Integrating {brandname} 10.0

Table of Contents

Chapter 1. Integrations

{brandname} can be integrated with a number of other projects, as detailed below.

1.1. 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.

1.2. 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.

1.3. 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.

1.3.1. Lucene compatibility

Apache Lucene versions 5.5.x

1.3.2. Maven dependencies

All you need is the following:

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-lucene-directory</artifactId>
  <!-- Replace ${version.infinispan} with the
  version of {brandname} that you're using. -->
  <version>${version.infinispan}</version>
</dependency>
```

1.3.3. 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 defaultCacheManager = new DefaultCacheManager();
Cache metadataCache = defaultCacheManager.getCache("metadataCache");
Cache dataCache = defaultCacheManager.getCache("dataCache");
Cache lockCache = defaultCacheManager.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.

1.3.4. 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>
```

1.3.5. 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* .

1.3.6. 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">
```

1.3.7. 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
location	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} Directory constructor.	none (mandatory)
autoChunkSize	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.

1.3.8. 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
- You 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.

1.3.9. 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.

1.3.10. 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.

1.3.11. 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

1.3.12. 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.

1.3.13. Maven dependencies

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-directory-provider</artifactId>
  <!-- Replace ${version.infinispan} with the
  version of {brandname} that you're using. -->
  <version>${version.infinispan}</version>
</dependency>
```

1.3.14. 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.

1.3.15. Configuration

Optional properties allow for a custom {brandname} configuration or to use an existent *EmbeddedCacheManager*:

Property	Description	Example value
`hibernate.search.infinispan.configuration_resourcename`	Custom configuration for {brandname}	config/infinispan.xml
`hibernate.search.infinispan.configuration.transport_override_resourcename`	Overrides the JGroups stack in the {brandname} configuration file	jgroups-ec2.xml
`hibernate.search.infinispan.cachemanager_jndiname`	Specifies the JNDI name under which the *EmbeddedCacheManager* to use is bound. Will cause the properties above to be ignored when present	`java:jboss/infinispan/container/hibernate-search`

1.3.16. 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.

Chapter 2. 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>
  <!-- Replace ${version.infinispan} with the
  version of {brandname} that you're using. -->
  <version>${version.infinispan}</version>
</dependency>
```

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>
  <!-- Replace ${version.infinispan} with a
  9.2.x version of {brandname}. -->
  <version>${version.infinispan}</version>
</dependency>
```

Hibernate 5.1

Use the following Maven coordinates:

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-hibernate-cache-v51</artifactId>
  <!-- Replace ${version.infinispan} with the
  version of {brandname} that you're using. -->
  <version>${version.infinispan}</version>
</dependency>
```

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 1. Enable second-level cache

JPA	`<property name="hibernate.cache.use_second_level_cache" value="true"/>`
Spring	`spring.jpa.properties.hibernate.cache.use_second_level_cache=true`

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 2. Enable selective shared cached mode

JPA	`<shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>`
Spring	`spring.jpa.properties.javax.persistence.sharedCache.mode=ENABLE_SELECTIVE`

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 3. Enable query cache

JPA	`<property name="hibernate.cache.use_query_cache" value="true"/>`
Spring	`spring.jpa.properties.hibernate.cache.use_query_cache=true`

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 4. Enable statistics

JPA	`<property name="hibernate.generate_statistics" value="true" />`
Spring	`spring.jpa.properties.hibernate.generate_statistics=true`

2.1. Deployment Scenarios

How to configure {brandname} to be the second level cache provider varies slightly depending on the deployment scenario:

2.1.1. 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.

2.1.2. 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.

2.1.3. 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.

2.1.4. 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.

2.1.5. 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.

2.1.6. 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 `clustered.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.

2.2. Configuration Reference

This section is dedicated at explaining configuration in detail as well as some extra configuration options.

2.2.1. 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.

2.2.2. 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:

Entities and Collections

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.

2.2.3. 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.

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`:

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.someCol
lection.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
ifespan"
   value= "65000"/>
```

2.3. 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 <code>read-write</code> and <code>transactional</code> use the same implementation. {brandname} provides the same consistency guarantees for both <code>transactional</code> and <code>read-write</code> 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 5. Cache concurrency strategy/cache mode compatibility table

Concurrency strategy	Cache transactions	Cache mode	Eviction
transactional	≤ 5.2 transactional	invalidation	yes
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.

