JBoss Enterprise Application Platform Common Criteria Certification 5

JBoss Cache User Guide

for Use with JBoss Enterprise Application Platform 5 Common Criteria Certification Edition 5.1.0

Last Updated: 2017-11-20

JBoss Enterprise Application Platform Common Criteria Certification 5 JBoss Cache User Guide

for Use with JBoss Enterprise Application Platform 5 Common Criteria Certification
Edition 5.1.0

Manik Surtani
manik@jboss.org

Brian Stansberry
brian.stansberry@jboss.com

Galder Zamarreño
galder.zamarreno@jboss.com

Mircea Markus
mircea.markus@jboss.com

Legal Notice

Abstract

This book is the User Guide for Cache of the JBoss Enterprise Application Platform 5.1.0.

Table of Contents

PREFACE

This is the official JBoss Cache Users' Guide. Along with its accompanying documents (an FAQ, a tutorial and a whole set of documents on POJO Cache), this is freely available on the JBoss Cache documentation website.

When used, JBoss Cache refers to JBoss Cache Core, a tree-structured, clustered, transactional cache. POJO Cache, also a part of the JBoss Cache distribution, is documented separately. (POJO Cache is a cache that deals with Plain Old Java Objects, complete with object relationships, with the ability to cluster such POJOs while maintaining their relationships. Please see the POJO Cache documentation for more information about this.)

This book is targeted at developers wishing to use JBoss Cache as either a standalone in-memory cache, a distributed or replicated cache, a clustering library, or an in-memory database. It is targeted at application developers who wish to use JBoss Cache in their code base, as well as "OEM" developers who wish to build on and extend JBoss Cache features. As such, this book is split into two major sections - one detailing the "User" API and the other going much deeper into specialist topics and the JBoss Cache architecture.

In general, a good knowledge of the Java programming language along with a strong appreciation and understanding of transactions and concurrent programming is necessary. No prior knowledge of JBoss Application Server is expected or required.

For further discussion, use the user forum available on the JBoss Cache website. We also provide a mechanism for tracking bug reports and feature requests on the JBoss Cache JIRA issue tracker.

If you are interested in the development of JBoss Cache or in translating this documentation into other languages, we'd love to hear from you. Please post a message on the JBoss Cache user forum or contact us by using the JBoss Cache developer mailing list.

This book is specifically targeted at the JBoss Cache release of the same version number. It may not apply to older or newer releases of JBoss Cache. It is important that you use the documentation appropriate to the version of JBoss Cache you intend to use.

I always appreciate feedback, suggestions and corrections, and these should be directed to the developer mailing list rather than direct emails to any of the authors. We hope you find this book useful, and wish you happy reading!

Manik Surtani, October 2008

PART I. INTRODUCTION TO JBOSS CACHE

This section covers what developers would need to quickly start using JBoss Cache in their projects. It covers an overview of the concepts and API, configuration and deployment information.

CHAPTER 1. OVERVIEW

1.1. WHAT IS JBOSS CACHE?

JBoss Cache is a tree-structured, clustered, transactional cache. It can be used in a standalone, non-clustered environment, to cache frequently accessed data in memory thereby removing data retrieval or calculation bottlenecks while providing "enterprise" features such as JTA compatibility, eviction and persistence.

JBoss Cache is also a clustered cache, and can be used in a cluster to replicate state providing a high degree of failover. A variety of replication modes are supported, including invalidation and buddy replication, and network communications can either be synchronous or asynchronous.

When used in a clustered mode, the cache is an effective mechanism of building high availability, fault tolerance and even load balancing into custom applications and frameworks. For example, the JBoss Application Server and Red Hat's Enterprise Application Platform make extensive use of JBoss Cache to cluster services such as HTTP and EJB sessions, as well as providing a distributed entity cache for JPA.

1.1.1. And what is POJO Cache?

POJO Cache is an extension of the core JBoss Cache API. POJO Cache offers additional functionality such as:

- maintaining object references even after replication or persistence
- fine grained replication, where only modified object fields are replicated
- "API-less" clustering model where POJOs are simply annotated as being clustered

POJO Cache has a complete and separate set of documentation, including a Users' Guide, FAQ and tutorial all available on the JBoss Cache documentation website. As such, POJO Cache will not be discussed further in this book.

1.2. SUMMARY OF FEATURES

1.2.1. Caching objects

JBoss Cache offers a simple and straightforward API, where data - simple Java objects - can be placed in the cache. Based on configuration options selected, this data may be one or all of:

- cached in-memory for efficient, thread-safe retrieval
- replicated to some or all cache instances in a cluster
- persisted to disk and/or a remote, in-memory cache cluster ("far-cache")
- garbage collected from memory when memory runs low, and passivated to disk so state isn't lost

In addition, JBoss Cache offers a rich set of enterprise-class features:

- being able to participate in JTA transactions (works with most Java EE compliant transaction managers).
- attach to JMX consoles and provide runtime statistics on the state of the cache.

