HIBERNATE - Relational Persistence for Idiomatic Java

Using JBoss Cache 2 as a Hibernate Second Level Cache

3.3.0.alpha1

HIBERNATE - Relational Persistence for Idiomatic Java

Copyright © 2008 Red Hat Middleware, LLC.

Legal Notice

1801 Varsity Drive
Raleigh, NC27606-2072USA
Phone: +1 919 754 3700
Phone: 888 733 4281
Fax: +1 919 754 3701
PO Box 13588Research Triangle Park, NC27709USA

Copyright © 2008 by Red Hat, Inc. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version is presently available at http://www.opencontent.org/openpub/).

Distribution of substantively modified versions of this document is prohibited without the explicit permission of the copyright holder.

Distribution of the work or derivative of the work in any standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from the copyright holder.

Red Hat and the Red Hat "Shadow Man" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

All other trademarks referenced herein are the property of their respective owners.

The GPG fingerprint of the security@redhat.com key is:

CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E

Preface v
1. Introduction 1
1.1. Overview 1
1.2. Requirements 1
1.3. Configuration Basics 2
2. Core Concepts 5
2.1. Types of Cached Data 5
2.1.1. Entities 5
2.1.2. Collections 7
2.1.3. Queries 8
2.1.4. Timestamps 9
2.2. Key JBoss Cache Behaviors 10
2.2.1. Replication vs. Invalidation vs. Local Mode 10
2.2.2. Synchronous vs. Asynchronous 11
2.2.3. Locking Scheme 12
2.2.4. Initial State Transfer 12
2.2.5. Cache Eviction 13
2.2.6. Buddy Replication and Cache Loading 13
2.3. Matching JBC Behavior to Types of Data 13
2.3.1. The `RegionFactory` Interface 14
2.3.2. The `CacheManager` API 14
2.3.3. Sharable JGroups Resources 15
2.3.4. Bringing It All Together 16
3. Configuration 19
3.1. Configuring the Hibernate Session Factory 19
3.1.1. Basics 19
3.1.2. Specifying the `RegionFactory` Implementation 20
3.1.3. The `SharedJBossCacheRegionFactory` 20
3.1.4. The `JndiSharedJBossCacheRegionFactory` 21
3.1.5. The `MultiplexedJBossCacheRegionFactory` 21
3.1.6. The `JndiMultiplexedJBossCacheRegionFactory` 23
3.2. Configuring JBoss Cache 23
3.2.1. Configuring a Single Standalone Cache 23
3.2.2. Managing Multiple Caches via a CacheManager 23
3.2.3. JBoss Cache Configuration Details 24
3.2.3.1. CacheMode 24
3.2.3.2. NodeLockingScheme 25
3.2.3.3. JGroups `Channel` Configuration 25
3.2.3.4. Initial State Transfer 25
3.2.3.5. Region-Based Marshalling 26
3.2.3.6. Eviction 26
3.2.4. Standard JBoss Cache Configurations 26
3.3. JGroups Configuration 27

3.3.1. Transport -- UDP vs. TCP ... 28
3.3.2. Standard JGroups Configurations ... 29
4. Cache Eviction ... 31
4.1. Overview ... 31
4.1.1. The Eviction Process ... 31
4.1.2. Eviction Regions ... 32
4.1.3. Eviction Policies ... 32
4.1.3.1. The `LRUPolicy` ... 32
4.1.3.2. The `NullEvictionPolicy` ... 33
4.2. Organization of Data in the Cache ... 33
4.2.1. Region Prefix and Region Name ... 33
4.2.2. Entities ... 34
4.2.3. Collections ... 34
4.2.4. Queries ... 35
4.2.5. Timestamps ... 35
4.3. Example Configuration ... 36
4.4. Best Practices ... 38
5. Architecture ... 41
5.1. Hibernate Interface to the Caching Subsystem ... 41
5.2. Single JBoss Cache Instance Architecture ... 42
5.3. Multiple JBoss Cache Instance Architecture ... 44

Preface

This document is focused on the use of the JBoss Cache [http://labs.jboss.org/jbosscache] clustered transactional caching library as a tool for caching data in a Hibernate-based application.

Working with object-oriented software and a relational database can be cumbersome and time consuming in today's enterprise environments. Hibernate is an object/relational mapping tool for Java environments. The term object/relational mapping (ORM) refers to the technique of mapping a data representation from an object model to a relational data model with a SQL-based schema.

Hibernate not only takes care of the mapping from Java classes to database tables (and from Java data types to SQL data types), but also provides data query and retrieval facilities and can significantly reduce development time otherwise spent with manual data handling in SQL and JDBC.

In any application that works with a relational database, caching data retrieved from the database can potentially improve application performance. Hibernate provides a number of facilities for caching data on the database client side, i.e. in the Java process in which Hibernate is running. The primary facility is the Hibernate `Session` itself, which maintains a transaction-scoped cache of persistent data. If you wish to cache data beyond the scope of a transaction, it is possible to configure a cluster or JVM-level (technically a `SessionFactory`-level) cache on a class-by-class, collection-by-collection and query-by-query basis. This type of cache is referred to as a *Second Level Cache*.

Hibernate provides a pluggable architecture for implementing its Second Level Cache, allowing it to integrate with a number of third-party caching libraries. This document is focused on the use of the JBoss Cache [http://labs.jboss.org/jbosscache] clustered transactional caching library as an implementation of the Second Level Cache. It specifically focuses on JBoss Cache 2.

If you are new to Hibernate and Object/Relational Mapping or even Java, please follow these steps:

1. Read the *Hibernate Reference Documentation*, particularly the *Introduction* and *Architecture* sections.

2. Have a look at the `eg/` directory in the Hibernate distribution, it contains a simple standalone application. Copy your JDBC driver to the `lib/` directory and edit `etc/hibernate.properties`, specifying correct values for your

database. From a command prompt in the distribution directory, type `ant eg` (using Ant), or under Windows, type `build eg`.

3. Use the *Hibernate Reference Documentation* as your primary source of information. Consider reading *Java Persistence with Hibernate* (http://www.manning.com/bauer2) if you need more help with application design or if you prefer a step-by-step tutorial. Also visit http://caveatemptor.hibernate.org and download the example application for Java Persistence with Hibernate.

4. FAQs are answered on the Hibernate website.

5. Third party demos, examples, and tutorials are linked on the Hibernate website.

6. The Community Area on the Hibernate website is a good resource for design patterns and various integration solutions (Tomcat, JBoss AS, Struts, EJB, etc.).

If you are new to the Hibernate Second Level Cache or to JBoss Cache, please follow these steps:

1. Read the *Hibernate Reference Documentation*, particularly the *Second Level Cache* and *Configuration* sections.

2. Read the JBoss Cache User Guide, Core Edition [http://labs.jboss.org/jbosscache/docs/index.html].

3. Use this guide as your primary source of information on the usage of JBoss Cache 2 as a Hibernate Second Level Cache.

If you have questions, use the user forum linked on the Hibernate website. The user forum on the JBoss Cache website is also useful. We also provide a JIRA issue tracking system for bug reports and feature requests. If you are interested in the development of Hibernate, join the developer mailing list. If you are interested in translating this documentation into your language, contact us on the developer mailing list.

Commercial development support, production support, and training for Hibernate is available through Red Hat Inc. (see http://www.hibernate.org/SupportTraining/). Hibernate is a Professional Open Source project and a critical component of the JBoss Enterprise Middleware System (JEMS) suite of products.

Chapter 1. Introduction

1.1. Overview

JBoss Cache is a tree-structured, clustered, transactional cache. It includes support for maintaining cache consistency across multiple cache instances running in a cluster. It integrates with JTA transaction managers, supporting transaction-scoped locking of cache elements and automatic rollback of cache changes upon transaction rollback. It supports both pessimistic and optimistic locking, with the tree-structure of the cache allowing maximum concurrency.

All of these features make JBoss Cache an excellent choice for use as a Hibernate *Second Level Cache*, particularly in a clustered environment. A Hibernate `Session` is a transaction-scoped cache of persistent data -- data accessed via the `Session` is cached in the `Session` for the duration of the current transaction, and then is cleared. A *Second Level Cache* is an optional cluster or JVM-level cache whose contents are maintained beyond the life of a transaction and whose contents can be shared across transactions. Use of a Second Level Cache is configured as part of the configuration of the Hibernate `SessionFactory`. If a Second Level Cache is enabled, caching of an instance of a particular entity class or of results of a particular query can be configured on a class-by-class, collection-by-collection and query-by-query basis. See the *Hibernate Reference Documentation* for more on Second Level Cache basics and how to configure entity classes, collections and queries for caching.

The JBoss Cache Second Level Cache integration supports the `transactional` and `read only` *cache concurrency strategies* discussed in the *Hibernate Reference Documentation*. It supports query caching and is, of course, cluster safe.

1.2. Requirements

Second level caching with JBoss Cache 2 requires the use of JBoss Cache 2.1.0 or later. The core JBoss Cache project is used; the related PojoCache project/library is not needed. The following jars, included with the JBoss Cache distribution, need to be on the classpath:

- jbosscache-core.jar
- commons-logging.jar
- jboss-common-core.jar

- jgroups.jar

JBoss Cache also needs to have the classes in the `javax.transaction` package on the classpath, but those are already included in the Hibernate distribution.

The `hibernate-jbosscache2.jar` that is included with the Hibernate distribution also needs to be on the classpath.

A JBoss Cache configuration file, and usually a JGroups configuration file[1], need to be on the classpath. The `hibernate-jbosscache2.jar` includes standard configuration files in the `org.hibernate.cache.jbc2.builder` package. The `jbc2-configs.xml` file is for JBoss Cache and the `jgroups-stacks.xml` file is for JGroups. See Section 3.2, “Configuring JBoss Cache” and Section 3.3, “JGroups Configuration” for more details on these files. Users can create their own versions and tell the `SessionFactory` to use them; see Section 3.1, “Configuring the Hibernate Session Factory” for details.

