client.properties`.

The following example shows the timeout properties in the context XML for `SpringRemoteCacheManagerFactoryBean`:

```
<bean id="springRemoteCacheManagerConfiguredUsingConfigurationProperties"
      class="
org.infinispan.spring.remote.provider.SpringRemoteCacheManagerFactoryBean">
  <property name="configurationProperties">
    <props>
      <prop key="infinispan.spring.operation.read.timeout">500</prop>
      <prop key="infinispan.spring.operation.write.timeout">700</prop>
    </props>
  </property>
</bean>
```

24.7.5. Externalizing Sessions Using Spring Session

[Spring Session](#) lets you externalize user session information into {brandname}.

To configure Spring Session integration in your application, do the following:

1. Add dependencies to your `pom.xml`.
 - Embedded mode: `infinispan-spring5-embedded`
 - Remote client-server mode: `infinispan-spring5-remote`

The following is an example with remote client-server mode:

```
<dependencies>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-core</artifactId>
  </dependency>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-spring5-remote</artifactId>
    <version>${version.infinispan}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${version.spring}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-session-core</artifactId>
    <version>${version.spring}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>${version.spring}</version>
  </dependency>
</dependencies>
```

Replace:

- `${version.infinispan}` with the appropriate version of {brandname}.
- `${version.spring}` with the appropriate version of Spring.
 1. Specify the appropriate FactoryBean to expose a `CacheManager` instance.
- Embedded mode: `SpringEmbeddedCacheManagerFactoryBean`
- Remote client-server mode: `SpringRemoteCacheManagerFactoryBean`

1. Enable Spring Session with the appropriate annotation.
- Embedded mode: `@EnableInfinispanEmbeddedHttpSession`
 - Remote client-server mode: `@EnableInfinispanRemoteHttpSession`

These annotations have optional parameters:

- `maxInactiveIntervalInSeconds` sets session expiration time in seconds. The default is `1800`.
- `cacheName` specifies the name of the cache that stores sessions. The default is `sessions`.

The following example shows a complete, annotation-based configuration:

```
@EnableInfinispanEmbeddedHttpSession
@Configuration
public class Config {

    @Bean
    public SpringEmbeddedCacheManagerFactoryBean springCacheManager() {
        return new SpringEmbeddedCacheManagerFactoryBean();
    }

    //An optional configuration bean responsible for replacing the default
    //cookie that obtains configuration.
    //For more information refer to the Spring Session documentation.
    @Bean
    public HttpSessionStrategy httpSessionStrategy() {
        return new HeaderHttpSessionStrategy();
    }
}
```

24.8. {brandname} modules for WildFly / EAP

As the {brandname} modules shipped with [WildFly / EAP](#) are tailored to its internal usage, it is recommend to install separate modules if you want to use {brandname} in your application that is deployed to WildFly / EAP. By installing these modules, it is possible to deploy user applications without packaging the {brandname} JARs within the deployments (WARs, EARs, etc), thus minimizing their size. Also, there will be no conflict with WildFly / EAP's internal modules since the slot will be different.

24.8.1. Installation

The modules for WildFly / EAP are available in the [downloads](#) section of our site. After extracting the zip, copy the contents of the `modules` directory to the `WILDFLY_HOME/modules` directory, so that for example the {brandname} core module would be under `WILDFLY_HOME/modules/system/add-ons/{moduleprefix}/org/infinispan/core`.

24.8.2. Application Dependencies

If you are using Maven to build your application, mark the {brandname} dependencies as *provided* and configure your artifact archiver to generate the appropriate MANIFEST.MF file:

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-core</artifactId>
    <version>${version.infinispan}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-cache-store-jdbc</artifactId>
    <version>${version.infinispan}</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-war-plugin</artifactId>
      <configuration>
        <archive>
          <manifestEntries>
            <Dependencies>org.infinispan.core:${version.slot} services,
org.infinispan.cache-store-jdbc:${version.slot} services</Dependencies>
          </manifestEntries>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Replace:

- `${version.infinispan}` with the appropriate version of {brandname}.
- `${version.slot}` with the slot version, ispn-10.0.

The next section illustrates the manifest entries for different types of {brandname}'s dependencies.

{brandname} core

In order expose only {brandname} core dependencies to your application, add the follow to the manifest:

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan:ispn-10.0 services
```

Remote

If you need to connect to remote {brandname} servers via Hot Rod, including execution of remote queries, use the module `org.infinispan.remote` that exposes the needed dependencies conveniently:

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan.remote:ispn-10.0 services
```

Embedded Query

For embedded querying, including the {brandname} Query DSL, Lucene and Hibernate Search Queries, add the following:

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan:ispn-10.0 services, org.infinispan.query:ispn-10.0
services
```

Lucene Directory

Lucene users who wants to simple use {brandname} as a `org.apache.lucene.store.Directory` don't need to add the query module, the entry below is sufficient:

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan.lucene-directory:ispn-10.0
```

Hibernate Search directory provider for {brandname}

The Hibernate Search directory provider for {brandname} is also contained within the {brandname} modules zip. It is not necessary to add an entry to the manifest file since the Hibernate Search module already has an optional dependency to it. When choosing the {brandname} module zip to use, start by checking which Hibernate Search is in use, more details below.

Usage with Wildfy's internal Hibernate Search modules

The Hibernate Search module present in Wildfly 10.x has slot "5.5", which in turn has an optional dependency to `org.infinispan.hibernate-search.directory-provider-for-hibernatesearch-5.5`. This

dependency will be available once the {brandname} modules are [installed](#).

Usage with other Hibernate Search modules

The module `org.hibernate.search:ispn-10.0` distributed with {brandname} is to be used together with {brandname} Query only (querying data from caches), and should not be used by Hibernate ORM applications. To use a Hibernate Search with a different version that is present in Wildfly, please consult the [Hibernate Search documentation](#).

Make sure that the chosen Hibernate Search optional slot for `org.infinispan.hibernate-search:directory-provider` matches the one distributed with {brandname}.