- allow client code to attach listeners and receive notifications on cache events.
- allow grouping of cache operations into batches, for efficient replication

1.2.2. Local and clustered modes

The cache is organized as a tree, with a single root. Each node in the tree essentially contains a map, which acts as a store for key/value pairs. The only requirement placed on objects that are cached is that they implement `java.io.Serializable`.

JBoss Cache can be either local or replicated. Local caches exist only within the scope of the JVM in which they are created, whereas replicated caches propagate any changes to some or all other caches in the same cluster. A cluster may span different hosts on a network or just different JVMs on a single host.

1.2.3. Clustered caches and transactions

When a change is made to an object in the cache and that change is done in the context of a transaction, the replication of changes is deferred until the transaction completes successfully. All modifications are kept in a list associated with the transaction of the caller. When the transaction commits, changes are replicated. Otherwise, on a rollback, we simply undo the changes locally and discard the modification list, resulting in zero network traffic and overhead. For example, if a caller makes 100 modifications and then rolls back the transaction, nothing is replicated, resulting in no network traffic.

If a caller has no transaction or batch associated with it, modifications are replicated immediately. E.g. in the example used earlier, 100 messages would be broadcast for each modification. In this sense, running without a batch or transaction can be thought of as analogous as running with auto-commit switched on in JDBC terminology, where each operation is committed automatically and immediately.

JBoss Cache works out of the box with most popular transaction managers, and even provides an API where custom transaction manager lookups can be written.

All of the above holds true for batches as well, which has similar behavior.

1.2.4. Thread safety

The cache is completely thread-safe. It employs multi-versioned concurrency control (MVCC) to ensure thread safety between readers and writers, while maintaining a high degree of concurrency. The specific MVCC implementation used in JBoss Cache allows for reader threads to be completely free of locks and synchronized blocks, ensuring a very high degree of performance for read-heavy applications. It also uses custom, highly performant lock implementations that employ modern compare-and-swap techniques for writer threads, which are tuned to multi-core CPU architectures.

Multi-versioned concurrency control (MVCC) is the default locking scheme since JBoss Cache 3.x. Optimistic and pessimistic locking schemes from older versions of JBoss Cache are still available but are deprecated in favor of MVCC, and will be removed in future releases. Use of these deprecated locking schemes are strongly discouraged.

The JBoss Cache MVCC implementation only supports READ_COMMITTED and REPEATABLE_READ isolation levels, corresponding to their database equivalents. See the section on Chapter 11, *Transactions and Concurrency* for details on MVCC.

1.3. REQUIREMENTS

JBoss Cache requires a Java 5.0 (or newer) compatible virtual machine and set of libraries, and is developed and tested on Sun's JDK 5.0 and JDK 6.

In addition to Java 5.0, at a minimum, JBoss Cache has dependencies on JGroups, and Apache's commons-logging. JBoss Cache ships with all dependent libraries necessary to run out of the box, as well as several optional jars for optional features.