1.3. Configuration Basics

The key steps in using JBoss Cache as a second level cache are to:

- Tell Hibernate in your `SessionFactory` configuration that you want to use JBoss Cache 2 as your Second Level Cache implementation:

  ```
  hibernate.cache.region.factory_class=
     org.hibernate.cache.jbc2.MultiplexedJBossCacheRegionFactory
  ```

  There are a number of values that can be provided for the `hibernate.cache.region.factory_class` property, depending on how you want the JBoss Cache integration to work. Based on what factory class you specify, there are additional detail configuration properties you can add to further control that factory. However, simply specifying the `MultiplexedJBossCacheRegionFactory` shown above provides a reasonable set of default values useful for many applications. See Section 3.1.2, “Specifying the `RegionFactory` Implementation” for more details.

  Do not set the legacy `hibernate.cache.provider_class` property when using JBoss Cache 2. That is a legacy property from before Hibernate 3.3's redesign of second level caching internals. It will not work with JBoss Cache 2.

[1] JGroups [http://labs.jboss.org/jgroups] is the group communication library used by JBoss Cache for intra-cluster communication.

- Tell Hibernate you want to enable caching of entities and collections. No need to set this property if you don't:

```
hibernate.cache.use_second_level_cache=true
```

- Tell Hibernate you want to enable caching of query results. No need to set this property if you don't:

```
hibernate.cache.use_query_cache=true
```

- If you have enabled caching of query results, tell Hibernate if you want to suppress costly replication of those results around the cluster. No need to set this property if you want query results replicated:

```
hibernate.cache.region.jbc2.query.localonly=true
```

See Chapter 3, *Configuration* for full details on configuration.

Chapter 2. Core Concepts

This chapter focuses on some of the core concepts underlying how the JBoss Cache-based implementation of the Hibernate Second Level Cache works. There's a fair amount of detail, which certainly doesn't all need to be mastered to use JBoss Cache with Hibernate. But, an understanding of some of the basic concepts here will help a user understand what some of the typical configurations discussed in the next chapter are all about.

If you want to skip the details for now, feel free to jump ahead to Section 2.3.4, “Bringing It All Together”

2.1. Types of Cached Data

The Second Level Cache can cache four different types of data: entities, collections, query results and timestamps. Proper handling of each of the types requires slightly different caching semantics. A major improvement in Hibernate 3.3 is the addition of the `org.hibernate.cache.RegionFactory` API, which allows Hibernate to tell the caching integration layer what type of data is being cached. Based on that knowledge, the cache integration layer can apply the semantics appropriate to that type.

2.1.1. Entities

Entities are the most common type of data cached in the second level cache. Entity caching requires the following semantics in a clustered cache:

- Newly created entities should only be stored on the node on which they are created until the transaction in which they were created commits. Until that transaction commits, the cached entity should only be visible to that transaction. After the transaction commits, cluster-wide the cache should be in a "consistent" state. The cache is consistent if on any node in the cluster, the new entity is either:

  - stored in the cache, with all non-collection fields matching what is in the database.

  - not stored in the cache at all.

  Maintaining cache consistency basically requires that the cluster-wide update messages that inform other nodes of the changes made during a transaction be made *synchronously* as part of the transaction commit process. This means that the transaction thread will block until the changes have been transmitted to all nodes in the cluster, those nodes have updated their internal state to reflect the changes, and have responded to

the originating node telling them of their success (or failure) in doing so. JBoss Cache uses a 2 phase commit protocol, so there will actually be 2 synchronous cluster-wide messages per transaction commit. If any node in the cluster fails in the initial prepare phase of the 2PC, the underlying transaction will be rolled back and in the second phase JBoss Cache will tell the other nodes in the cluster to revert the change.

- For existing entities that are modified in the course of a transaction, the updated entity state should only be stored on the node on which the modification occurred until the transaction commits. Until that transaction commits, the changed entity state should only be visible to that transaction. After the transaction commits, cluster-wide the cache should be in a "consistent" state, as described above.

  Concurrent cache updates of the same entity anywhere in the cluster should not be possible, as Hibernate will acquire an exclusive lock on the database representation of the entity before attempting to update the cache.

- A read of a cached entity holds a transaction scope read lock on the relevant portion of cache. The presence of a read lock held by one transaction should not prevent a concurrent read by another transaction. Whether the presence of that read lock prevents a concurrent write depends on whether the cache is configured for READ_COMMITTED or REPEATABLE_READ semantics and whether the cache is using optimistic locking. READ_COMMITTED will allow a concurrent write to proceed; pessimistic locking with REPEATABLE_READ will cause the write to block until the transaction with the read lock commits. Optimistic locking allows a REPEATABLE_READ semantic without forcing the writing transaction to block.

  A read of a cached entity does not result in any messages to other nodes in the cluster or any cluster-wide read locks.

- The basic operation of storing an entity that has been directly read from the database should have a *fail-fast* semantic. This type of operation is referred to as a *put* and is the most common type of operation. Basically, the rules for handling new entities or entity updates discussed above mean the cache's representation of an entity should always match the database's. So, if a `put` attempt encounters any existing copy of the entity in the cache, it should assume that existing copy is either newer or the same as what it is trying to store, and the `put` attempt should promptly and silently abort, with no impact on any ongoing transactions.

  A `put` operation should not acquire any long-lasting locks on the cache.

If the cache is configured to use replication, the replication of the `put` should occur immediately, not waiting for transaction commit and without the calling thread needing to block waiting for responses from the other nodes in the cluster. This is a "fire-and-forget" semantic that JBoss Cache refers to as *asynchronous replication*. When other nodes receive a replicated `put`, they use the same fail-fast semantics as a local `put` -- i.e. promptly and silently abort if the entity is already cached.

If the cache is configured to use invalidation, a `put` should not result in any cluster-wide message at all. The fact that one node in the cluster has cached an entity should not invalidate another node's cache of that same entity -- both caches are storing the same information.

- A removal of an entity from the cache (i.e. to reflect a DELETE from the underlying database) is basically a special case of a modification; the removal should not be visible on other nodes or to other transactions until the transaction that did the remove commits. Cache consistency after commit means the removed entity is no longer in the cache on any node in the cluster.

2.1.2. Collections

Collection caching refers to the case where a cached entity has as one of its fields a collection of other entities. Hibernate handles this field specially in the second level cache; a special area in the cache is created where the primary keys of the entities in the collection are stored.

The management of collection caching is very similar to entity caching, with a few differences:

- When a new entity is created that includes a collection, no attempt is made to insert the collection into the cache.
- When a transaction updates the contents of a collection, no attempt is made to reflect the new contents of the collection in the cache. Instead, the existing collection is simply removed from the cache across the cluster, using the same semantics as an entity removal.

In essence, for collections Hibernate only supports cache reads and the `put` operation, with any modification of the collection resulting in cluster-wide invalidation of that collection from the cache. If the collection field is accessed again, a new read from the database will be done, followed by another cache `put`.

2.1.3. Queries

Hibernate supports caching of query results in the second level cache. The HQL statement that comprised the query is cached (including any parameter values) along with the primary keys of all entities that comprise the result set.

The semantics of query caching are significantly different from those of entity caching. A database row that reflects an entity's state can be locked, with cache updates applied with that lock in place. The semantics of entity caching take advantage of this fact to help ensure cache consistency across the cluster. There is no clear database analogue to a query result set that can be efficiently locked to ensure consistency in the cache. As a result, the fail-fast semantics used with the entity caching `put` operation are not available; instead query caching has semantics akin to an entity insert, including costly synchronous cluster updates and the JBoss Cache two phase commit protocol. Furthermore, Hibernate must agressively invalidate query results from the cache any time any instance of one of the entity classes involved in the query's WHERE clause changes. All such query results are invalidated, even if the change made to the entity instance would not have affected the query result. It is not performant for Hibernate to try to determine if the entity change would have affected the query result, so the safe choice is to invaldiate the query. See Section 2.1.4, “Timestamps” for more on query invalidation.

The effect of all this is that query caching is less likely to provide a performance boost than entity/collection caching. Use it with care and benchmark your application with it enabled and disabled. Be careful about replicating query results; caching them locally only on the node that executed the query will be more performant unless the query is quite expensive, is very likely to be repeated on other nodes, and is unlikely to be invalidated out of the cache.[1].

The JBoss Cache-based implementation of query caching adds a couple of interesting semantics, both designed to ensure that query cache operations don't block transactions from proceeding:

- The insertion of a query result into the cache is very much like the insertion of a new entity. The difference is it is possible for two transactions, possibly on different nodes, to try to insert the same query at the same time. (If this happened with entities, the database would throw an exception with a primary key violation before any caching work could start). This could lead

[1]See the discussion of the `hibernate.cache.region.jbc2.query.localonly` property in Section 3.1, “Configuring the Hibernate Session Factory”

to long delays as the transactions compete for cache locks. To prevent such delays, the cache integration layer will set a very short (a few ms) lock timeout before attempting to cache a query result. If there is any sort of locking conflict, it will be detected quickly, and the attempt to cache the result will be quietly abandonded.

- A read of a query result does not result in any long-lasting read lock in the cache. Thus, the fact that an uncommitted transaction had read a query result does not prevent concurrent transactions from subsequently invalidating that result and caching a new result set. However, an insertion of a query result into the cache will result in an exclusive write lock that lasts until the transaction that did the insert commits; this lock will prevent other transactions from reading the result. Since the point of query caching is to improve performance, blocking on a cache read for an extended period seems suboptimal. So, the cache integration code will set a very low lock acquisition timeout before attempting the read; if there is a lock conflict, the read will silently fail, resulting in a cache miss and a re-execution of the query against the database.

2.1.4. Timestamps

Timestamp caching is an internal detail of query caching. As part of each query result, Hibernate stores the timestamp of when the query was executed. There is also a special area in the cache (the *timestamps cache*) where, for each entity class, the timestamp of the last update to any instance of that class is stored. When a query result is read from the cache, its timestamp is compared to the timestamps of all entities involved in the query. If any entity has a later timestamp, the cached result is discarded and a new query against the database is executed.