24.8.3. Usage

There are two possible ways for your application to utilize {brandname} within Wildfly, embedded mode and server mode.

Embedded Mode

All CacheManagers and cache instances are created in your application logic. The lifecycle of your EmbeddedCacheManager is tightly coupled with your application's lifecycle, resulting in any manager instances created by your application being destroyed with your application.

Server Mode

In server mode, it is possible for cache containers and caches to be created before runtime as part of Wildfly's standalone/domain.xml configuration. This allows cache instances to be shared across multiple applications, with the lifecycle of the underlying cache container being independent of the deployed application.

Configuration

To enable server mode, it is necessary to make the following additions to your wildfly configuration in `standalone/domain.xml`. Note, that only steps 1-4 are required for local cache instances:

1. Add the {brandname} extensions to your `<extensions>` section.

```
<extensions>
  <extension module="org.infinispan.extension:ispn-10.0"/>
  <extension module="org.infinispan.server.endpoint:ispn-10.0"/>
  <extension module="org.jgroups.extension:ispn-10.0"/>

  <!--Other wildfly extensions-->
</extensions>
```

2. Configure the {brandname} subsystem, along with your required containers and caches, in the server profile which requires {brandname}. Note, it's important that the module attribute is defined so that correct {brandname} classes are loaded.


```
<subsystem xmlns="urn:infinispan:server:core:9.4">
  <cache-container module="org.infinispan.extension:ispn-10.0" name=
    "infinispan_container" default-cache="default">
    <transport/>
    <global-state/>
    <distributed-cache name="default"/>
    <distributed-cache name="memcachedCache"/>
    <distributed-cache name="namedCache"/>
  </cache-container>
</subsystem>
```

3. Define the Wildfly [socket-bindings](#) required by the endpoint and/or JGroup subsystems
4. Configure any endpoints that you require via the endpoint subsystem:

```
<subsystem xmlns="urn:infinispan:server:endpoint:9.4">
  <hotrod-connector socket-binding="hotrod" cache-container="infinispan_container">
    <topology-state-transfer lazy-retrieval="false" lock-timeout="1000" replication-
      timeout="5000"/>
  </hotrod-connector>
  <rest-connector socket-binding="rest" cache-container="infinispan_container">
    <authentication security-realm="ApplicationRealm" auth-method="BASIC"/>
  </rest-connector>
</subsystem>
```

5. Define JGroups transport, ensuring that you define the model attribute for all protocols specified.

```

<subsystem xmlns="urn:infinispan:server:jgroups:9.4">
  <channels default="cluster">
    <channel name="cluster" stack="udp"/>
  </channels>
  <stacks>
    <stack name="udp">
      <transport type="UDP" socket-binding="jgroups-udp" module="org.jgroups:ispn-10.0" />
      <protocol type="PING" module="org.jgroups:ispn-10.0" />
      <protocol type="MERGE3" module="org.jgroups:ispn-10.0" />
      <protocol type="FD SOCK" socket-binding="jgroups-udp-fd" module="org.jgroups:ispn-10.0" />
      <protocol type="FD_ALL" module="org.jgroups:ispn-10.0" />
      <protocol type="VERIFY_SUSPECT" module="org.jgroups:ispn-10.0" />
      <protocol type="pbcast.NAKACK2" module="org.jgroups:ispn-10.0" />
      <protocol type="UNICAST3" module="org.jgroups:ispn-10.0" />
      <protocol type="pbcast.STABLE" module="org.jgroups:ispn-10.0" />
      <protocol type="pbcast.GMS" module="org.jgroups:ispn-10.0" />
      <protocol type="UFC" module="org.jgroups:ispn-10.0" />
      <protocol type="MFC" module="org.jgroups:ispn-10.0" />
      <protocol type="FRAG2" module="org.jgroups:ispn-10.0" />
    </stack>
  </stacks>
</subsystem>

```

Accessing Containers and Caches

Once a container has been defined in your server's configuration, it is possible to inject an instance of a `CacheContainer` or `Cache` into your application using the `@Resource` JNDI lookup. A container is accessed using the following string `java:jboss/datagrid-infinispan/container/<container_name>` and similarly a cache is accessed via `java:jboss/datagrid-infinispan/container/<container_name>/cache/<cache_name>`.

The example below shows how to inject the `CacheContainer` called "infinispan_container" and the distributed cache "namedCache" into an application.

```

public class ExampleApplication {
    @Resource(lookup = "java:jboss/datagrid-infinispan/container/infinispan_container")
    CacheContainer container;

    @Resource(lookup = "java:jboss/datagrid-infinispan/container/infinispan_container/cache/namedCache")
    Cache cache;
}

```

24.8.4. Troubleshooting

Enable logging

Enabling trace on `org.jboss.modules` can be useful to debug issues like `LinkageError` and `ClassNotFoundException`. To enable it at runtime using the Wildfly CLI:

```
bin/jboss-cli.sh -c '/subsystem=logging/logger=org.jboss.modules:add'
bin/jboss-cli.sh -c '/subsystem=logging/logger=org.jboss.modules:write-attribute(name=level,value=TRACE)'
```

Print dependency tree

The following command can be used to print all dependencies for a certain module. For example, to obtain the tree for the module `org.infinispan:ispn-10.0`, execute from `WILDFLY_HOME`:

```
java -jar jboss-modules.jar -deptree -mp modules/ "org.infinispan:ispn-10.0"
```

Chapter 25. Grid File System

{brandname}'s GridFileSystem is an experimental API that exposes an {brandname}-backed data grid as a file system.



This is an *experimental* API. Use at your own risk.

Specifically, the API works as an extension to the JDK's [File](#) , [InputStream](#) and [OutputStream](#) classes: specifically, [GridFile](#), [GridInputStream](#) and [GridOutputStream](#). A helper class, [GridFilesystem](#), is also included.