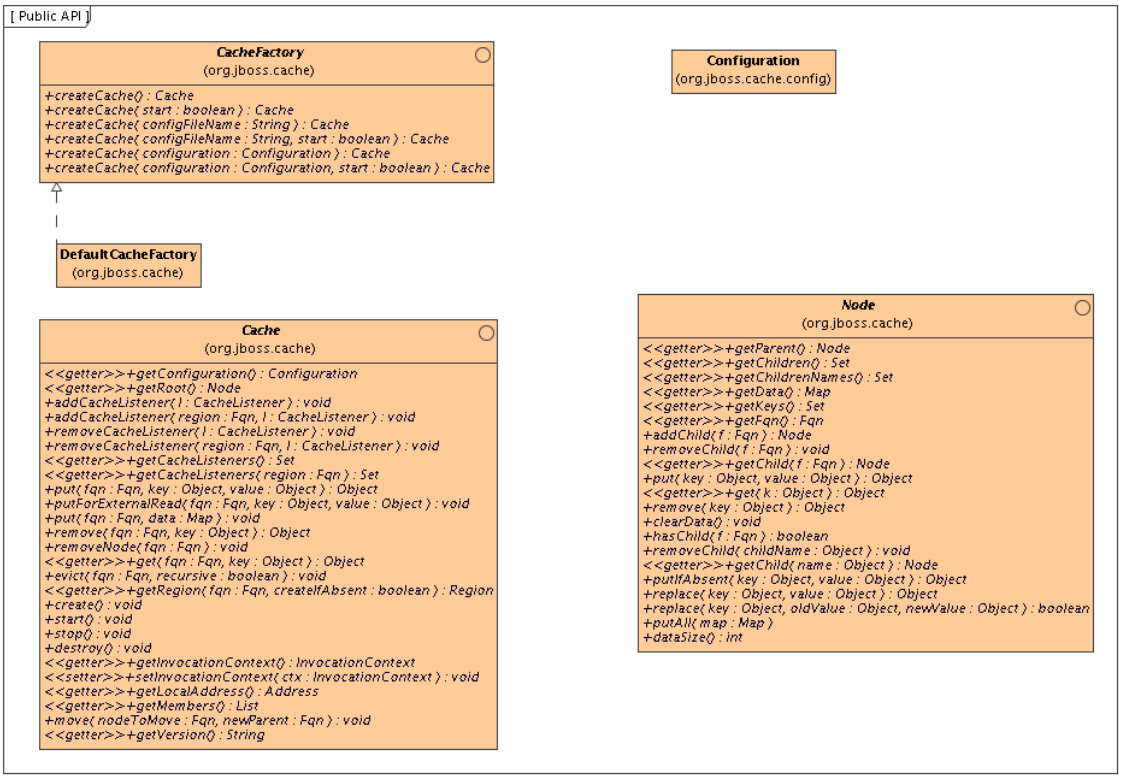
1.4. LICENSE

JBoss Cache is an open source project, using the business and OEM-friendly OSI-approved LGPL license. Commercial development support, production support and training for JBoss Cache is available through JBoss, a division of Red Hat Inc.

CHAPTER 2. USER API

2.1. API CLASSES

The **Cache** interface is the primary mechanism for interacting with JBoss Cache. It is constructed and optionally started using the **CacheFactory**. The **CacheFactory** allows you to create a **Cache** either from a **Configuration** object or an XML file. The cache organizes data into a tree structure, made up of nodes. Once you have a reference to a **Cache**, you can use it to look up **Node** objects in the tree structure, and store data in the tree.



Note that the diagram above only depicts some of the more popular API methods. Reviewing the Javadoc for the above interfaces is the best way to learn the API. Below, we cover some of the main points.

2.2. INSTANTIATING AND STARTING THE CACHE

An instance of the **Cache** interface can only be created via a **CacheFactory**. This is unlike JBoss Cache 1.x, where an instance of the old **TreeCache** class could be directly instantiated.

The **CacheFactory** provides a number of overloaded methods for creating a **Cache**, but they all fundamentally do the same thing:

- Gain access to a **Configuration**, either by having one passed in as a method parameter or by parsing XML content and constructing one. The XML content can come from a provided input stream, from a classpath or filesystem location. See the Chapter 3, *Configuration* for more on obtaining a **Configuration**.
- Instantiate the **Cache** and provide it with a reference to the **Configuration**.
- Optionally invoke the cache's **create()** and **start()** methods.

Here is an example of the simplest mechanism for creating and starting a cache, using the default configuration values:

```
    CacheFactory factory = new DefaultCacheFactory();
    Cache cache = factory.createCache();
```

In this example, we tell the **`CacheFactory`** to find and parse a configuration file on the classpath:

```
    CacheFactory factory = new DefaultCacheFactory();
    Cache cache = factory.createCache("cache-configuration.xml");
```

In this example, we configure the cache from a file, but want to programatically change a configuration element. So, we tell the factory not to start the cache, and instead do it ourselves:

```
    CacheFactory factory = new DefaultCacheFactory();
    Cache cache = factory.createCache("/opt/configurations/cache-
configuration.xml", false);
    Configuration config = cache.getConfiguration();
    config.setClusterName(this.getClusterName());

    // Have to create and start cache before using it
    cache.create();
    cache.start();
```

2.3. CACHING AND RETRIEVING DATA

Next, lets use the **`Cache`** API to access a **`Node`** in the cache and then do some simple reads and writes to that node.

```
    // Let's get a hold of the root node.
    Node rootNode = cache.getRoot();

    // Remember, JBoss Cache stores data in a tree structure.
    // All nodes in the tree structure are identified by Fqn objects.
    Fqn peterGriffinFqn = Fqn.fromString("/griffin/peter");

    // Create a new Node
    Node peterGriffin = rootNode.addChild(peterGriffinFqn);

    // let's store some data in the node
    peterGriffin.put("isCartoonCharacter", Boolean.TRUE);
    peterGriffin.put("favoriteDrink", new Beer());

    // some tests (just assume this code is in a JUnit test case)
    assertTrue(peterGriffin.get("isCartoonCharacter"));
    assertEquals(peterGriffinFqn, peterGriffin.getFqn());
    assertTrue(rootNode.hasChild(peterGriffinFqn));

    Set keys = new HashSet();
    keys.add("isCartoonCharacter");
    keys.add("favoriteDrink");

    assertEquals(keys, peterGriffin.getKeys());
```

```
// let's remove some data from the node
peterGriffin.remove("favoriteDrink");

assertNull(peterGriffin.get("favoriteDrink");

// let's remove the node altogether
rootNode.removeChild(peterGriffinFqn);

assertFalse(rootNode.hasChild(peterGriffinFqn));
```