The semantics of of the timestamp cache are quite different from those of the entity, collection and query caches.

- For all nodes in the cluster, the contents of the timestamp cache should be identical, with all timestamps represented. For the other cache types, it is acceptable for some nodes in the cluster to not store some data, as long as everyone who does store an item stores the same thing. Not so with timestamps -- everyone must store all timestamps. Using a JBoss Cache configured for invalidation is not allowed for the timestamps cache. Further, configuring JBoss Cache eviction to remove old or infrequently used data from the timestamps cache should not be done. Also, when a new node joins a running cluster, it must acquire the current state of all timestamps from another member, performing what is known as an *initial state transfer*. For other cache types, an initial state transfer is not required.

- A timestamp represents an entire entity class, not a single instance. Thus it is quite likely that two concurrent transactions will both attempt to update the same timestamp. These updates need to be serialized, but no long lasting exclusive lock on the timestamp is held.
- As soon as a timestamp is updated, the new value needs to be propagated around the cluster. Waiting until the transaction that changed the timestamp commits is inadequate. So, changes to timestamps can be quite "chatty" in terms of how many messages are sent around the cluster. Sending the timestamp update messages synchronously would have a serious impact on performance, and would quite likely result in cluster-wide lock conflicts that would prevent transactions from progressing for tens of seconds at a time. To mitigate these issues, timestamp updates are sent asynchronously.

2.2. Key JBoss Cache Behaviors

JBoss Cache is a very flexible tool and includes a great number of configuration options. See the *JBoss Cache User Guide* for an in depth discussion of these options. Here we focus on the main concepts that are most important to the Second Level Cache use case. This discussion will focus on concepts; see Section 3.2, “Configuring JBoss Cache” for details on the actual configurations involved.

2.2.1. Replication vs. Invalidation vs. Local Mode

JBoss Cache provides three different choices for how a node in the cluster should interact with the rest of the cluster when its local state is updated:

- *Replication:* The updated cache will send its new state (e.g. the new values for an entity's fields) to the other members of the cluster. This is heavy in terms of the size of network traffic, since the new state needs to be transmitted. It also has the effect forcing the data that is cached in each node in the cluster to be the same, even if the users accessing the various nodes are interested in different data. So, if a user on node A reads an `Order` entity with primary key `12343439485030` from the database, with replication that entity will be cached on every node in the cluster, even though no other users are interested in that particular `Order`.

  Because of these downsides, replication is generally not the best choice for entity, collection and query caching. However, in a cluster *replication is the only valid choice for the timestamps cache*.
- *Invalidation:* The updated cache will send a message to the other members of the cluster telling them that a particular piece of data (e.g. a particular

entity) has been modified. Upon receipt of this message, the other nodes will remove this data from their local cache, if it is stored there. Invalidation is lighter than replication in terms of network traffic, since only the "id" of the data needs to be transmitted, not the entire new state. The downside to invalidation is that if the invalidated data is needed again, it has to be re-read from the database. However, in most cases data that many nodes in the cluster all want to have in memory is not data that is frequently changed.

Invalidation makes no sense for query caching; if it is used for a query cache region the Hibernate/JBoss Cache integration will detect this and switch any query cache related calls to Local mode. Invalidation must not be used for timestamp caching; Hibernate will throw an exception during session factory startup if it finds that it is.

- *Local:* The updated cache does not even know if there are any other nodes, and will not attempt to update them. If JBoss Cache is used as a Second Level Cache in a non-clustered environment, Local mode should be used. *If there is a cluster, Local mode should never be used for entity, collection or timestamp caching.* Local mode can be used for query caching in a cluster, since the replicated timestamps cache will ensure that outdated cached queries are not used. Often Local mode is the best choice for query caching, as query results can be large and the cost of replicating them high.

  If the same underlying JBoss Cache instance is used for the query cache and any of the other caches, the `SessionFactory` can be configured to suppress replication of the queries, essentially making the queries operate in Local mode. This is done by adding the following configuration:

  ```
  hibernate.cache.region.jbc2.query.localonly=true
  ```

  If the JBoss Cache instance that the query cache is using is configured for invalidation, setting this property isn't even necessary; the Hibernate/JBoss Cache integration will detect this condition and switch any query cache-related calls to Local mode.

2.2.2. Synchronous vs. Asynchronous

In JBoss Cache terms, *synchronous* vs. *asynchronous* refers to whether a thread that initiates a cluster-wide message blocks until that message has been received, processed and acknowledged by the other members of the cluster. Synchronous means the thread blocks; asynchronous means the message is sent and the thread immediately returns; more of a "fire and

forget". An example of a message would be a set of cache inserts, updates and removes sent out to the cluster as part of a transaction commit.

In almost all cases, the cache should be configured to use synchronous messages, as these are the only way to ensure data consistency across the cluster. JBoss Cache supports programatically overriding the default configured behavior on a per-call basis, so for the special cases where sending a message asynchronously is appropriate, the Hibernate/JBoss Cache integration code will force the call to go asynchronously. So, configure your caches to use synchronous messages.

2.2.3. Locking Scheme

JBoss Cache supports both optimistic and pessimistic locking schemes. See the *JBoss Cache User Guide* for an in depth discussion of these options. In the Second Level Cache use case, the main benefit of optimistic locking is that updates of cached entities by one transaction do not block reads of the cached entity by other transactions, yet REPEATABLE_READ semantics are preserved. Optimistic locking has a higher level of runtime overhead, however.

If you are using optimistic locking and data versioning in Hibernate for your entities, you should use it in the entity cache as well.

2.2.4. Initial State Transfer

When a new node joins a running cluster, it can request from an existing member a copy of that member's current cache contents. This process is known as an *initial state transfer*. Doing an initial state transfer allows the new member to have a "hot cache"; i.e. as user requests come in, data will already be cached, helping avoid an initial set of reads from the database.

However, doing an initial state transfer comes at a cost. The node providing the state needs to lock its entire tree, serialize it and send it over the network. This could be a large amount of data and the transfer could take a considerable period of time to process. While the transfer is ongoing, the state provider is holding locks on its cache, preventing any local or replicated updates from proceeeding. All work around the cache can come to a halt for a period.

Because of this cost, we generally recommend avoiding initial state transfers in the second level cache use case. The exception to this is the timestamps cache. *For the timestamps cache, an initial state transfer is required.*

2.2.5. Cache Eviction

Eviction refers to the process by which old, relatively unused, or excessively voluminous data can be dropped from the cache, allowing the cache to remain within a memory budget. Generally, applications that use the Second Level Cache should configure eviction. See Chapter 4, *Cache Eviction* for details.

2.2.6. Buddy Replication and Cache Loading

Buddy replication refers to a JBoss Cache feature whereby each node in the cluster replicates its cached data to a limited number (often one) of "buddies" rather than to all nodes in the cluster. *Buddy replication should not be used in a Second Level Cache*. It is intended for use cases where one node in the cluster "owns" some data , and just wants to make a backup copy of the data to provide high availability. The data (e.g. a web session) is never meant to be accessed simultaneously on two different nodes in the cluster. Second Level Cache data does not meet this "ownership" criteria; entities are meant to be simultaneously usable by all nodes in the cluster.

Cache Loading refers to a JBoss Cache feature whereby cached data can be persisted to disk. The persisted data either serves as a highly available copy of the data in case of a cluster restart, or as an overflow area for when the amount of cached data exceeds an application's memory budget. *Cache loading should not be used in a Second Level Cache*. The underlying database from which the cached data comes already serves the same functions; adding a cache loader to the mix is just wasteful.

2.3. Matching JBC Behavior to Types of Data

The preceding discussion has gone into a lot of detail about what Hibernate wants to accomplish as it caches data, and what JBoss Cache configuration options are available. What should be clear is that the configurations that are best for caching one type of data are not the best (and are sometimes completely incorrect) for other types. Entities likely work best with synchronous invalidation; timestamps *require* replication; query caching might do best in local mode.

Prior to Hibernate 3.3 and JBoss Cache 2.1, the conflicting requirements between the different cache types led to a real dilemna, particularly if query caching was enabled. This conflict arose because all four cache types needed to share a single underlying cache, with a single configuration. If query caching was enabled, the requirements of the timestamps cache basically forced use of synchronous replication, which is the worst performing

choice for the more critical entity cache and is often inappropriate for the query cache.

With Hibernate 3.3 and JBoss Cache 2.1 it has become possible, even easy, to use separate underlying JBoss Cache instances for the different cache types. As a result, the entity cache can be optimally configured for entities while the necessary configuration for the timestamps cache is maintained.

There were three key changes that make this improvement possible:

2.3.1. The `RegionFactory` Interface

As mentioned previously, Hibernate 3.3 introduced the `RegionFactory` API as its mechanism for managing the Second Level Cache. This API makes it possible for implementations to know at all times whether they are working with entities, collections, queries or timestamps. That knowledge allows the Hibernate/JBoss Cache integration layer to make the best use of the various options JBoss Cache provides.

A Hibernate user doesn't need to understand the `RegionFactory` API in any detail at all; the main point is internally it makes possible independent management of the different cache types.

2.3.2. The `CacheManager` API

The `CacheManager` API is a new feature of JBoss Cache 2.1. It provides an API for managing multiple distinct JBoss Cache instances in the same VM. Basically a `CacheManager` is instantiated and provided a set of *named* cache configurations. An application like the Hibernate/JBoss Cache integration layer accesses the `CacheManager` and asks for a cache configured with a particular named configuration.

Again,a Hibernate user doesn't need to understand the `CacheManager`; it's an internal detail. The thing to understand is that the task of a Hibernate Second Level Cache user is to:

- Provide a set of named JBoss Cache configurations in an XML file (or just use the default set included in the `jbc2-configs.xml` file found in the `org.hibernate.cache.jbc2.builder` package in `hibernate-jbosscache2.jar`).
- Tell Hibernate which cache configurations to use for entity, collection, query and timestamp caching. In practice, this can be quite simple, as there is a reasonable set of defaults.