Essentially, the [GridFilesystem](#) is backed by 2 {brandname} caches - one for metadata (typically replicated) and one for the actual data (typically distributed). The former is replicated so that each node has metadata information locally and would not need to make RPC calls to list files, etc. The latter is distributed since this is where the bulk of storage space is used up, and a scalable mechanism is needed here. Files themselves are chunked and each chunk is stored as a cache entry, as a byte array.

Here is a quick code snippet demonstrating usage:

```
Cache<String,byte[]> data = cacheManager.getCache("distributed");
Cache<String,GridFile.Metadata> metadata = cacheManager.getCache("replicated");
GridFilesystem fs = new GridFilesystem(data, metadata);

// Create directories
File file=fs.getFile("/tmp/testfile/stuff");
fs.mkdirs(); // creates directories /tmp/testfile/stuff

// List all files and directories under "/usr/local"
file=fs.getFile("/usr/local");
File[] files=file.listFiles();

// Create a new file
file=fs.getFile("/tmp/testfile/stuff/README.txt");
file.createNewFile();
```

Copying stuff to the grid file system:

```
InputStream in=new FileInputStream("/tmp/my-movies/dvd-image.iso");
OutputStream out=fs.getOutputStream("/grid-movies/dvd-image.iso");
byte[] buffer=new byte[20000];
int len;
while((len=in.read(buffer, 0, buffer.length)) != -1) out.write(buffer, 0, len);
in.close();
out.close();
```

Reading stuff from the grid:

```
InputStream in=in.getInput("/grid-movies/dvd-image.iso");
OutputStream out=new FileOutputStream("/tmp/my-movies/dvd-image.iso");
byte[] buffer=new byte[200000];
int len;
while((len=in.read(buffer, 0, buffer.length)) != -1) out.write(buffer, 0, len);
in.close();
out.close();
```

25.1. WebDAV demo

{brandname} ships with a demo [WebDAV](#) application that makes use of the grid file system APIs. This demo app is packaged as a [WAR](#) file which can be deployed in a servlet container, such as JBoss AS or Tomcat, and exposes the grid as a file system over WebDAV. This could then be mounted as a remote drive on your operating system.

Chapter 26. Cross site replication

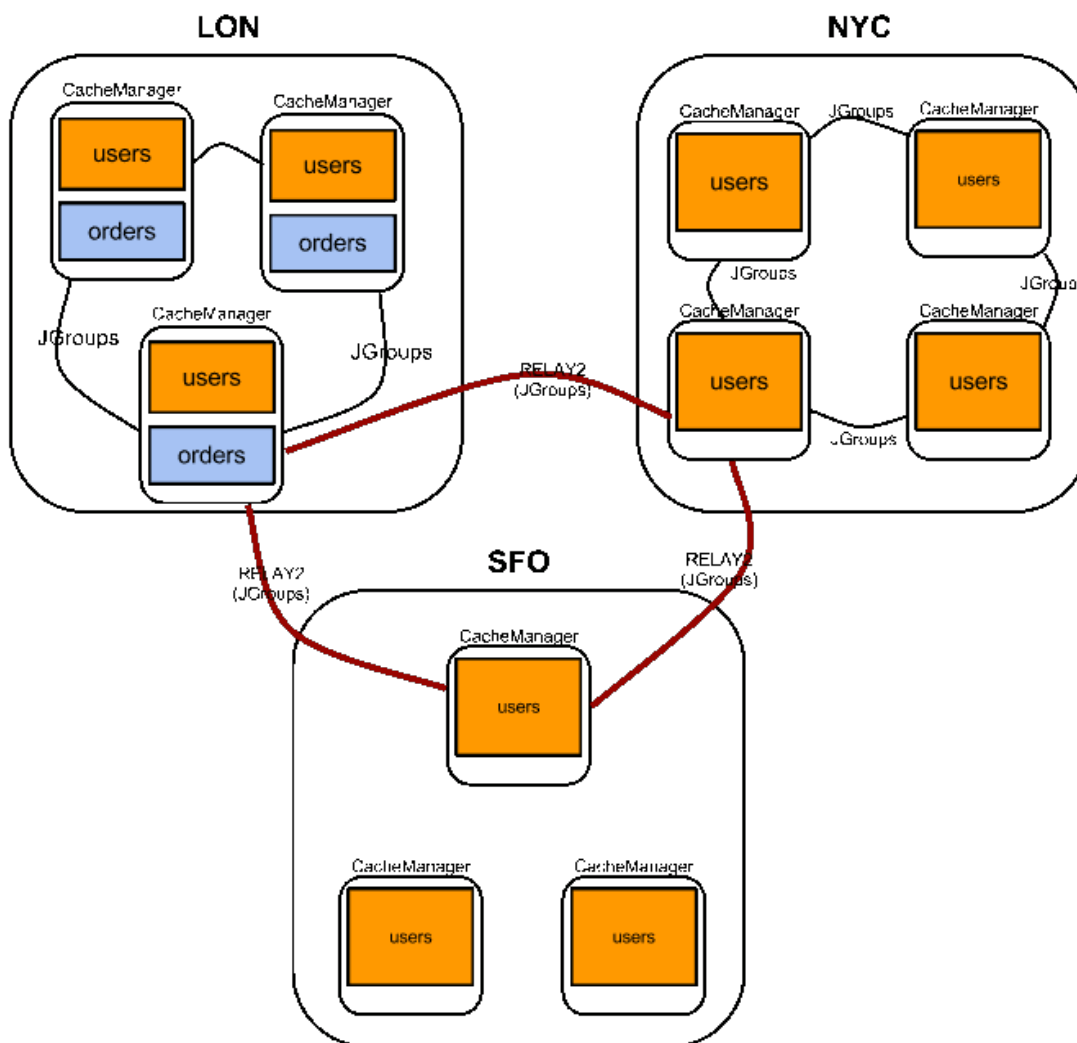
Cross site (x-site) replication allows backing up the data from one cluster to other clusters, potentially situated in different geographical location. The cross-site replication is built on top of JGroups' [RELAY2 protocol](#) . [This document](#) describes the technical design of cross site replication in more detail.