The **Cache** interface also exposes put/get/remove operations that take an Section 2.4, " The **Fqn** Class " as an argument, for convenience:

```
Fqn peterGriffinFqn = Fqn.fromString("/griffin/peter");

cache.put(peterGriffinFqn, "isCartoonCharacter", Boolean.TRUE);
cache.put(peterGriffinFqn, "favoriteDrink", new Beer());

assertTrue(peterGriffin.get(peterGriffinFqn, "isCartoonCharacter"));
assertTrue(cache.getRootNode().hasChild(peterGriffinFqn));

cache.remove(peterGriffinFqn, "favoriteDrink");

assertNull(cache.get(peterGriffinFqn, "favoriteDrink");

cache.removeNode(peterGriffinFqn);

assertFalse(cache.getRootNode().hasChild(peterGriffinFqn));
```

2.3.1. Organizing Your Data and Using the Node Structure

A Node should be viewed as a named logical grouping of data. A node should be used to contain data for a single data record, for example information about a particular person or account. It should be kept in mind that all aspects of the cache - locking, cache loading, replication and eviction - happen on a per-node basis. As such, anything grouped together by being stored in a single node will be treated as a single atomic unit.

2.4. THE FQN CLASS

The previous section used the **Fqn** class in its examples; now let's learn a bit more about that class.

A Fully Qualified Name (Fqn) encapsulates a list of names which represent a path to a particular location in the cache's tree structure. The elements in the list are typically **String**s but can be any **Object** or a mix of different types.

This path can be absolute (i.e., relative to the root node), or relative to any node in the cache. Reading the documentation on each API call that makes use of **Fqn** will tell you whether the API expects a relative or absolute **Fqn**.

The **Fqn** class provides are variety of factory methods; see the Javadoc for all the possibilities. The following illustrates the most commonly used approaches to creating an Fqn:

```
// Create an Fqn pointing to node 'Joe' under parent node 'Smith'
// under the 'people' section of the tree
```

```
// Parse it from a String
Fqn abc = Fqn.fromString("/people/Smith/Joe/");

// Here we want to use types other than String
Fqn acctFqn = Fqn.fromElements("accounts", "NY", new Integer(12345));
```

Note that

```
Fqn f = Fqn.fromElements("a", "b", "c");
```

is the same as

```
Fqn f = Fqn.fromString("/a/b/c");
```

2.5. STOPPING AND DESTROYING THE CACHE

It is good practice to stop and destroy your cache when you are done using it, particularly if it is a clustered cache and has thus used a JGroups channel. Stopping and destroying a cache ensures resources like network sockets and maintenance threads are properly cleaned up.

```
cache.stop();
cache.destroy();
```

Not also that a cache that has had **`stop()`** invoked on it can be started again with a new call to **`start()`** . Similarly, a cache that has had **`destroy()`** invoked on it can be created again with a new call to **`create()`** (and then started again with a **`start()`** call).

2.6. CACHE MODES

Although technically not part of the API, the *mode* in which the cache is configured to operate affects the cluster-wide behavior of any **`put`** or **`remove`** operation, so we'll briefly mention the various modes here.

JBoss Cache modes are denoted by the **`org.jboss.cache.config.Configuration.CacheMode`** enumeration. They consist of:

- *LOCAL* - local, non-clustered cache. Local caches don't join a cluster and don't communicate with other caches in a cluster.
- *REPL_SYNC* - synchronous replication. Replicated caches replicate all changes to the other caches in the cluster. Synchronous replication means that changes are replicated and the caller blocks until replication acknowledgements are received.
- *REPL_ASYNC* - asynchronous replication. Similar to REPL_SYNC above, replicated caches replicate all changes to the other caches in the cluster. Being asynchronous, the caller does not block until replication acknowledgements are received.
- *INVALIDATION_SYNC* - if a cache is configured for invalidation rather than replication, every time data is changed in a cache other caches in the cluster receive a message informing them that their data is now stale and should be evicted from memory. This reduces replication overhead while still being able to invalidate stale data on remote caches.
- *INVALIDATION_ASYNC* - as above, except this invalidation mode causes invalidation messages to be broadcast asynchronously.

See the Chapter 8, *Cache Modes and Clustering* for more details on how cache mode affects behavior. See the Chapter 3, *Configuration* for info on how to configure things like cache mode.

2.7. ADDING A CACHE LISTENER - REGISTERING FOR CACHE EVENTS

JBoss Cache provides a convenient mechanism for registering notifications on cache events.

```
Object myListener = new MyCacheListener();
cache.addCacheListener(myListener);
```

Similar methods exist for removing or querying registered listeners. See the Javadocs on the **`Cache`** interface for more details.

Basically any public class can be used as a listener, provided it is annotated with the **`@CacheListener`** annotation. In addition, the class needs to have one or more methods annotated with one of the method-level annotations (in the **`org.jboss.cache.notifications.annotation`** package). Methods annotated as such need to be public, have a void return type, and accept a single parameter of type **`org.jboss.cache.notifications.event.Event`** or one of its subtypes.