See Chapter 3, *Configuration* for more on how to do this.

2.3.3. Sharable JGroups Resources

JGroups is the group communication library JBoss Cache uses to send messages around a cluster. Each cache has a JGroups `Channel`; different channels around the cluster that have the same name and compatible configurations detect each other and form a group for message transmission.

A `Channel` is a fairly heavy object, typically using a good number of threads, several sockets and some good sized network I/O buffers. Creating multiple different channels in the same VM was therefore costly, and was an administrative burden as well, since each channel would need separate configuration to use different network addresses or ports. Architecturally, this mitigated against having multiple JBoss Cache instances in an application, since each would need its own `Channel`.

Added in JGroups 2.5 and much improved in the JGroups 2.6 series is the concept of sharable JGroups resources. Basically, the heavyweight JGroups elements can be shared. An application (e.g. the Hibernate/JBoss Cache integration layer) uses a JGroups `ChannelFactory`. The `ChannelFactory` is provided with a set of *named* channel configurations. When a `Channel` is needed (e.g. by a JBoss Cache instance), the application asks the `ChannelFactory` for the channel by name. If different callers ask for a channel with the same name, the `ChannelFactory` ensures that they get channels that share resources.

The effect of all this is that if a user wants to use four separate JBoss Cache instances, one for entity caching, one for collection caching, one for query caching and one for timestamp caching, those four caches can all share the same underlying JGroups resources.

The task of a Hibernate Second Level Cache user is to:

- Provide a set of named JGroups configurations in an XML file (or just use the default set included in the `jgroups-stacks.xml` file found in the `org.hibernate.cache.jbc2.builder` package in the `hibernate-jbosscache2.jar`).
- Tell Hibernate where to find that set of configurations on the classpath. See Section 3.1, “Configuring the Hibernate Session Factory” for details on how to do this. This is not necessary if the default set included in `hibernate-jbosscache2.jar` is used.
- In the JBoss Cache configurations you are using specify the name of the channel you want to use. This should be one of the named configurations in the JGroups XML file. The default set of JBoss Cache configurations found in the `hibernate-jbosscache2.jar` already have appropriate default

choices. See Section 3.2.3.3, “JGroups `Channel` Configuration” for details on how to set this if you don't wish to use the defaults.

See Section 3.3, “JGroups Configuration” for more on JGroups.

2.3.4. Bringing It All Together

So, we've seen that Hibernate caches up to four different types of data (entities, collections, queries and timestamps) and that Hibernate 3.3 + JBoss Cache 2 gives you the flexibility to use a separate underlying JBoss Cache, with different behavior, for each type. You can actually deploy four separate caches, one for each type.

In practice, four separate caches are unnecessary. For example, entities and collection caching have similar enough semantics that there is no reason not to share a JBoss Cache instance between them. The queries can usually use the same cache as well. Similarly, queries and timestamps can share a JBoss Cache instance configured for replication, with the `hibernate.cache.region.jbc2.query.localonly=true` configuration letting you turn off replication for the queries if you want to.

Here's a decision tree you can follow:

1. Decide if you want to enable query caching.
2. Decide if you want to use invalidation or replication for your entities and collections. Invalidation is generally recommended for entities and collections.
   - If you want invalidation, and you want query caching, you will need two JBoss Cache instances, one with synchronous invalidation for the entities and collections, and one with synchronous replication for the timestamps. The queries will go in the timestamp cache if you want them to replicate; they can go with the entities and collections otherwise.
   - If you want invalidation but don't want query caching, you can use a single JBoss Cache instance, configured for synchronous invalidation.
   - If you want replication, whether or not you want query caching, you can use a single JBoss Cache instance, configured for synchronous replication.
3. If you are using query caching, from the above decision tree you've either got your timestamps sharing a cache with other data types, or they are by themselves. Either way, the cache being used for timestamps *must have initial state transfer enabled*. Now, if the timestamps are sharing a cache

with entities, collections or queries, decide whether you want initial state transfer for that other data. See Section 2.2.4, “Initial State Transfer” for the implications of this. If you don't want initial state transfer for the other data, you'll need to have a separate cache for the timestamps.

4. Finally, if your queries are sharing a cache configured for replication, decide if you want the cached query results to replicate. (The timestamps cache *must* replicate.) If not, you'll want to set the `hibernate.cache.region.jbc2.query.localonly=true` option when you configure your `SessionFactory`

Once you've made these decisions, you know whether you need just one underlying JBoss Cache instance, or more than one. Next we'll see how to actually configure the setup you've selected.

Chapter 3. Configuration

There are three main areas of configuration involved in using JBoss Cache 2 for your Hibernate Second Level Cache: configuring the Hibernate `SessionFactory`, configuring the underlying JBoss Cache instance(s), and configuring the JGroups `ChannelFactory`. If you use the standard JBoss Cache and JGroups configurations that ship with the `hibernate-jbosscache2.jar`, then all you need to worry about is the `SessionFactory` configuration.

3.1. Configuring the Hibernate Session Factory

3.1.1. Basics

There are four basic steps to configuring the `SessionFactory`:

- Tell Hibernate you whether to enable caching of entities and collections. No need to set this property if you don't:

```
hibernate.cache.use_second_level_cache=true
```

- Tell Hibernate you want to enable caching of query results. No need to set this property if you don't:

```
hibernate.cache.use_query_cache=true
```

- If you have enabled caching of query results, tell Hibernate if you want to suppress costly replication of those results around the cluster. No need to set this property if you want query results replicated:

```
hibernate.cache.region.jbc2.query.localonly=true
```

- Finally, you need to tell Hibernate what `RegionFactory` implementation to use to manage your caches. You do this by setting the `hibernate.cache.region.factory_class` configuration option.

```
hibernate.cache.region.factory_class=
   org.hibernate.cache.jbc2.MultiplexedJBossCacheRegionFactory
```

  To determine the correct factory class, you must decide whether you need just one underlying JBoss Cache instance to support the different types of caching you will be doing, or whether you need more than one. See Chapter 2, *Core Concepts* and particularly Section 2.3.4, “Bringing It All Together” for more on how to make that decision. Once you know the

answer, see Section 3.1.2, “Specifying the `RegionFactory` Implementation” to find the factory class that best meets your needs.

Once you've specified your factory class, there may be other factory-class-specific configurations you may want to set. The available options are explained below.

3.1.2. Specifying the `RegionFactory` Implementation

Hibernate 3.3 ships with the following `RegionFactory` implementations that work with JBoss Cache 2. Select the one that is appropriate to your needs and use it with the `hibernate.cache.region.factory_class` configuration option.

`org.hibernate.cache.jbc2.SharedJBossCacheRegionFactory`
: Instantiates a single JBoss Cache instance for use with all cache data types (entities, collections, queries, timestamps).

`org.hibernate.cache.jbc2.JndiSharedJBossCacheRegionFactory`
: Uses a single JBoss Cache instance for all cache data types. However, does not instantiate the JBoss Cache instance itself; instead looks for an existing cache in JNDI. This allows sharing of a single JBoss Cache instance across multiple Hibernate session factories running in the same environment.

`org.hibernate.cache.jbc2.MultiplexedJBossCacheRegionFactory`
: Supports using multiple JBoss Cache instances with different cache data types (entities, collections, queries, timestamps) assigned to different JBoss Cache instances. Instantiates a JBoss Cache `CacheManager` to manage the JBoss Cache instances.

`org.hibernate.cache.jbc2.JndiMultiplexedJBossCacheRegionFactory`
: Functions like `JndiMultiplexedJBossCacheRegionFactory`, but instead of instantiating its own `CacheManager` it looks for an existing one in JNDI. This allows sharing of the various JBoss Cache instances across multiple Hibernate session factories running in the same environment.

3.1.3. The `SharedJBossCacheRegionFactory`

The `SharedJBossCacheRegionFactory` supports a number of additional configuration options:

`hibernate.cache.region.jbc2.cfg.shared`
: Classpath or filesystem resource containing JBoss Cache configuration settings the underlying cache should use. Default value is `treecache.xml`.

`hibernate.cache.region.jbc2.cfg.multiplexer.stacks`
: Classpath or filesystem resource containing JGroups protocol stack configurations the `ChannelFactory` should use. Default is `org/hibernate/cache/jbc2/builder/jgroups-stacks.xml`, a file found in the `hibernate-jbosscache2.jar`.

Note that the default `treecache.xml` file does not exist; it is up to the user to provide it.

3.1.4. The `JndiSharedJBossCacheRegionFactory`

The `JndiSharedJBossCacheRegionFactory` supports an additional configuration option:

`hibernate.cache.region.jbc2.cfg.shared`
: Specifies the JNDI name under which the JBoss Cache instance to use is bound. Note that although this configuration property has the same name as the one used by `SharedCacheInstanceManager`, the meaning here is different. Note also that in this class' usage of the property, there is no default value -- the user must specify the property.

The `JndiSharedJBossCacheRegionFactory` requires that the JBoss Cache instance is already bound in JNDI; it will not create and bind one if it isn't. It is up to the the user to ensure the cache is created and bound in JNDI before the Hibernate `SessionFactory` is created.

3.1.5. The `MultiplexedJBossCacheRegionFactory`

The `MultiplexedJBossCacheRegionFactory` supports a number of additional configuration options:

`hibernate.cache.region.jbc2.configs`
: Classpath or filesystem resource containing JBoss Cache configurations the `CacheManager` should use. Default is `org/hibernate/cache/jbc2/builder/jbc2-configs.xml`, a file found in the `hibernate-jbosscache2.jar`.

`hibernate.cache.region.jbc2.cfg.multiplexer.stacks`
: Classpath or filesystem resource containing JGroups protocol stack configurations the `ChannelFactory` should use. Default is `org/hibernate/cache/jbc2/builder/jgroups-stacks.xml`, a file found in the `hibernate-jbosscache2.jar`.