Cross site replication needs the backup cache running in the site master node(s) (i.e. node which receives the backup and applies it). The backup cache starts automatically when it receives the first backup request.

26.1. Sample deployment

The diagram below depicts a possible setup of replicated sites, followed by a description of individual elements present in the deployment. Options are then explained at large in future paragraphs. Comments on the diagram above:



- there are 3 sites: LON, NYC and SFO.
- in each site there is a running {brandname} cluster with a (potentially) different number of

physical nodes: 3 nodes in LON, 4 nodes in NYC and 3 nodes in SFO

- the "users" cache is active in LON, NYC and SFO. Updates on the "users" cache in any of these sites gets replicated to the other sites as well
- it is possible to use different replication mechanisms between sites. E.g. One can configure SFO to backup data synchronously to NYC and asynchronously to LON
- the "users" cache can have a different configuration from one site to the other. E.g. it might be configured as distributed with numOwners=2 in the LON site, REPL in the NYC site and distributed with numOwners=1 in the SFO site
- JGroups is used for both inter-site and intra-site communication. [RELAY2](#) is used for inter-site communication
- "orders" is a site local to LON, i.e. updates to the data in "orders" don't get replicated to the remote sites The following sections discuss specific aspects of cross site replication into more detail. The foundation of the cross-site replication functionality is RELAY2 so it highly recommended to read JGroups' [RELAY2](#) documentation before moving on into cross-site. Configuration

The cross-site replication configuration spreads over the following files:

1. the backup policy for each individual cache is defined in the {brandname} .xml configuration file ([infinispan.xml](#))
2. cluster's JGroups xml configuration file: [RELAY2](#) protocol needs to be added to the JGroups protocol stack ([jgroups.xml](#))
3. RELAY2 configuration file: RELAY2 has an own configuration file ([relay2.xml](#))
4. the JGroups channel that is used by RELAY2 has its own configuration file ([jgroups-relay2.xml](#)) {brandname} XML configuration file

The local site is defined in the the global configuration section. The local is the site where the node using this configuration file resides (in the example above local site is "LON").

infinispan.xml

```
<transport site="LON" />
```

The same setup can be achieved programatically:

```
GlobalConfigurationBuilder lonGc = GlobalConfigurationBuilder.defaultClusteredBuilder  
();  
lonGc.site().localSite("LON");
```

The names of the site (case sensitive) should match the name of a site as defined within JGroups' RELAY2 protocol configuration file. Besides the global configuration, each cache specifies its backup policy in the "site" element:

```
<distributed-cache name="users">
  <backups>
    <backup site="NYC" failure-policy="WARN" strategy="SYNC" timeout="12000"/>
    <backup site="SFO" failure-policy="IGNORE" strategy="ASYNC"/>
    <backup site="LON" strategy="SYNC" enabled="false"/>
  </backups>
</distributed-cache>
```

The "users" cache backs up its data to the "NYC" and "SFO" sites. Even though the "LON" appears as a backup site, it has the "enabled" attribute set to *false* so it will be ignored. For each site backup, the following configuration attributes can be specified:

- *strategy* - the strategy used for backing up data, either "SYNC" or "ASYNC". Defaults to "ASYNC"
- *failure-policy* - Decides what the system would do in case of failure during backup. Possible values are:
 - *IGNORE* - allow the local operation/transaction to succeed
 - *WARN* - same as IGNORE but also logs a warning message. Default.
 - *FAIL* - only in effect if "strategy" is "SYNC" - fails local cluster operation/transaction by throwing an exception to the user
 - *CUSTOM* - user provided, see "failurePolicyClass" below
- *failurePolicyClass* - If the 'failure-policy' is set to 'CUSTOM' then this attribute is required and should contain the fully qualified name of a class implementing `org.infinispan.xsite.CustomFailurePolicy`
- *timeout* - The timeout(milliseconds) to be used when backing up data remotely. Defaults to 10000 (10 seconds)

The same setup can be achieved programmatically:

```
ConfigurationBuilder lon = new ConfigurationBuilder();
lon.sites().addBackup()
    .site("NYC")
    .backupFailurePolicy(BackupFailurePolicy.WARN)
    .strategy(BackupConfiguration.BackupStrategy.SYNC)
    .replicationTimeout(12000)
    .sites().addInUseBackupSite("NYC")
lon.sites().addBackup()
    .site("SFO")
    .backupFailurePolicy(BackupFailurePolicy.IGNORE)
    .strategy(BackupConfiguration.BackupStrategy.ASYNC)
    .sites().addInUseBackupSite("SFO")
```

The "users" cache above doesn't know on which cache on the remote sites its data is being replicated. By default the remote site writes the backup data to a cache having the same name as

the originator, i.e. "users". This behaviour can be overridden with an "backupFor" element. For example the following configuration in SFO makes the "usersLONBackup" cache act as the backup cache for the "users" cache defined above in the LON site:

infinispan.xml

```
<infinispan>
  <cache-container default-cache="">
    <distributed-cache name="usersLONBackup">
      <backup-for remote-cache="users" remote-site="LON"/>
    </distributed-cache>
  </cache-container>
</infinispan>
```

The same setup can be achieved programatically:

```
ConfigurationBuilder cb = new ConfigurationBuilder();
cb.sites().backupFor().remoteCache("users").remoteSite("LON");
```

26.1.1. Local cluster's jgroups .xml configuration

This is the configuration file for the local (intra-site) {brandname} cluster. It is referred from the {brandname} configuration file, see "configurationFile" below:

infinispan.xml

```
<infinispan>
  <jgroups>
    <stack-file name="external-file" path="jgroups.xml"/>
  </jgroups>
  <cache-container>
    <transport stack="external-file" />
  </cache-container>

  ...

</infinispan>
```

In order to allow inter-site calls, the RELAY2 protocol needs to be added to the protocol stack defined in the jgroups configuration (see attached [jgroups.xml](#) for an example).

26.1.2. RELAY2 configuration file

The RELAY2 configuration file is linked from the jgroups.xml (see attached [relay2.xml](#)). It defines the sites seen by this cluster and also the JGroups configuration file that is used by RELAY2 in order to communicate with the remote sites.