- **`@CacheStarted`** - methods annotated such receive a notification when the cache is started. Methods need to accept a parameter type which is assignable from **`CacheStartedEvent`** .
- **`@CacheStopped`** - methods annotated such receive a notification when the cache is stopped. Methods need to accept a parameter type which is assignable from **`CacheStoppedEvent`** .
- **`@NodeCreated`** - methods annotated such receive a notification when a node is created. Methods need to accept a parameter type which is assignable from **`NodeCreatedEvent`** .
- **`@NodeRemoved`** - methods annotated such receive a notification when a node is removed. Methods need to accept a parameter type which is assignable from **`NodeRemovedEvent`** .
- **`@NodeModified`** - methods annotated such receive a notification when a node is modified. Methods need to accept a parameter type which is assignable from **`NodeModifiedEvent`** .
- **`@NodeMoved`** - methods annotated such receive a notification when a node is moved. Methods need to accept a parameter type which is assignable from **`NodeMovedEvent`** .
- **`@NodeVisited`** - methods annotated such receive a notification when a node is started. Methods need to accept a parameter type which is assignable from **`NodeVisitedEvent`** .
- **`@NodeLoaded`** - methods annotated such receive a notification when a node is loaded from a **`CacheLoader`** . Methods need to accept a parameter type which is assignable from **`NodeLoadedEvent`** .
- **`@NodeEvicted`** - methods annotated such receive a notification when a node is evicted from memory. Methods need to accept a parameter type which is assignable from **`NodeEvictedEvent`** .
- **`@NodeInvalidated`** - methods annotated such receive a notification when a node is evicted from memory due to a remote invalidation event. Methods need to accept a parameter type which is assignable from **`NodeInvalidatedEvent`** .

- **@NodeActivated** - methods annotated such receive a notification when a node is activated. Methods need to accept a parameter type which is assignable from **NodeActivatedEvent** .
- **@NodePassivated** - methods annotated such receive a notification when a node is passivated. Methods need to accept a parameter type which is assignable from **NodePassivatedEvent** .
- **@TransactionRegistered** - methods annotated such receive a notification when the cache registers a **javax.transaction.Synchronization** with a registered transaction manager. Methods need to accept a parameter type which is assignable from **TransactionRegisteredEvent** .
- **@TransactionCompleted** - methods annotated such receive a notification when the cache receives a commit or rollback call from a registered transaction manager. Methods need to accept a parameter type which is assignable from **TransactionCompletedEvent** .
- **@ViewChanged** - methods annotated such receive a notification when the group structure of the cluster changes. Methods need to accept a parameter type which is assignable from **ViewChangedEvent** .
- **@CacheBlocked** - methods annotated such receive a notification when the cluster requests that cache operations are blocked for a state transfer event. Methods need to accept a parameter type which is assignable from **CacheBlockedEvent** .
- **@CacheUnblocked** - methods annotated such receive a notification when the cluster requests that cache operations are unblocked after a state transfer event. Methods need to accept a parameter type which is assignable from **CacheUnblockedEvent** .
- **@BuddyGroupChanged** - methods annotated such receive a notification when a node changes its buddy group, perhaps due to a buddy falling out of the cluster or a newer, closer buddy joining. Methods need to accept a parameter type which is assignable from **BuddyGroupChangedEvent**.

Refer to the Javadocs on the annotations as well as the **Event** subtypes for details of what is passed in to your method, and when.

Example:

```
@CacheListener
public class MyListener
{

   @CacheStarted
   @CacheStopped
   public void cacheStartStopEvent(Event e)
   {
      switch (e.getType())
      {
         case CACHE_STARTED:
            System.out.println("Cache has started");
            break;
         case CACHE_STOPPED:
            System.out.println("Cache has stopped");
            break;
      }
   }
```

```
    @NodeCreated
    @NodeRemoved
    @NodeVisited
    @NodeModified
    @NodeMoved
    public void logNodeEvent(NodeEvent ne)
    {
       log("An event on node " + ne.getFqn() + " has occured");
    }
}
```

2.7.1. Synchronous and Asynchronous Notifications

By default, all notifications are synchronous, in that they happen on the thread of the caller which generated the event. As such, it is good practise to ensure cache listener implementations don't hold up the thread in long-running tasks. Alternatively, you could set the **`CacheListener.sync`** attribute to **`false`**, in which case you will not be notified in the caller's thread. See the Table 12.13, “The **`<listeners />`** Element” on tuning this thread pool and size of blocking queue.

2.8. USING CACHE LOADERS

Cache loaders are an important part of JBoss Cache. They allow persistence of nodes to disk or to remote cache clusters, and allow for passivation when caches run out of memory. In addition, cache loaders allow JBoss Cache to perform 'warm starts', where in-memory state can be preloaded from persistent storage. JBoss Cache ships with a number of cache loader implementations.