`hibernate.cache.region.jbc2.cfg.entity`
: Name of the configuration that should be used for entity caches. Default value is `optimistic-entity`.

`hibernate.cache.region.jbc2.cfg.collection`
: Name of the configuration that should be used for collection caches. No default value, as by default we try to use the same JBoss Cache instance that is used for entity caching.

`hibernate.cache.region.jbc2.cfg.ts`
: Name of the configuration that should be used for timestamp caches. Default value is `timestamps-cache`.

`hibernate.cache.region.jbc2.cfg.query`
: Name of the configuration that should be used for query caches. By default, tries to use the same cache as is used for entity caching. If there is no entity cache or it uses invalidation, the default value is `local-query`.

Many of the default values name JBoss Cache configurations in the standard `jbc2-configs.xml` file found in the `hibernate-jbosscache2.jar`. See Section 3.2.4, “Standard JBoss Cache Configurations” for details on those configurations. If you want to set `hibernate.cache.region.jbc2.configs` and use your own JBoss Cache configuration file, you can still take advantage of these defaults names; just name the configurations in your file to match.

This is all looks a bit complex, so let's show what happens if you just configure the defaults, with query caching enabled:

```
hibernate.cache.use_second_level_cache=true
hibernate.cache.use_query_cache=true
hibernate.cache.region.factory_class=
   org.hibernate.cache.jbc2.MultiplexedJBossCacheRegionFactory
```

You would end up using two JBoss Cache instances:

- One, for the entities, collection and queries, would use the `optimistic-entity` configuration. This cache would use optimistic locking, synchronous invalidation and would disable initial state transfer.
- The second, for timestamps, would use the `timestamps-cache` configuration. This cache would use pessimistic locking, asynchronous replication and would enable initial state transfer.

See Section 3.2.4, “Standard JBoss Cache Configurations” for more on these standard cache configurations.

If you hadn't set `hibernate.cache.use_query_cache=true` you'd just have the single `optimistic-entity` cache, shared by the entities and collections.

3.1.6. The `JndiMultiplexedJBossCacheRegionFactory`

The `JndiMultiplexedJBossCacheRegionFactory` supports an additional configuration option:

`hibernate.cache.region.jbc2.cachefactory`
: Specifies the JNDI name under which the `CacheManager` to use is bound. There is no default value -- the user must specify the property.

The `JndiMultiplexedJBossCacheRegionFactory` requires that the JBoss Cache `CacheManager` is already bound in JNDI; it will not create and bind one if it isn't. It is up to the the user to ensure the cache is `CacheManager` and bound in JNDI before the Hibernate `SessionFactory` is created.

3.2. Configuring JBoss Cache

JBoss Cache provides a great many different configuration options; here we are just going to look at a few that are most relevant to the Second Level Cache use case. Please see the *JBoss Cache User Guide* for full details.

3.2.1. Configuring a Single Standalone Cache

To configure a single standalone cache (i.e. for use by a `SharedJBossCacheRegionFactory`), you need to create a standard JBoss Cache XML configuration file and place it on the classpath. See the *JBoss Cache User Guide* and the JBoss Cache distribution for examples of JBoss Cache configuration files.

If the resource path to your file is `treecache.xml`, the `SharedJBossCacheRegionFactory` will find it by default; otherwise you will need to use a configuration option to tell it where it is. See Section 3.1.3, “The `SharedJBossCacheRegionFactory`” for instruction on how to do that.

3.2.2. Managing Multiple Caches via a CacheManager

If you are using `MultiplexedJBossCacheRegionFactory` you will need to provide a set of JBoss Cache configurations for its `CacheManager` to use. (Or, use the set in the `jbc2-configs.xml` file that ships with `hibernate-jbosscache2.jar`.) The XML file used by a `CacheManager` is very similar to the usual config file used by a standalone cache; the biggest difference is it can include multiple, named, configurations. The format looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>

<cache-configs>
```

```
    <!-- A config appropriate for entity/collection caching. -->
    <cache-config name="optimistic-entity">

        <!-- Node locking scheme -->
        <attribute name="NodeLockingScheme">OPTIMISTIC</attribute>

        ..... other configuration attributes

    </cache-config>


    <!-- A config appropriate for entity/collection caching that
         uses pessimistic locking -->
    <cache-config name="pessimistic-entity">

        <!-- Node locking scheme -->
        <attribute name="NodeLockingScheme">PESSIMISTIC</attribute>


        ..... other configuration attributes

    </cache-config>

</cache-configs>
```

Each `<cache-config>` element contains a complete cache configuration, with the same contents as what you would place in the `<mbean>` element in a standalone cache configuration file. The `name` attribute on the `<cache-config>` element provides the identifier for the configuration; users of the `CacheManager` will provide this name when requesting a JBoss Cache instance that uses this configuration.

3.2.3. JBoss Cache Configuration Details

Let's look at how to specify a few of the JBoss Cache configuration options that are most relevant to the Hibernate use case.

3.2.3.1. CacheMode

The JBoss Cache *CacheMode* attribute encapsulates whether the cache uses replication, invalidation or local mode, as well as whether messages should be synchronous or asynchronous. See Section 2.2.1, “Replication vs. Invalidation vs. Local Mode” and Section 2.2.2, “Synchronous vs. Asynchronous” for a discussion of these concepts.

The CacheMode is configured as follows:

```
<!-- Legal modes are LOCAL
                     REPL_ASYNC
                     REPL_SYNC
```

```
                         INVALIDATION_ASYNC
                         INVALIDATION_SYNC
-->
<attribute name="CacheMode">INVALIDATION_SYNC</attribute>
```

3.2.3.2. NodeLockingScheme

The JBoss Cache *NodeLockingScheme* attribute configures whether optimistic locking or pessimistic locking should be used. See Section 2.2.3, “Locking Scheme” for a discussion of locking.

The NodeLockingScheme is configured as follows:

```
<!-- Node locking scheme:
        OPTIMISTIC
        PESSIMISTIC (default)
-->
<attribute name="NodeLockingScheme">OPTIMISTIC</attribute>
```

3.2.3.3. JGroups `Channel` Configuration

Each JBoss Cache instance (except those with CacheMode LOCAL) will need a JGroups `Channel`. The cache configuration needs to tell JGroups how to set up the channel's protocol stack. This is configured as follows:

```
<attribute name="MultiplexerStack">udp</attribute>
```

An alternate approach is to include the full protocol stack configuration in the JBoss Cache configuration:

```
<!-- JGroups protocol stack properties. -->
<attribute name="ClusterConfig">
   <config>
      <UDP mcast_addr="228.10.10.10"
           mcast_port="45588"
           tos="8"

           ... many more details

      <pbcast.STATE_TRANSFER/>
      <pbcast.FLUSH timeout="0"/>
   </config>
</attribute>
```

See Section 3.3, “JGroups Configuration” for more on JGroups configuration.

3.2.3.4. Initial State Transfer

See Section 2.2.4, “Initial State Transfer” for a discussion of the concept of initial state transfer.

Initial State Transfer is configured as follows:

```
<!-- Whether or not to fetch state on joining a cluster. -->
<attribute name="FetchInMemoryState">false</attribute>
```

3.2.3.5. Region-Based Marshalling

JBoss Cache includes a feature called *Region Based Marshalling* that helps ensure the correct classloader is in place when objects are serialized and deserialized as part of replication and invalidation messages. This feature adds some overhead, but it allows your cache to work in complex classloading environments such as those found in many application servers. It can be disabled if your application meets the following criteria:

- No cached entities use custom types (i.e. types persisted as BLOBs or CLOBs or as a Hibernate UserType) in their fields, and no entities use complex primary keys.
- *OR* all custom types and complex primary key types are visible to the classloader that loads JGroups.

Region based marshalling is configured as follows:

```
<!--
 Whether to use marshalling or not. Default is "false".
-->
<attribute name="UseRegionBasedMarshalling">true</attribute>
<!-- Must match the value of "UseRegionBasedMarshalling" -->
<attribute name="InactiveOnStartup">true</attribute>
```

Region based marshalling is enabled in the standard cache configurations that ship with `hibernate-jbosscache2.jar`

3.2.3.6. Eviction

This topic deserves a chapter of it's own. See Chapter 4, *Cache Eviction*.

3.2.4. Standard JBoss Cache Configurations

Hibernate ships with a number of standard JBoss Cache configurations in the `hibernate-jbosscache2.jar`'s `jbc2-configs.xml` file. The following table highlights the key features of each configuration.

Table 3.1. Standard JBoss Cache Configurations

Name	Valid For	Optimal For	CacheMode	Locking	State Transfer
optimistic-entity	E/C/Q	E/C/Q	INVALIDATION_SYNC	OPTIMISTIC	no
pessimistic-entity	E/C/Q	E/C/Q	INVALIDATION_SYNC	PESSIMISTIC	no
local-query	Q	Q	LOCAL	PESSIMISTIC	no
replicated-query	Q	--	LOCAL	OPTIMISTIC	no
timestamps-cache	T	T	REPL_ASYNC	PESSIMISTIC	yes
optimistic-shared	E/C/Q/T	--	REPL_SYNC	OPTIMISTIC	yes
pessimistic-shared	E/C/Q/T	--	REPL_SYNC	PESSIMISTIC	yes

A few notes on the above table:

- *Valid For* and *Optimal For* describe the suitability of the configuration for the four types of data -- Entities, Collections, Queries and Timestamps.
- *State Transfer* indicates whether an initial state transfer will be performed when a cache region is activated.
- Each of the configurations uses the `udp` JGroups protocol stack configuration (except `local-query`, which doesn't use JGroups at all). Since they all use the same stack, if more than one of these caches is created they will share their JGroups resources. See Section 3.3.2, “Standard JGroups Configurations” for a description of the standard stacks.

These standard configurations are a good choice for many applications. The primary reason users may want to use their own configurations is to support more complex eviction setups. See Chapter 4, *Cache Eviction* for more on the kinds of things you can do with eviction.

3.3. JGroups Configuration

JGroups configuration is a complex area that goes well beyond the scope of this document. Users interested in exploring the details are encouraged to

visit the JGroups website at http://www.jgroups.org as well as the JGroups wiki page at jboss.com.

The `jgroups-stacks.xml` file found in the `org.hibernate.cache.jbc2.builder` package in the `hibernate-jbosscache2.jar` provides a good set of standard JGroups configurations; these should be suitable for most needs. If you need to create your own configuration set, we recommend that you start with this file as a base.

3.3.1. Transport -- UDP vs. TCP

The JGroups *transport* refers to the mechanism JGroups uses for sending messages to the group members. Choosing which transport to use is the main JGroups-related decision users will need to make. There are three transport types:

- *UDP* -- uses UDP multicast for transmitting messages to all members of the group. Multicast is efficient for the Second Level Cache use case, particularly with larger clusters, since maintaining cache consistency means many messages need to be sent to all members. UDP-based channel configurations are used in the JBoss Cache configurations in the standard `jbc2-configs.xml` file that ships with Hibernate.

- *TCP* -- uses multiple TCP unicasts for transmitting messages to all members of the group. Less efficient for the Second Level Cache use case, particularly as clusters get larger, i.e. over 4 to 6 nodes. However, in some network environments UDP multicast is not an option, in which case TCP is available.

  If a TCP-based channel is used, there are a couple of options available for how a channel "discovers" the other members in the group when it first starts.

  - *TCP + TCPPING* -- here a static list of all the possible nodes in the cluster must be provided. The TCPPING "discovery" protocol uses that static list to probe for available peers, using TCP messages. The downside here is the need to provide and maintain the static list. The upside, for multicast-averse environments, is all communication uses TCP.

  - *TCP + MPING* -- here the MPING "discovery" protocol uses UDP multicast to probe for available peers; once they are detected all regular message traffic uses TCP. The downside here, for the multicast-averse, is that multicast is used. The upside is there is no need for a static configuration of all possible peers.

The TCP-based configurations in the `jgroups-stacks.xml` file found in `hibernate-jbosscache2.jar` all use TCP + MPING. This is because there is no way for the Hibernate authors to provide a meaningful static configuration for TCPPING. If you want to use TCP + TCPPING you will need to provide your own JGroups configuration file.

- *TUNNEL* -- a specialized protocol designed for cases where cluster members need to communicate across firewalls. See the JGroups documentation for details.

3.3.2. Standard JGroups Configurations

By default the Second Level Cache will use JGroups configurations found in the `jgroups-stacks.xml` file included in `hibernate-jbosscache2.jar`. Following are their names and a brief description of each:

`udp`
: UDP multicast-based config; appropriate for all cache types. Includes JGroups flow control (FC) which is needed for caches that send significant numbers of asynchronous messages (i.e. timestamp caches and entity/collection caches configured for replication instead of invalidation). If you're not sure and want to use UDP multicast, this is the best choice.

`udp-sync`
: UDP multicast-based config that omits JGroups flow control (FC). Optimal for caches that send little or no asynchronous messages, i.e. entity/collection caches configured for invalidation. Unsuitable for timestamp caches or entity/collection caches configured for replication.

`tcp`
: TCP-based config with UDP multicast discovery (MPING). Includes JGroups flow control (FC); see `udp` above for the significance of that.

`tcp-sync`
: TCP-based config with UDP multicast discovery (MPING). Omits JGroups flow control (FC); see `udp-sync` above for the significance of that.

`tunnel`
: Uses a specialized protocol designed for cases where cluster members need to communicate across firewalls. See the JGroups documentation for details.

Chapter 4. Cache Eviction

4.1. Overview

Eviction refers to the process by which old, relatively unused, or excessively voluminous data can be dropped from the cache, allowing the cache to remain within a memory budget. Generally, applications that use the Second Level Cache should configure eviction, unless only a relatively small amount of reference data is cached. This chapter provides a brief overview of how JBoss Cache eviction works, and then explains how to configure eviction to effectively manage the data stored in a Hibernate Second Level Cache. A basic understanding of JBoss Cache eviction and of concepts like FQNs is assumed; see the *JBoss Cache User Guide* for more information.

4.1.1. The Eviction Process

The JBoss Cache eviction process is fairly straightforward. Whenever a node in a cache read or written to, added or removed, the cache finds the *eviction region* (see below) that contains the node and passes an *eviction event* object to the *eviction policy* (see below) associated with the region. The eviction policy uses the stream of events it receives to track activity in the region. Periodically, a background thread runs and contacts each region's eviction policy. The policy uses its knowledge of the activity in the region, along with any configuration it was provided at startup, to determine which if any cache nodes should be evicted from memory. It then tells the cache to evict those nodes. Evicting a node means dropping it from the cache's in-memory state. The eviction only occurs on that cache instance; there is no cluster-wide eviction.

An important point to understand is that eviction proceeds independently on each peer in the cluster, with what gets evicted depending on the activity on that peer. There is no "global eviction" where JBoss Cache removes a piece of data in every peer in the cluster in order to keep memory usage inside a budget. The Hibernate/JBC integration layer may remove some data globally, but that isn't done for the kind of memory management reasons we're discussing in this chapter.

An effect of this is that even if a cache is configured for replication, if eviction is enabled the contents of a cache will be different between peers in the cluster; some may have evicted some data, while others will have evicted different data. What gets evicted is driven by what data is accessed by users on each peer.

Controlling when data is evicted from the cache is a matter of setting up appropriate eviction regions and configuring appropriate eviction policies for each region.

4.1.2. Eviction Regions

JBoss Cache stores its data in a set of nodes organized in a tree structure. An eviction region is a just a portion of the tree to which an eviction policy has been assigned. The name of the region is the FQN of the topmost node in that portion of the tree. An eviction configuration always needs to include a special region named `_default_`; this region is rooted in the root node of the tree and includes all nodes not covered by other regions.

It's possible to define regions that overlap. In other words, one region can be defined for */a/b/c*, and another defined for */a/b/c/d* (which is just the *d* subtree of the */a/b/c* sub-tree). The algorithm that assigns eviction events to eviction regions handles scenarios like this consistently by always choosing the first region it encounters. So, if the algorithm needed to decide how to handle an event affecting */a/b/c/d/e*, it would start from there and work its way up the tree until it hits the first defined region - in this case */a/b/c/d*.

4.1.3. Eviction Policies

An *Eviction Policy* is a class that knows how to handle eviction events to track the activity in its region. It may have a specialized set of configuration properties that give it rules for when a particular node in the region should be evicted. It can then use that configuration and its knowledge of activity in the region to to determine what nodes to evict.

JBoss Cache ships with a number of eviction policies. See the *JBoss Cache User Guide* for a discussion of all of them. Here we are going to focus on just two.

4.1.3.1. The `LRUPolicy`

The `org.jboss.cache.eviction.LRUPolicy` evicts nodes that have been Least Recently Used. It has the following configuration parameters:

- `maxNodes` - This is the maximum number of nodes allowed in this region. 0 denotes no limit. If the region has more nodes than this, the least recently used nodes will be evicted until the number of nodes equals this limit.
- `timeToLiveSeconds` - The amount of time a node is not written to or read (in seconds) before the node should be evicted. 0 denotes no limit. Nodes that exceed this limit will be evicted whether or not a `maxNodes` limit has been breached.

- `maxAgeSeconds` - Lifespan of a node (in seconds) regardless of idle time before the node is swept away. 0 denotes no limit. Nodes that exceed this limit will be evicted whether or not a `maxNodes` or `timeToLiveSeconds` limit has been breached.

- `minTimeToLiveSeconds` - the minimum amount of time a node must be allowed to live after being accessed before it is allowed to be considered for eviction. 0 denotes that this feature is disabled, which is the default value. Should be set to a value less than `timeToLiveSeconds`. It is recommended that this be set to a value slightly greater than the maximum amount of time a transaction that affects the region should take to complete. Configuring this is particularly important when optimistic locking is used in conjunction with invalidation.

4.1.3.2. The `NullEvictionPolicy`

The `org.jboss.cache.eviction.NullEvictionPolicy` is a simple policy that very efficiently does ... nothing. It is used to efficiently short-circuit eviction handling for regions where you don't want objects to be evicted (e.g. the timestamps cache, which should *never* have data evicted). Since the `NullEvictionPolicy` doesn't actually evict anything, it doesn't take any configuration parameters.

4.2. Organization of Data in the Cache

In order to understand how to configure eviction, you need to understand how Hibernate organizes data in the cache.

4.2.1. Region Prefix and Region Name

All FQNs in a second level cache include two elements:

- A *Region Prefix*, which is simply any value assigned to the `hibernate.cache.region_prefix` Hibernate configuration property. If no Region Prefix is set, this portion of the FQN is omitted.

  If different session factories are sharing the same underlying JBoss Cache instance(s) it is *strongly encouraged* that a distinct Region Prefix be assigned to each. This will help ensure that the different session factories cache their data in different subtrees in JBoss Cache.

- A *Region Name*, which is either

  - any value assigned to a `<cache>` element's `region` attribute in a class or collection mapping. See Section 4.2.2, “Entities” for an example.

- Any value assigned to a Hibernate `Query` object's `cacheRegion` property. See Section 4.2.4, “Queries” for an example.
- The *escaped class name* of the type being cached. An *escaped class name* is simply a fully-qualified class name with any `.` replaced with a `/` -- for example `org/example/Foo`.

4.2.2. Entities

The FQN for the cache region where entities of a particular class is stored is derived as follows:

`/` + *Region Prefix* + `/` + *Region Name* + `/ENTITY`

If no region prefix was specified, the leading `/` and *Region Prefix* is not included in the FQN. So, if `hibernate.cache.region_prefix` was set to "appA" and a class was mapped like this:

```
<class name="org.example.Foo">
    <cache usage="transactional" region="foo_region"/>
    ....
</class>
```

The FQN of the region where `Foo` entities would be cached is `/appA/foo_region/ENTITY`.

If the class mapping does not include a `region` attribute, the region name is based on the name of the entity class, e.g.