26.2. Data replication

For both transactional and non-transactional caches, the backup calls are performed in parallel with local cluster calls, e.g. if we write data to node N1 in LON then replication to the local nodes N2 and N3 and remote backup sites SFO and NYC happen in parallel.

26.2.1. Non transactional caches

In the case of non-transactional caches the replication happens during each operation. Given that data is sent in parallel to backups and local caches, it is possible for the operations to succeed locally and fail remotely, or the other way, causing inconsistencies

26.2.2. Transactional caches

For synchronous transactional caches, {brandname} internally uses a two phase commit protocol: lock acquisition during the 1st phase (prepare) and apply changes during the 2nd phase (commit). For asynchronous caches the two phases are merged, the "apply changes" message being sent asynchronously to the owners of data. This 2PC protocol maps to 2PC received from the JTA transaction manager. For transactional caches, both optimistic and pessimistic, the backup to remote sites happens during the prepare and commit phase only.

Synchronous local cluster with async backup

In this scenario the backup call happens during local commit phase(2nd phase). That means that if the local prepare fails, no remote data is being sent to the remote backup.

Synchronous local cluster with sync backup

In this case there are two backup calls:

- during prepare a message is sent across containing all the modifications that happened within this transaction
- if the remote backup cache is transactional then a transaction is started remotely and all these modifications are being written within this transaction's scope. The transaction is not committed yet (see below)
- if the remote backup cache is not transactional, then the changes are applied remotely
- during the commit/rollback, a commit/rollback message is sent across
- if the remote backups cache is transactional then the transaction started at the previous phase is committed/rolled back
- if the remote backup is not transactional then this call is ignored

Both the local and the backup call(if the "backupFailurePolicy" is set to "FAIL") can veto transaction's prepare outcome

Asynchronous local cluster

In the case of asynchronous local clusters, the backup data is sent during the commit phase. If the

backup call fails and the "backupFailurePolicy" is set to "FAIL" then the user is notified through an exception.

26.3. Taking a site offline

If backing up to a site fails for a certain number of times during a time interval, then it is possible to automatically mark that site as offline. When a site is marked as offline the local site won't try to backup data to it anymore. In order to be taken online a system administrator intervention being required.

26.3.1. Configuration

The taking offline of a site can be configured as follows:

infinispan.xml

```
<replicated-cache name="bestEffortBackup">
  ...
  <backups>
    <backup site="NYC" strategy="SYNC" failure-policy="FAIL">
      <take-offline after-failures="500" min-wait="10000"/>
    </backup>
  </backups>
  ...
</replicated-cache>
```

The *take-offline* element under the *backup* configures the taking offline of a site:

- *after-failures* - the number of failed backup operations after which this site should be taken offline. Defaults to 0 (never). A negative value would mean that the site will be taken offline after *minTimeToWait*
- *min-wait* - the number of milliseconds in which a site is not marked offline even if it is unreachable for 'afterFailures' number of times. If smaller or equal to 0, then only *afterFailures* is considered.

The equivalent programmatic configuration is:

```
lon.sites().addBackup()
    .site("NYC")
    .backupFailurePolicy(BackupFailurePolicy.FAIL)
    .strategy(BackupConfiguration.BackupStrategy.SYNC)
    .takeOffline()
        .afterFailures(500)
        .minTimeToWait(10000);
```

26.3.2. Bringing Sites Back Online

After taking sites offline, invoke the `bringSiteOnline(siteName)` operation via the following JMX MBeans to bring sites back online:

- `XSiteAdmin` enables replication on caches across clusters in a remote site.
- `GlobalXSiteAdminOperations` enables replication on cache containers across clusters in a remote site.

The `bringSiteOnline(siteName)` operation enables replication only and does not do a full update. For this reason, when you bring sites back online, they only contain new entries. You should push transfer to sites after you bring them online to synchronize the most recent data.



Pushing state transfer brings sites back online. You can do that instead of invoking `bringSiteOnline(siteName)`.

26.4. Pushing State Transfer to Sites

Transferring state from one site to another synchronizes the data between the two sites.

You should always transfer state from the currently active site, which contains the most up-to-date data, to another site. The site to which you push state transfer can be online or offline. If you push state transfer to an offline site, it brings that site back online.

Caches on sites to which you transfer state from another site do not need to be empty. However, state transfer only overwrites existing keys on receiving sites and does not delete keys.

For example, key K exists on site A and site B. You transfer state from site A to site B. In this case, {brandname} overwrites key K. However, key Y exists on site B but not site A. After you transfer state from site A to site B, key Y still exists on site B.



You can receive state transfer from only one site at a time. Likewise, if you invoke a state transfer operation on a site, {brandname} ignores subsequent invocations on that site.

Invoke the `pushState(String)` operation via the following JMX MBeans to bring sites back online:

- `XSiteAdminOperations` synchronizes state for caches between the site on which you invoke the operation and a remote site. Brings the site online if it is offline.
- `GlobalXSiteAdminOperations` synchronizes state for cache containers between the site on which you invoke the operation and a remote site. Brings the site online if it is offline.

When you invoke the `pushState(String)` operation on the site that transfers state, you specify the name of the site that receives the state.