- **`org.jboss.cache.loader.FileCacheLoader`** - a basic, filesystem based cache loader that persists data to disk. Non-transactional and not very performant, but a very simple solution. Used mainly for testing and not recommended for production use.
- **`org.jboss.cache.loader.JDBCCacheLoader`** - uses a JDBC connection to store data. Connections could be created and maintained in an internal pool (uses the c3p0 pooling library) or from a configured DataSource. The database this CacheLoader connects to could be local or remotely located.
- **`org.jboss.cache.loader.BdbjeCacheLoader`** - uses Oracle's BerkeleyDB file-based transactional database to persist data. Transactional and very performant, but potentially restrictive license.
- **`org.jboss.cache.loader.JdbmCacheLoader`** - an open source alternative to the BerkeleyDB.
- **`org.jboss.cache.loader.tcp.TcpCacheLoader`** - uses a TCP socket to "persist" data to a remote cluster, using a "far cache" pattern.
- **`org.jboss.cache.loader.ClusteredCacheLoader`** - used as a "read-only" cache loader, where other nodes in the cluster are queried for state. Useful when full state transfer is too expensive and it is preferred that state is lazily loaded.

These cache loaders, along with advanced aspects and tuning issues, are discussed in the Chapter 9, *Cache Loaders*.

2.9. USING EVICTION POLICIES

Eviction policies are the counterpart to cache loaders. They are necessary to make sure the cache does not run out of memory and when the cache starts to fill, an eviction algorithm running in a separate thread evicts in-memory state and frees up memory. If configured with a cache loader, the state can then be retrieved from the cache loader if needed.

Eviction policies can be configured on a per-region basis, so different subtrees in the cache could have different eviction preferences. JBoss Cache ships with several eviction policies:

- **`org.jboss.cache.eviction.LRUPolicy`** - an eviction policy that evicts the least recently used nodes when thresholds are hit.
- **`org.jboss.cache.eviction.LFUPolicy`** - an eviction policy that evicts the least frequently used nodes when thresholds are hit.
- **`org.jboss.cache.eviction.MRUPolicy`** - an eviction policy that evicts the most recently used nodes when thresholds are hit.
- **`org.jboss.cache.eviction.FIFOPolicy`** - an eviction policy that creates a first-in-first-out queue and evicts the oldest nodes when thresholds are hit.
- **`org.jboss.cache.eviction.ExpirationPolicy`** - an eviction policy that selects nodes for eviction based on an expiry time each node is configured with.
- **`org.jboss.cache.eviction.ElementSizePolicy`** - an eviction policy that selects nodes for eviction based on the number of key/value pairs held in the node.

Detailed configuration and implementing custom eviction policies are discussed in the Chapter 10, *Eviction*.

CHAPTER 3. CONFIGURATION

3.1. CONFIGURATION OVERVIEW

The **`org.jboss.cache.config.Configuration`** class (along with its Section 3.3, “Composition of a **`Configuration`** Object ”) is a Java Bean that encapsulates the configuration of the **`Cache`** and all of its architectural elements (cache loaders, evictions policies, etc.)

The **`Configuration`** exposes numerous properties which are summarized in the Section 12.2, “Configuration File Quick Reference” section of this book and many of which are discussed in later chapters. Any time you see a configuration option discussed in this book, you can assume that the **`Configuration`** class or one of its component parts exposes a simple property setter/getter for that configuration option.

3.2. CREATING A `CONFIGURATION`

As discussed in the Section 2.2, “Instantiating and Starting the Cache”, before a **`Cache`** can be created, the **`CacheFactory`** must be provided with a **`Configuration`** object or with a file name or input stream to use to parse a **`Configuration`** from XML. The following sections describe how to accomplish this.

3.2.1. Parsing an XML-based Configuration File

The most convenient way to configure JBoss Cache is via an XML file. The JBoss Cache distribution ships with a number of configuration files for common use cases. It is recommended that these files be used as a starting point, and tweaked to meet specific needs.

The simplest example of a configuration XML file, a cache configured to run in LOCAL mode, looks like this:

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

<jbosscache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xmlns="urn:jboss:jbosscache-core:config:3.1">
</jbosscache>
```

This file uses sensible defaults for isolation levels, lock acquisition timeouts, locking modes, etc. Another, more complete, sample XML file is included in the Section 12.1, “Sample XML Configuration File” section of this book, along with Section 12.2, “Configuration File Quick Reference” explaining the various options.

3.2.2. Validating Configuration Files

By default JBoss Cache will validate your XML configuration file against an XML schema and throw an exception if the configuration is invalid. This can be overridden with the **`-Djbosscache.config.validate=false`** JVM parameter. Alternately, you could specify your own schema to validate against, using the **`-Djbosscache.config.schemaLocation=url`** parameter.

By default though, configuration files are validated against the JBoss Cache configuration schema, which is included in the **`jbosscache-core.jar`** or on **`http://www.jboss.org/jbosscache/jbosscache-config-3.0.xsd`**. Most XML editing tools

can be used with this schema to ensure the configuration file you create is correct and valid.

3.2.3. Programmatic Configuration

In addition to the XML-based configuration above, the **`Configuration`** can be built up programatically, using the simple property mutators exposed by **`Configuration`** and its components. When constructed, the **`Configuration`** object is preset with JBoss Cache defaults and can even be used as-is for a quick start.