```
<class name="org.example.Bar">
    <cache usage="transactional"/>
    ....
</class>
```

the FQN of the region where `Bar` entities would be cached is `/appA/org/example/Bar/ENTITY`.

4.2.3. Collections

The FQN for the cache region where entities of a particular class is stored is derived as follows:

`/` + *Region Prefix* + `/` + *Region Name* + `/COLL`

So, let's say our example `Foo` entity included a collection field `bars` that we wanted to cache:

```
<class name="org.example.Foo">
    <cache usage="transactional"/>
    ....
    <one-to-many name="bars" class="org.example.Bar">
        <cache usage="transactional" region="foo_region"/>
    </one-to-many>
</class>
```

The FQN of the region where the collection would be cached would be `/appA/foo_region/COLL`.

If the collection's `<cache>` element did not include a `region`, the FQN would be `/appA/org/example/Foo/COLL`.

4.2.4. Queries

Queries follow this pattern:

`/` + *Region Prefix* + `/` + *Region Name* + `/QUERY`

Say we had the following query (again with a region prefix set to "appA"):

```
List blogs = sess.createQuery("from Blog blog " +
                              "where blog.blogger = :blogger")
    .setEntity("blogger", blogger)
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

The FQN of the region where this query's results would be cached would be `/appA/frontpages/QUERY`.

If the call to `setCacheRegion("frontpages")` were ommitted, the *Region Name* portion of the FQN would be based on a Hibernate class: `/appA/org/hibernate/cache/StandardQueryCache/QUERY`

4.2.5. Timestamps

Timestamps follow this pattern:

`/TS/` + *Region Prefix* + `/org/hibernate/cache/UpdateTimestampsCache`

again with a `/` and the *Region Prefix* portion omitted if no region prefix was set.

Note that in the timestamps case the special constant ("TS") comes at the start of the FQN rather than the end. This makes it easier to ensure that eviction is never enabled for the timestamps region.

4.3. Example Configuration

So far we've been looking at things in the abstract; let's see an example of how this comes together. In this example, imagine we have a Hibernate application with the following characteristics.

- Query caching is enabled.
- There is a region prefix set as part of the Hibernate configuration: `hibernate.cache.region_prefix==appA`
- Some cachable entities and collections have a region name of "reference" set in their Hibernate mapping.
- Some cachable queries have the "reference" region name set when they are created.
- Other cachable entities and collections in the `org.example.hibernate` package don't have a region name set in their Hibernate mapping.
- Other cachable queries don't have a region name set when they are created.

Let's see a possible eviction configuration for this scenario:

```
<attribute name="EvictionPolicyConfig">
  <config>

    <attribute name="wakeUpIntervalSeconds">5</attribute>
    <attribute
 name="policyClass">org.jboss.cache.eviction.LRUPolicy</attribute>

    <!--
      Default region to pick up anything we miss in the more
 specific
      regions below.
    -->
    <region name="/_default_">
       <attribute name="maxNodes">500</attribute>
       <attribute name="timeToLiveSeconds">300</attribute>
       <attribute name="minTimeToLiveSeconds">120</attribute>
    </region>

    <!--  Don't ever evict modification timestamps -->
    <region name="/TS"
       policyClass="org.jboss.cache.eviction.NullEvictionPolicy"/>

    <!-- Reference data -->
    <region name="/appA/reference">
       <!-- Keep all reference data if it's being used -->
```

```
        <attribute name="maxNodes">0</attribute>
        <!-- Keep it around a long time (4 hours) -->
        <attribute name="timeToLiveSeconds">14400</attribute>
        <attribute name="minTimeToLiveSeconds">120</attribute>
     </region>

     <!-- Be more aggressive about queries on reference data -->
     <region name="/appA/reference/QUERY">
        <attribute name="maxNodes">200</attribute>
        <attribute name="timeToLiveSeconds">1000</attribute>
        <attribute name="minTimeToLiveSeconds">120</attribute>
     </region>

     <!--
        Lots of entity instances from this package, but different
        users are unlikely to share them. So, we can cache
        a lot, but evict unused ones pretty quickly.
     -->
     <region name="/appA/org/example/hibernate">
        <attribute name="maxNodes">50000</attribute>
        <attribute name="timeToLiveSeconds">1200</attribute>
        <attribute name="minTimeToLiveSeconds">120</attribute>
     </region>

     <!-- Clean up misc queries very promptly -->
     <region name="/appA/org/hibernate/cache/StandardQueryCache">
        <attribute name="maxNodes">200</attribute>
        <attribute name="timeToLiveSeconds">240</attribute>
        <attribute name="minTimeToLiveSeconds">120</attribute>
     </region>

   </config>
</attribute>
```

Notes on the above:

- The `wakeUpIntervalSeconds` configuration controls how often the background eviction process kicks in to evict nodes.
- The first `policyClass` configuration sets the default eviction policy class to use for each region. Here we want to use the standard `LRUPolicy` This can be overridden on a per-region basis, as is done here for the `/TS` region.
- We set up a `/_default_` region. Having such a region is a requirement if eviction is used. Here we don't expect any data to end up in this default region, but if by mistake someone adds a new entity type that doesn't fall into one of our other regions, we may not have a large memory budget for it so we evict fairly agressively.
- Evicting timestamps is forbidden, so we add a `/TS` region that disables it. Here we see how to override the default eviction policy.

- The `/appA/reference` region covers our reference data entities and collections. This is our most likely to be reused data, so we configure the cache to be very slow to evict it.
- The queries related to our reference data are less likely to be reused, and may take up a lot of memory, so we override the `/appA/reference` region with a `/appA/reference/QUERY` region that is more agressive about eviction.
- The `org.example.hibernate` package includes a lot of entity classes like `Order`, where there are hundreds of thousands of records in the database. These are unlikely to be reused across users, but we have a lot of users and want to be able to cache many of them so a user can have fast access to his or her data during the course of their interaction with the system. So we create a `/appA/org/example/hibernate` region with a high `maxNodes` value but a fairly low `timeToLiveSeconds`. The low time-to-live ensures an `Order` is evicted quickly once a user is done with it.
- Finally, cacheable queries that aren't assigned to to the `reference` region will end up in `/appA/org/hibernate/cache/StandardQueryCache`. We've elected not to keep these around long at all.

4.4. Best Practices

Some best practices to follow:

- Set `hibernate.cache.region_prefix` in your configuration. It makes it simple to ensure the different session factories don't step on each other if they share a JBoss Cache instance.
- Always set up an eviction region for the `/TS` FQN that uses the `NullEvictionPolicy`. This will ensure that timestamps never get evicted. Even if you are not doing query caching or aren't caching timestamps in a particular cache, this is still a good practice, as it costs almost nothing and helps to ensure that timestamp eviction doesn't slip in unnoticed later.
- Assign a region to your entities, collections and queries rather than relying on class names to compose the FQN. It makes it easier to set up eviction, and helps prevent your eviction setup breaking if class names are refactored.
- Assign a different region name to your entities, collections or queries that have different desirable eviction characteristics. Put objects like often used reference data in one region, data probably only accessed by a single user in another. Aggressively evict the latter region; be less agressive with the former if you evict it at all.

- In some cases, there is an external application (i.e. outside of Hibernate's control) that can modify data in the database. Generally, a Second Level Cache should not be used in this sort of case, since it can result in data in the cache being out of date with respect to the database. But sometimes application designers can tolerate having out of date data in the cache. In this sort of situation, use an `LRUPolicy` with a fairly low `maxAgeSeconds`. This will ensure that out-of-date data eventually gets flushed from the cache.

Chapter 5. Architecture

We've now gone through all the main concepts and the configuration details; now we'll look a bit under the covers to understand a bit more about the architectural design of the Hibernate/JBoss Cache integration. Readers can skip this chapter if they aren't interested in a look under the covers.

5.1. Hibernate Interface to the Caching Subsystem

The rest of Hibernate interacts with the Second Level Cache subsystem via the `org.hibernate.cache.RegionFactory` interface. What implementation of the interface is used is determined by the value of the `hibernate.cache.region.factory_class` configuration property. The interface itself is straightforward:

```
void start(Settings settings, Properties properties)
       throws CacheException;

void stop();

boolean isMinimalPutsEnabledByDefault();

long nextTimestamp();

EntityRegion buildEntityRegion(String regionName,
                               Properties properties,
                               CacheDataDescription metadata)
            throws CacheException;

CollectionRegion buildCollectionRegion(String regionName,
                                       Properties properties,
                                       CacheDataDescription cdd)
            throws CacheException;

QueryResultsRegion buildQueryResultsRegion(String regionName,
                                           Properties properties)
            throws CacheException;

TimestampsRegion buildTimestampsRegion(String regionName,
                                       Properties properties)
            throws CacheException;
```

- The `start` method is invoked during `SessionFactory` startup and allows the region factory implementation to access all the configuration settings and initialize itself. The `stop` method is invoked during `SessionFactory` shutdown.
- The various `build***Region` methods are invoked as Hibernate detects it needs to cache different data. Hibernate can invoke these methods