The following figure shows the `pushState(String)` operation in JConsole:

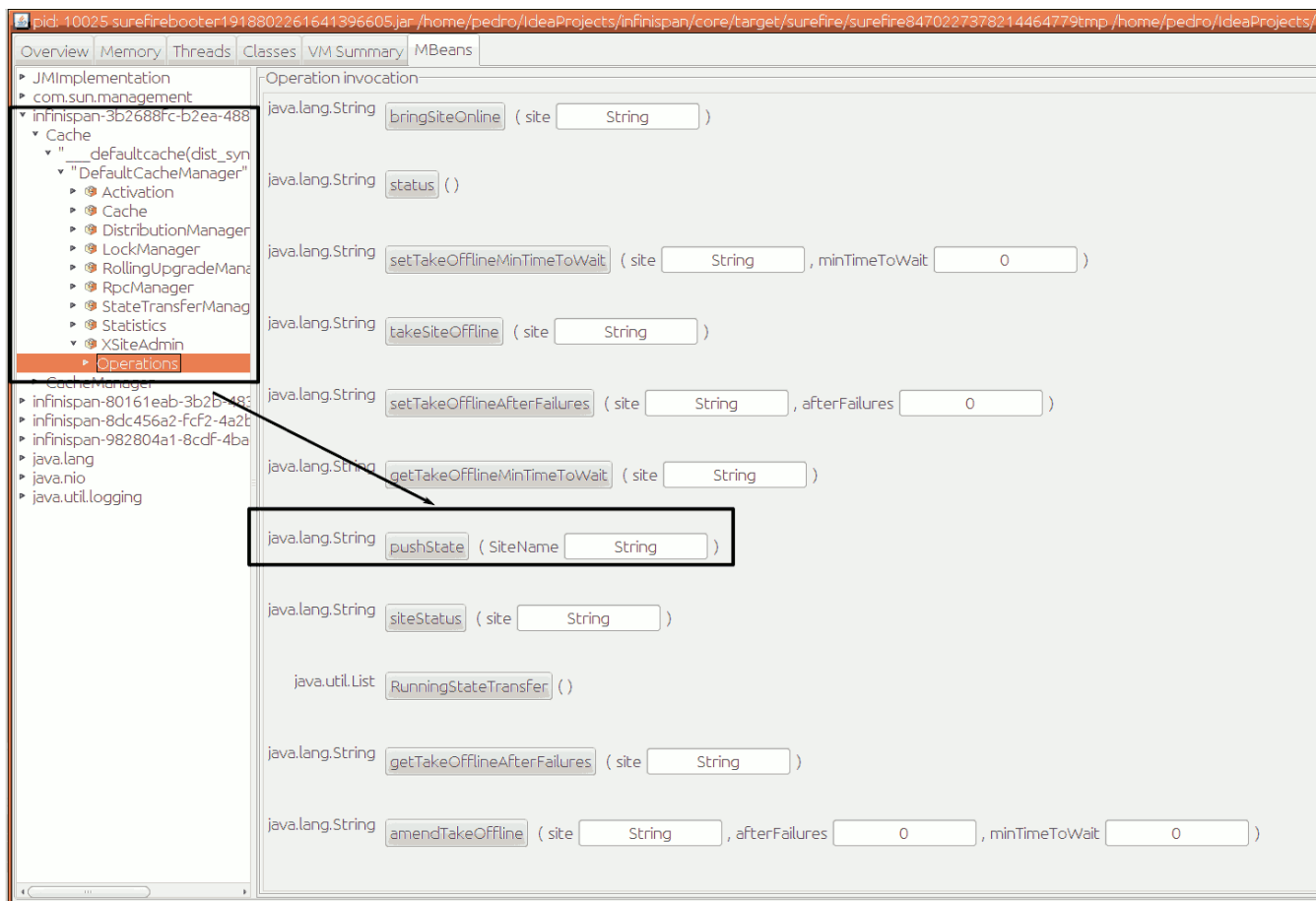


Figure 22. Pushing state via JConsole

26.4.1. Handling join/leave nodes

The current implementation automatically handles the topology changes in producer or consumer site. Also, the cross-site state transfer can run in parallel with a local site state transfer.

26.4.2. Handling broken link between sites

A System Administrator action is needed if the link between the producer and consumer site is broken during the cross-site state transfer (data consistency is not ensured in consumer site). The producer site retries for a while before giving up. Then, it gets back to normal state. However, the consumer site is not able to get back to normal state and, here, an action from System Administrator is needed. The System Administrator should use the operation `cancelReceiveState(String siteName)` to bring the consumer site to normal state.

26.4.3. System Administrator Operations

A set of operations can be performed to control the cross-site state transfer:

- `pushState(String siteName)` - It starts the cross-site state transfer to the site name specified;
- `cancelPushState(String siteName)` - It cancels the cross-site state transfer to the site name specified;
- `getRunningStateTransfer()` - It returns a list of site name to which this site is pushing the state;
- `getSendingSiteName()` - It returns the site name that is pushing state to this site;

- `cancelReceiveState(String siteName)` - It restores the site to normal state. Should be used when the link between the sites is broken during the state transfer (as described above);
- `getPushStateStatus()` - It returns the status of completed cross-site state transfer;
- `clearPushStateStatus()` - It clears the status of completed cross-site state transfer.

For more technical information, you can check the Cross Site design document. See [Reference](#).

26.4.4. Configuration

State transfer between sites cannot be enabled or disabled but it allows to tune some parameters. The values shown below are the default values:

infinispan.xml

```
<replicated-cache name="xSiteStateTransfer">
    ...
    <backups>
        <backup site="NYC" strategy="SYNC" failure-policy="FAIL">
            <state-transfer chunk-size="512" timeout="1200000" max-retries="30" wait-
time="2000" />
        </backup>
    </backups>
    ...
</replicated-cache>
```

The equivalent programmatic configuration is:

```
lon.sites().addBackup()
    .site("NYC")
    .backupFailurePolicy(BackupFailurePolicy.FAIL)
    .strategy(BackupConfiguration.BackupStrategy.SYNC)
    .stateTransfer()
        .chunkSize(512)
        .timeout(1200000)
        .maxRetries(30)
        .waitingTimeBetweenRetries(2000);
```

Below, it is the parameters description:

- *chunk-size* - The number of keys to batch before sending them to the consumer site. A negative or a zero value is **not** a valid value. Default value is 512 keys.
- *timeout* - The time (in milliseconds) to wait for the consumer site acknowledge the reception and appliance of a state chunk. A negative or zero value is **not** a valid value. Default value is 20 minutes.
- *max-retries* - The maximum number of retries when a push state command fails. A negative or a zero value means that the command will not retry in case of failure. Default value is 30.