```
Configuration config = new Configuration();
config.setTransactionManagerLookupClass(
  GenericTransactionManagerLookup.class.getName()
  );
config.setIsolationLevel(IsolationLevel.READ_COMMITTED);
config.setCacheMode(CacheMode.LOCAL);
config.setLockAcquisitionTimeout(15000);

CacheFactory factory = new DefaultCacheFactory();
Cache cache = factory.createCache(config);
```

Even the above fairly simple configuration is pretty tedious programming; hence the preferred use of XML-based configuration. However, if your application requires it, there is no reason not to use XML-based configuration for most of the attributes, and then access the **`Configuration`** object to programatically change a few items from the defaults, add an eviction region, etc.

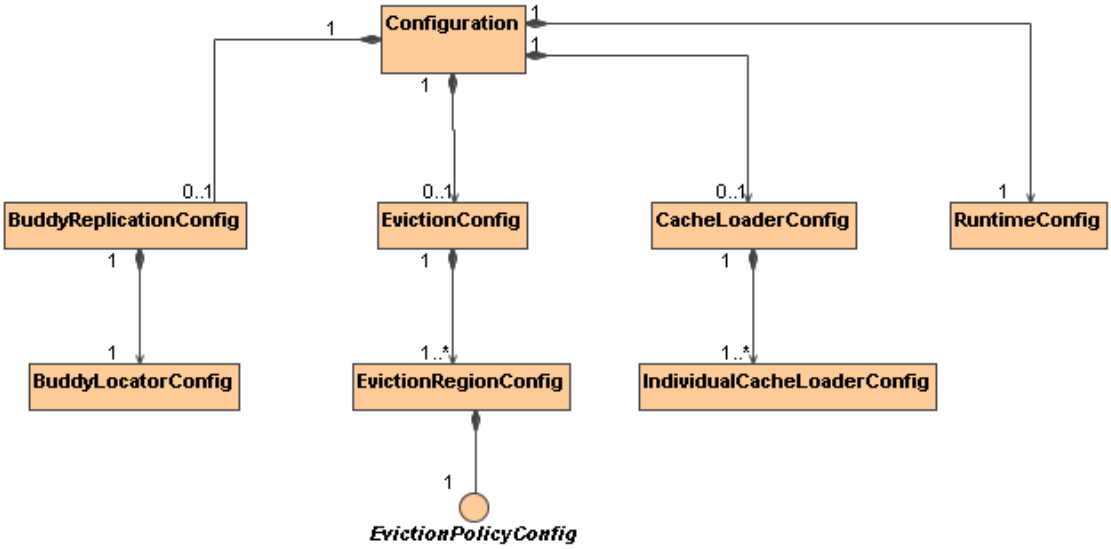
Note that configuration values may not be changed programmatically when a cache is running, except those annotated as **`@Dynamic`** . Dynamic properties are also marked as such in the Section 12.2, “Configuration File Quick Reference” table. Attempting to change a non-dynamic property will result in a **`ConfigurationException`** .

3.2.4. Using an IOC Framework

The **`Configuration`** class and its Section 3.3, “Composition of a **`Configuration`** Object ” are all Java Beans that expose all configuration elements via simple setters and getters. Therefore, any good IOC framework such as JBoss Microcontainer should be able to build up a **`Configuration`** from an XML file in the framework's own format. See the Section 5.2, “Via JBoss Microcontainer (JBoss AS 5.x)” section for an example of this.

3.3. COMPOSITION OF A `CONFIGURATION` OBJECT

A **`Configuration`** is composed of a number of subobjects:



Following is a brief overview of the components of a **`Configuration`** . See the Javadoc and the linked chapters in this book for a more complete explanation of the configurations associated with each component.

- **`Configuration`** : top level object in the hierarchy; exposes the configuration properties listed in the Section 12.2, "Configuration File Quick Reference" section of this book.
- **`BuddyReplicationConfig`** : only relevant if Section 8.1.2.2, "Buddy Replication" is used. General buddy replication configuration options. Must include a:
- **`BuddyLocatorConfig`** : implementation-specific configuration object for the **`BuddyLocator`** implementation being used. What configuration elements are exposed depends on the needs of the **`BuddyLocator`** implementation.
- **`EvictionConfig`** : only relevant if Chapter 10, *Eviction* is used. General eviction configuration options. Must include at least one:
- **`EvictionRegionConfig`** : one for each eviction region; names the region, etc. Must include a:
- **`EvictionAlgorithmConfig`** : implementation-specific configuration object for the **`EvictionAlgorithm`** implementation being used. What configuration elements are exposed depends on the needs of the **`EvictionAlgorithm`** implementation.
- **`CacheLoaderConfig`** : only relevant if a Chapter 9, *Cache Loaders* is used. General cache loader configuration options. Must include at least one:
- **`IndividualCacheLoaderConfig`** : implementation-specific configuration object for the **`CacheLoader`** implementation being used. What configuration elements are exposed depends on the needs of the **`CacheLoader`** implementation.
- **`RuntimeConfig`** : exposes to cache clients certain information about the cache's runtime environment (e.g. membership in buddy replication groups if Section 8.1.2.2, "Buddy Replication" is used.) Also allows direct injection into the cache of needed external services like a JTA **`TransactionManager`** or a JGroups **`ChannelFactory`** .

3.4. DYNAMIC RECONFIGURATION

Dynamically changing the configuration of *some* options while the cache is running is supported, by programmatically obtaining the **`Configuration`** object from the running cache and changing values. E.g.,