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
```

2.4. 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.

2.5. 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.

Chapter 3. Using {brandname} with Spring

{brandname} integrates with the Spring Framework to make it easy to add caching capabilities to your applications.

3.1. Spring Boot Starter

Check out the {brandname} Spring Boot Starter on GitHub to quickly get up and running.

3.2. Setting Up {brandname} as a Spring Cache Provider

{brandname} implements the Spring SPI to offer high-performance, in-memory caching capabilities.

3.2.1. 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>
        <!-- Replace ${version.infinispan} with the
        version of {brandname} that you're using. -->
        <version>${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>
        <!-- Replace ${version.spring} with your Spring version. -->
        <version>${version.spring}</version>
    </dependency>
</dependencies>
```

3.2.2. 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();
   }

}
```

3.3. Adding Caching to Your Application

Add the `@Cacheable` and `@CacheEvict` annotations to your application code.

3.3.1. 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.

3.3.2. Deleting Cache Entries

The `@CacheEvict` annotation deletes entries from a defined cache.

Annotate the `deleteBook(Integer bookId)` method with `@CacheEvict` as follows:

```
Unresolved directive in ../../topics/integrations_spring.adoc - include::
code_examples/ SpringCacheEvictExample.java[]
```

3.4. 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>
```

3.5. 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:

```xml
<dependencies>
    <dependency>
        <groupId>org.infinispan</groupId>
        <artifactId>infinispan-core</artifactId>
    </dependency>
    <dependency>
        <groupId>org.infinispan</groupId>
        <artifactId>infinispan-spring5-remote</artifactId>
        <!-- Replace ${version.infinispan} with the
        version of {brandname} that you're using. -->
        <version>${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>
        <!-- Replace ${version.spring} with your Spring version. -->
        <version>${version.spring}</version>
   </dependency>
</dependencies>
```

2. Specify the appropriate FactoryBean to expose a `CacheManager` instance.
   - Embedded mode: `SpringEmbeddedCacheManagerFactoryBean`
   - Remote client-server mode: `SpringRemoteCacheManagerFactoryBean`
3. 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();
   }
}
```

3.6. {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.

3.6.1. 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`.

3.6.2. 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>
    <!-- Replace ${version.infinispan} with the
    version of {brandname} that you're using. -->
    <version>${version.infinispan}</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-cachestore-jdbc</artifactId>
    <!-- Replace ${version.infinispan} with the
    version of {brandname} that you're using. -->
    <version>${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:ispn-10.0 services,
org.infinispan.cachestore.jdbc:ispn-10.0 services</Dependencies>
           </manifestEntries>
         </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

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 {wildflybrandname} Internal Hibernate Search Modules

The Hibernate Search module present in {wildflybrandname} 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 {wildflybrandname}, 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}.

Usage

There are two possible ways for your application to utilize {brandname} within {wildflybrandname}, 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 {wildflybrandname}'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, make the following additions to your {wildflybrandname} configuration in `standalone/domain.xml`:

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 and your required containers and caches in the server profile that requires {brandname}.

Be sure that you define the `module` attribute to load the correct {brandname} classes.

```
<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 {wildflybrandname} 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 the JGroups transport, ensuring that you define the `model` attribute for each protocol.

You do not neet to define a JGroups transport for local cache configurations. The JGroups transport is required for clustered cache configurations only.

```
<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_NB" module="org.jgroups:ispn-10.0"/>
     <protocol type="MFC_NB" 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;
}
```

3.6.3. 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 {wildflybrandname} 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"
```