- *wait-time* - The waiting time (in milliseconds) between each retry. A negative or a zero value is **not** a valid value. Default value is 2 seconds.

26.5. Reference

[This document](#) describes the technical design of cross site replication in more detail.

Chapter 27. Performing Rolling Upgrades

Upgrade {brandname} without downtime or data loss. You can perform rolling upgrades in Remote Client/Server Mode to start using a more recent version of {brandname}.



This section explains how to upgrade {brandname} servers, see the appropriate documentation for your Hot Rod client for upgrade procedures.

From a high-level, you do the following to perform rolling upgrades:

1. Set up a target cluster. The target cluster is the {brandname} version to which you want to migrate data. The source cluster is the {brandname} deployment that is currently in use. After the target cluster is running, you configure all clients to point to it instead of the source cluster.
2. Synchronize data from the source cluster to the target cluster.

27.1. Setting Up a Target Cluster

1. Start the target cluster with unique network properties or a different JGroups cluster name to keep it separate from the source cluster.
2. Configure a `RemoteCacheStore` on the target cluster for each cache you want to migrate from the source cluster.

`RemoteCacheStore` settings

- `remote-server` must point to the source cluster via the `outbound-socket-binding` property.
- `remoteCacheName` must match the cache name on the source cluster.
- `hotrod-wrapping` must be `true` (enabled).
- `shared` must be `true` (enabled).
- `purge` must be `false` (disabled).
- `passivation` must be `false` (disabled).
- `protocol-version` matches the Hot Rod protocol version of the source cluster.

Example RemoteCacheStore Configuration

```
<distributed-cache>
  <remote-store cache="MyCache" socket-timeout="60000" tcp-no-delay="true"
protocol-version="2.5" shared="true" hotrod-wrapping="true" purge="false"
passivation="false">
    <remote-server outbound-socket-binding="remote-store-hotrod-server"/>
  </remote-store>
</distributed-cache>

...
<socket-binding-group name="standard-sockets" default-interface="public"
port-offset="{jboss.socket.binding.port-offset:0}">
  ...
  <outbound-socket-binding name="remote-store-hotrod-server">
    <remote-destination host="198.51.100.0" port="11222"/>
  </outbound-socket-binding>
  ...
</socket-binding-group>
```

3. Configure the target cluster to handle all client requests instead of the source cluster:
 - a. Configure all clients to point to the target cluster instead of the source cluster.
 - b. Restart each client node.

The target cluster lazily loads data from the source cluster on demand via `RemoteCacheStore`.

27.2. Synchronizing Data from the Source Cluster

1. Call the `synchronizeData()` method in the `TargetMigrator` interface. Do one of the following on the target cluster for each cache that you want to migrate:

JMX

Invoke the `synchronizeData` operation and specify the `hotrod` parameter on the `RollingUpgradeManager` MBean.

CLI

```
$ JDG_HOME/bin/cli.sh --connect controller=127.0.0.1:9990 -c
"/subsystem=datagrid-infinispan/cache-container=clustered/distributed-
cache=MyCache:synchronize-data(migrator-name=hotrod)"
```

Data migrates to all nodes in the target cluster in parallel, with each node receiving a subset of the data.

Use the following parameters to tune the operation:

- `read-batch` configures the number of entries to read from the source cluster at a time. The default value is `10000`.

- `write-threads` configures the number of threads used to write data. The default value is the number of processors available.

For example:

```
synchronize-data(migrator-name=hotrod, read-batch=100000, write-threads=3)
```

2. Disable the `RemoteCacheStore` on the target cluster. Do one of the following:

JMX

Invoke the `disconnectSource` operation and specify the `hotrod` parameter on the `RollingUpgradeManager` MBean.

CLI

```
$ JDG_HOME/bin/cli.sh --connect controller=127.0.0.1:9990 -c  
"/subsystem=datagrid-infinispan/cache-container=clustered/distributed-  
cache=MyCache:disconnect-source(migrator-name=hotrod)"
```

3. Decommission the source cluster.

Chapter 28. Extending {brandname}

{brandname} can be extended to provide the ability for an end user to add additional configurations, operations and components outside of the scope of the ones normally provided by {brandname}.

28.1. Custom Commands

{brandname} makes use of a [command/visitor pattern](#) to implement the various top-level methods you see on the public-facing API. This is explained in further detail in the [Architectural Overview](#) section. While the core commands - and their corresponding visitors - are hard-coded as a part of {brandname}'s core module, module authors can extend and enhance {brandname} by creating new custom commands.

As a module author (such as [infinispan-query](#), etc.) you can define your own commands.

You do so by:

1. Create a `META-INF/services/org.infinispan.commands.module.ModuleCommandExtensions` file and ensure this is packaged in your jar.
2. Implementing `ModuleCommandFactory`, `ModuleCommandInitializer` and `ModuleCommandExtensions`
3. Specifying the fully-qualified class name of the `ModuleCommandExtensions` implementation in `META-INF/services/org.infinispan.commands.module.ModuleCommandExtensions`.
4. Implement your custom commands and visitors for these commands

28.1.1. An Example

Here is an example of an `META-INF/services/org.infinispan.commands.module.ModuleCommandExtensions` file, configured accordingly:

org.infinispan.commands.module.ModuleCommandExtensions

```
org.infinispan.query.QueryModuleCommandExtensions
```

For a full, working example of a sample module that makes use of custom commands and visitors, check out [{brandname} Sample Module](#).

28.1.2. Preassigned Custom Command Id Ranges

This is the list of `Command` identifiers that are used by {brandname} based modules or frameworks. {brandname} users should avoid using ids within these ranges. (RANGES to be finalised yet!) Being this a single byte, ranges can't be too large.

{brandname} Query:	100 - 119
Hibernate Search:	120 - 139
Hot Rod Server:	140 - 141

28.2. Extending the configuration builders and parsers

If your custom module requires configuration, it is possible to enhance {brandname}'s configuration builders and parsers. Look at the [custom module tests](#) for a detail example on how to implement this.

