Integrating Infinispan 12.1

Table of Contents

Find out how to integrate Infinispan with other projects.

Chapter 1. Infinispan Modules for WildFly

To use Infinispan inside applications deployed to WildFly, you should install Infinispan modules that:

- Let you deploy applications without packaging Infinispan JAR files in your WAR or EAR file.
- Allow you to use a Infinispan version that is independent to the one bundled with WildFly.

Infinispan modules are deprecated and planned for removal. These modules provide a temporary solution until WildFly directly manages the `infinispan` subsystem.

1.1. Installing Infinispan Modules

Download and install Infinispan modules for WildFly.

Prerequisites

1. JDK 8 or later.
2. An existing WildFly installation.

Procedure

1. Download the ZIP archive for the modules from the Infinispan software downloads.
2. Extract the ZIP archive and copy the contents of `modules` to the `modules` directory of your WildFly installation so that you get the resulting structure:

```
$WILDFLY_HOME/modules/system/add-ons/{moduleprefix}/org/infinispan/ispn-12.1
```

1.2. Configuring Applications to Use Infinispan Modules

After you install Infinispan modules for WildFly, configure your application to use Infinispan functionality.

Procedure

1. In your project `pom.xml` file, mark the required Infinispan dependencies as *provided*.
2. Configure your artifact archiver to generate the appropriate `MANIFEST.MF` file.

pom.xml

```
<dependencies>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-core</artifactId>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-cachestore-jdbc</artifactId>
    <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:ispn-12.1 services</Dependencies>
           </manifestEntries>
         </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Infinispan functionality is packaged as a single module, `org.infinispan`, that you can add as an entry to your application's manifest as follows:

MANIFEST.MF

```
Manifest-Version: 1.0
Dependencies: org.infinispan:ispn-12.1 services
```

AWS dependencies

If you require AWS dependencies, such as S3_PING, add the following module to your application's manifest:

```
Manifest-Version: 1.0
Dependencies: com.amazonaws.aws-java-sdk:ispn-12.1 services
```

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 Infinispan implements. This enables Infinispan 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 Infinispan as second-level cache provider with Hibernate.

Applications running in environments where Infinispan is not default cache provider for Hibernate will need to depend on the correct cache provider version.

The Infinispan 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>
```

Hibernate 5.2

Hibernate 5.2 is supported in Infinispan 9.2.x only.

Use the following Maven coordinates:

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-hibernate-cache</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Hibernate 5.1

Use the following Maven coordinates:

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-hibernate-cache-v51</artifactId>
</dependency>
```

Hibernate version 5.0 and earlier: the Infinispan cache provider is shipped by Hibernate. Documentation and Maven coordinates are located in the Hibernate documentation.

Apart from Infinispan 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 Infinispan 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 Infinispan integration.

Enabling Infinispan second level cache provider inside a JPA/Hibernate application that runs in single node is very straightforward. First, make sure the Hibernate Infinispan 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 Infinispan second-level cache provider uses an Infinispan configuration that's designed for clustered environments. However, Infinispan 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 Infinispan 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 Infinispan as second-level cache provider.

As in the previous case, start by making sure that Hibernate Infinispan 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 Infinispan second-level cache provider uses an Infinispan configuration that's designed for clustered environments. However, Infinispan 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 Infinispan as Hibernate cache provider in a Spring application can be found here.

2.1.3. Single-Node WildFly Application

In WildFly, Infinispan 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`. Infinispan'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 Infinispan 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 Infinispan 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 Infinispan 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 Infinispan second-level cache, it is necessary to cluster the second-level cache so that cache consistency can be guaranteed. Clustering the Infinispan 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 Infinispan 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 Infinispan as second level cache, the cache needs to be clustered. Clustering the Infinispan 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 Infinispan 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, Infinispan 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 Infinispan 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 Infinispan 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 Infinispan 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

Infinispan 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

Infinispan 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 Infinispan 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, Infinispan 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 Infinispan configuration

```
<property
    name="hibernate.cache.infinispan.cfg"
    value="my-infinispan-configuration.xml" />
```

If the cache is configured as transactional, Infinispan cache provider automatically sets transaction manager so that the TM used by Infinispan 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 Infinispan 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 Infinispan 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

Infinispan 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. Infinispan 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 Infinispan 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 Infinispan 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.

Chapter 3. Using Infinispan with Spring

Infinispan integrates with the Spring Framework to make it easy to add caching capabilities to your applications.

3.1. Setting Up Infinispan as a Spring Cache Provider

Infinispan implements the Spring SPI to offer high-performance, in-memory caching capabilities.

3.1.1. 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 Infinispan 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>
    </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>
```

3.1.2. Configuring Infinispan as the Spring Cache Provider

The Spring cache provider SPI has two interfaces through which it interacts with Infinispan: `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();
   }

}
```

3.2. Adding Caching to Your Application

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

3.2.1. 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.

3.2.2. 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) {...]}
```

3.3. Configuring Timeouts for Cache Operations

The Infinispan 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>
```

3.4. Externalizing Sessions Using Spring Session

Spring Session lets you externalize user session information into Infinispan.

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>
    </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>
```

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();
   }
}
```