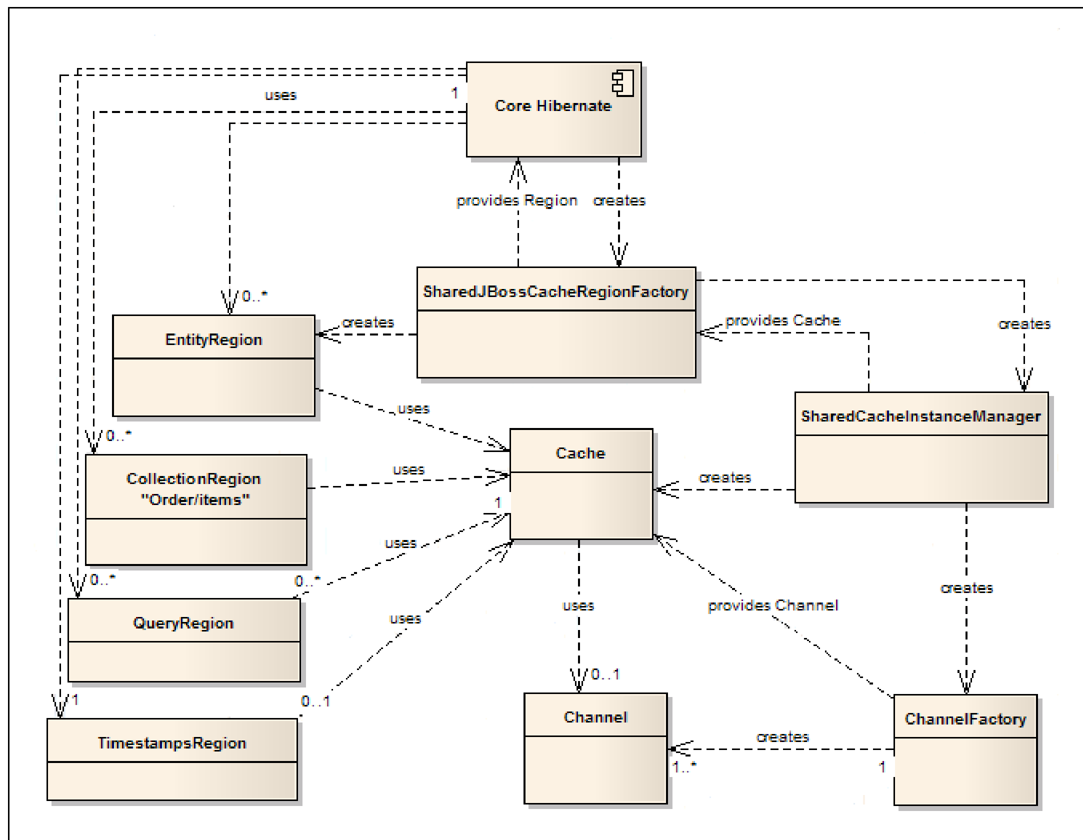
multiple times, with different `regionName` values; each call results in the establishment of a separate area in the underlying JBoss Cache instance(s). For example, if an application includes an entity class `org.example.Order` and another entity class `org.example.LineItem`, you would see two calls to `buildEntityRegion`, one for the `Order` class and one for the `LineItem` class. (Note that it is possible, and recommended, to configure one or more shared regions for entities, collections and queries. See Section 4.2, “Organization of Data in the Cache” for some examples.)

- For each call to a `build***Region` method, the region factory returns a region object implementing the `EntityRegion`, `CollectionRegion`, `QueryResultsRegion` or `TimestampsRegion` interface. Each interface specifies the needed semantics for caching the relevant type of data. Thereafter, the Hibernate core invokes on that region object to manage reading and writing data in the cache.

Next, we'll look at the architecture of how the JBoss Cache integration implements these interfaces, first in the case where a single JBoss Cache instance is used, next in the case where multiple instances are desired.

5.2. Single JBoss Cache Instance Architecture

The following diagram illustrates the key elements involved when a single JBoss Cache instance is used:

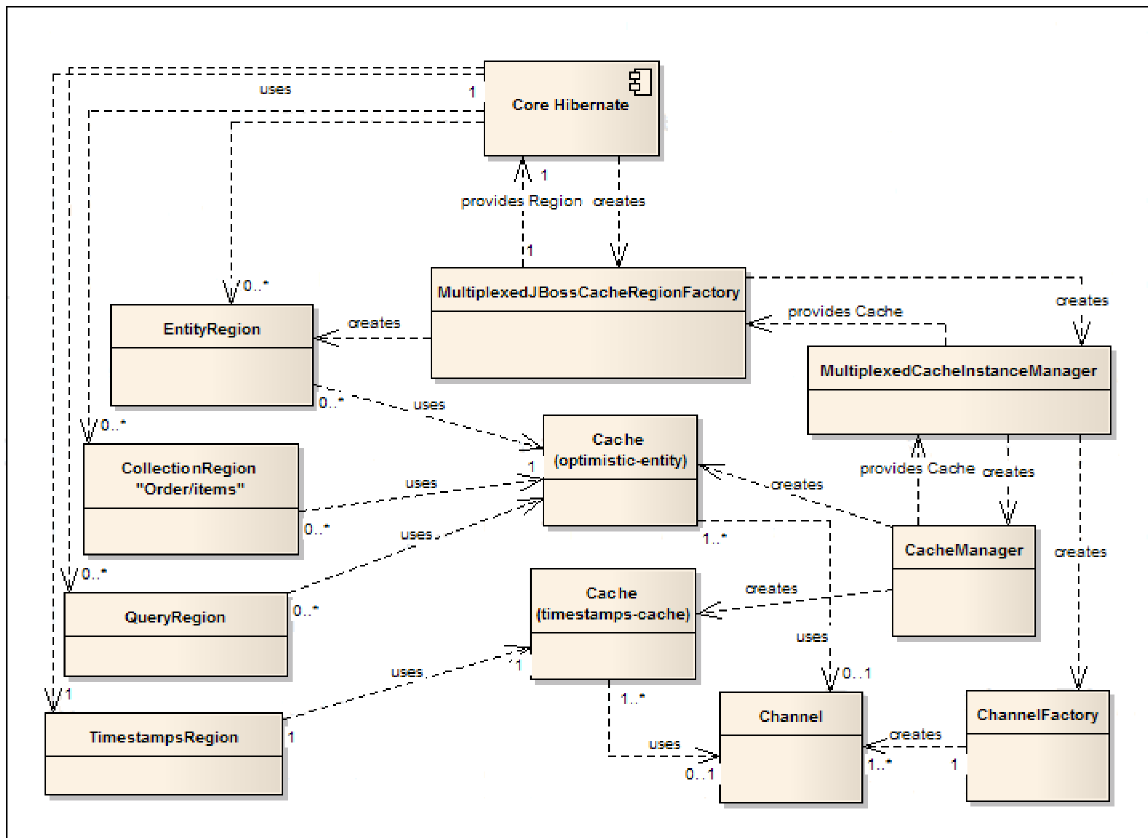


- For the single cache case, the user should specify `SharedJBossCacheRegionFactory` as their `hibernate.cache.region.factory_class`.
- As part of its startup process, te region factory delegates responsibility for providing JBoss Cache instances to an implementation of the `org.hibernate.cache.jbc2.CacheInstanceManager` interface. The region factory separately requests a JBoss Cache instance for entities, one for collections, one for queries and one for timestamps. Whether the `CacheInstanceManager` provides the same underlying JBoss Cache instance for each request or provides multiple caches is an implementation detail of the `CacheInstanceManager`.
- `SharedJBossCacheRegionFactory` creates an instance of `SharedCacheInstanceManager` as its `CacheInstanceManager`. `SharedCacheInstanceManager` uses the JBoss Cache configuration file specified by the user to create a single `org.jboss.cache.Cache` instance, and provides that same instance to the region factory when it requests the cache for entities, collections, queries and timestamps. `SharedCacheInstanceManager` also creates an `org.jgroups.ChannelFactory` and passes it to the `Cache`. The `ChannelFactory` provides the cache with the `org.jgroups.Channel` it uses for intra-cluster communication.

- At this point, the region factory has a reference to a cache for entities, a reference to a cache for collections, one for queries and one for timestamps. In this particular case, each reference points to the same underlying `Cache` instance. When core Hibernate invokes the `buildEntityRegion` operation on the region factory, it instantiates an implementation of the `EntityRegion` interface that knows how to interface with JBoss Cache, passing it a reference to its entity cache. Same thing happens for collections, etc.
- Core Hibernate invokes on the `EntityRegion`, which in turn invokes read and write operations on the underlying JBoss Cache. The cache uses its `Channel` to propagate changes to its peers in the cluster.
- When the `SessionFactory` shuts down, it invokes `stop()` on the region factory, which in turn ensures that the JBoss Cache instance is stopped and destroyed (which in turn closes the JGroups channel).

5.3. Multiple JBoss Cache Instance Architecture

The situation when multiple JBoss Cache instances are used is very similar to the single cache case:



- Here the user should specify `MultiplexedJBossCacheRegionFactory` as their `hibernate.cache.region.factory_class`. The

`MultiplexedJBossCacheRegionFactory` shares almost all its code with `SharedJBossCacheRegionFactory`; the main difference is it constructs a different `CacheInstanceManager` implementation -- the `MultiplexedCacheInstanceManager`.

- `MultiplexedCacheInstanceManager` differs from `SharedCacheInstanceManager` in that it does not directly instantiate a cache. Rather, it creates an instance of `org.jboss.cache.CacheManager`, providing it with a `ChannelFactory` and the location of the user-specified cache configuration file. The `CacheManager` parses the configuration file.

- `MultiplexedCacheInstanceManager` analyzes Hibernate's configuration to determine the name of the desired cache configuration for entities, collections, queries and timestamps. See Section 3.1.5, “The `MultiplexedJBossCacheRegionFactory`” for details. It then asks its `CacheManager` to provide each needed cache. In the diagram, two different caches are needed:

  - One, using the "optimistic-entity" configuration, that is used for entities, collections and queries

  - Another, with the "timestamps-cache" configuration, that is used for timestamps.

  Both the "optimistic-entity" configuration and the "timestamps-cache" configuration specify the use of the "udp" JGroups channel configuration, so the `CacheManager`'s `ChannelFactory` will ensure that they share the underlying JGroups resources.

- The way the region factory creates regions is exactly the same as in the single JBoss Cache case; the only difference is the region factory's internal reference to its timestamps cache now points to a different cache object from the entity, collection and query cache references.