Chapter 29. Architectural Overview

This section contains a high level overview of {brandname}'s internal architecture. This document is geared towards people with an interest in extending or enhancing {brandname}, or just curious about {brandname}'s internals.

29.1. Cache hierarchy

{brandname}'s Cache interface extends the JRE's ConcurrentMap interface which provides for a familiar and easy-to-use API.

```
---
public interface Cache<K, V> extends BasicCache<K, V> {
    ...
}
```

```
public interface BasicCache<K, V> extends ConcurrentMap<K, V> { ... } ---
```

Caches are created by using a CacheContainer instance - either the EmbeddedCacheManager or a RemoteCacheManager. In addition to their capabilities as a factory for Caches, CacheContainers also act as a registry for looking up Caches.

EmbeddedCacheManagers create either clustered or standalone Caches that reside in the same JVM. RemoteCacheManagers, on the other hand, create RemoteCaches that connect to a remote cache tier via the Hot Rod protocol.

29.2. Commands

Internally, each and every cache operation is encapsulated by a command. These command objects represent the type of operation being performed, and also hold references to necessary parameters. The actual logic of a given command, for example a ReplaceCommand, is encapsulated in the command's perform() method. Very object-oriented and easy to test.

All of these commands implement the VisitableCommand interface which allow a Visitor (described in next section) to process them accordingly.

```
---
public class PutKeyValueCommand extends VisitableCommand {
    ...
}
```

```
public class GetKeyValueCommand extends VisitableCommand { ... }
```

i. etc ... ---

29.3. Visitors

Commands are processed by the various Visitors. The visitor interface, displayed below, exposes methods to visit each of the different types of commands in the system. This gives us a type-safe mechanism for adding behaviour to a call. Commands are processed by `Visitor`s. The visitor interface, displayed below, exposes methods to visit each of the different types of commands in the system. This gives us a type-safe mechanism for adding behaviour to a call.

```
---
public interface Visitor {
    Object visitPutKeyValueCommand(InvocationContext ctx, PutKeyValueCommand command)
    throws Throwable;
```

```
Object visitRemoveCommand(InvocationContext ctx, RemoveCommand command) throws
Throwable;
```

```
Object visitReplaceCommand(InvocationContext ctx, ReplaceCommand command) throws
Throwable;
```

```
Object visitClearCommand(InvocationContext ctx, ClearCommand command) throws
Throwable;
```

```
Object visitPutMapCommand(InvocationContext ctx, PutMapCommand command) throws
Throwable;
```

```
i. etc ... } ---
```

An `AbstractVisitor` class in the `org.infinispan.commands` package is provided with no-op implementations of each of these methods. Real implementations then only need override the visitor methods for the commands that interest them, allowing for very concise, readable and testable visitor implementations.

29.4. Interceptors

Interceptors are special types of Visitors, which are capable of visiting commands, but also acts in a chain. A chain of interceptors all visit the command, one in turn, until all registered interceptors visit the command.

The class to note is the `CommandInterceptor`. This abstract class implements the interceptor pattern, and also implements Visitor. {brandname}'s interceptors extend `CommandInterceptor`, and these add specific behaviour to specific commands, such as distribution across a network or writing through to disk.

There is also an experimental asynchronous interceptor which can be used. The interface used for asynchronous interceptors is [AsyncInterceptor](#) and a base implementation which should be used when a custom implementation is desired [BaseCustomAsyncInterceptor](#). Note this class also implements the [Visitor](#) interface.

29.5. Putting it all together

So how does this all come together? Invocations on the cache cause the cache to first create an invocation context for the call. Invocation contexts contain, among other things, transactional characteristics of the call. The cache then creates a command for the call, making use of a command factory which initialises the command instance with parameters and references to other subsystems.

The cache then passes the invocation context and command to the `InterceptorChain`, which calls each and every registered interceptor in turn to visit the command, adding behaviour to the call. Finally, the command's `perform()` method is invoked and the return value, if any, is propagated back to the caller.

29.6. Subsystem Managers

The interceptors act as simple interception points and don't contain a lot of logic themselves. Most behavioural logic is encapsulated as managers in various subsystems, a small subset of which are:

29.6.1. DistributionManager

Manager that controls how entries are distributed across the cluster.

29.6.2. TransactionManager

Manager than handles transactions, usually supplied by a third party.

29.6.3. RpcManager

Manager that handles replicating commands between nodes in the cluster.

29.6.4. LockManager

Manager that handles locking keys when operations require them.

29.6.5. PersistenceManager

Manager that handles persisting data to any configured cache stores.

29.6.6. DataContainer

Container that holds the actual in memory entries.

29.6.7. Configuration

A component detailing all of the configuration in this cache.

29.7. ComponentRegistry

A registry where the various managers above and components are created and stored for use in the cache. All of the other managers and crucial components are accessible through the registry.

The registry itself is a lightweight dependency injection framework, allowing components and managers to reference and initialise one another. Here is an example of a component declaring a dependency on a `DataContainer` and a `Configuration`, and a `DataContainerFactory` declaring its ability to construct `DataContainers` on the fly.

```
---
@Inject
public void injectDependencies(DataContainer container, Configuration
configuration) {
    this.container = container;
    this.configuration = configuration;
}
```

```
@DefaultFactoryFor
public class DataContainerFactory extends AbstractNamedCacheComponentFactory {
---
```

Components registered with the `ComponentRegistry` may also have a lifecycle, and methods annotated with `@Start` or `@Stop` will be invoked before and after they are used by the component registry.

```
---
@Start
public void init() {
    useWriteSkewCheck = configuration.locking().writeSkewCheck();
}
```

```
@Stop(priority=20)
public void stop() {
    notifier.removeListener(listener);
    executor.shutdownNow();
}
---
```

In the example above, the optional `priority` parameter to `@Stop` is used to indicate the order in which the component is stopped, in relation to other components. This follows a Unix Sys-V style

ordering, where smaller priority methods are called before higher priority ones. The default priority, if not specified, is 10.