```
Configuration liveConfig = cache.getConfiguration();
liveConfig.setLockAcquisitionTimeout(2000);
```

A complete listing of which options may be changed dynamically is in the Section 12.2, "Configuration File Quick Reference" section. An **`org.jboss.cache.config.ConfigurationException`** will be thrown if you attempt to change a setting that is not dynamic.

3.4.1. Overriding the Configuration via the Option API

The Option API allows you to override certain behaviors of the cache on a per invocation basis. This involves creating an instance of **`org.jboss.cache.config.Option`** , setting the options you wish to override on the **`Option`** object and passing it in the **`InvocationContext`** before invoking your method on the cache.

E.g., to force a write lock when reading data (when used in a transaction, this provides semantics similar to SELECT FOR UPDATE in a database)

```
      // first start a transaction

cache.getInvocationContext().getOptionOverrides().setForceWriteLock(true);
      Node n = cache.getNode(Fqn.fromString("/a/b/c"));
      // make changes to the node
      // commit transaction
```

E.g., to suppress replication of a put call in a REPL_SYNC cache:

```
      Node node = cache.getChild(Fqn.fromString("/a/b/c"));

cache.getInvocationContext().getOptionOverrides().setLocalOnly(true);
      node.put("localCounter", new Integer(2));
```

See the Javadocs on the **`Option`** class for details on the options available.

CHAPTER 4. BATCHING API

4.1. INTRODUCTION

The batching API, introduced in JBoss Cache 3.x, is intended as a mechanism to batch the way calls are replicated independent of JTA transactions.

This is useful when you want to batch up replication calls within a scope finer than that of any ongoing JTA transactions.

4.2. CONFIGURING BATCHING

To use batching, you need to enable invocation batching in your cache configuration, either on the **`Configuration`** object:

```
    Configuration.setInvocationBatchingEnabled(true);
```

or in your XML file:

```
    <invocationBatching enabled="true"/>
```

By default, invocation batching is disabled. Note that you do *not* have to have a transaction manager defined to use batching.

4.3. BATCHING API

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

```
   Cache cache = getCache();

   // not using a batch
   cache.put("/a", "key", "value"); // will replicate immediately

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

   cache.startBatch();
   cache.put("/a", "key", "value");
   cache.put("/b", "key", "value");
   cache.put("/c", "key", "value");
   cache.endBatch(false); // This will "discard" changes made in the batch
```

CHAPTER 5. DEPLOYING JBOSS CACHE

5.1. STANDALONE USE/PROGRAMATIC DEPLOYMENT

When used in a standalone Java program, all that needs to be done is to instantiate the cache using the **`CacheFactory`** and a **`Configuration`** instance or an XML file, as discussed in the Section 2.2, "Instantiating and Starting the Cache" and Section 3.2, "Creating a **`Configuration`** " chapters.

The same techniques can be used when an application running in an application server wishes to programatically deploy a cache rather than relying on an application server's deployment features. An example of this would be a webapp deploying a cache via a **`javax.servlet.ServletContextListener`**.

After creation, you could share your cache instance among different application components either by using an IOC container such as JBoss Microcontainer, or by binding it to JNDI, or simply holding a static reference to the cache.

If, after deploying your cache you wish to expose a management interface to it in JMX, see Section 5.4.2, "Registering the CacheJmxWrapper with the MBeanServer".

5.2. VIA JBOSS MICROCONTAINER (JBOSS AS 5.X)

Beginning with AS 5, JBoss AS supports deployment of POJO services via deployment of a file whose name ends with **`-jboss-beans.xml`**. A POJO service is one whose implementation is via a "Plain Old Java Object", meaning a simple java bean that isn't required to implement any special interfaces or extend any particular superclass. A **`Cache`** is a POJO service, and all the components in a **`Configuration`** are also POJOs, so deploying a cache in this way is a natural step.

Deployment of the cache is done using the JBoss Microcontainer that forms the core of JBoss AS. JBoss Microcontainer is a sophisticated IOC framework similar to Spring. A **`-jboss-beans.xml`** file is basically a descriptor that tells the IOC framework how to assemble the various beans that make up a POJO service.

For each configurable option exposed by the **`Configuration`** components, a getter/setter must be defined in the configuration class. This is required so that JBoss Microcontainer can, in typical IOC way, call these methods when the corresponding properties have been configured.

You need to ensure that the **`jbosscache-core.jar`** and **`jgroups.jar`** libraries are in your server's **`lib`** directory. This is usually the case when you use JBoss AS in its **`all`** configuration. Note that you will have to bring in any optional jars you require, such as **`jdbm.jar`** based on your cache configuration.

The following is an example **`-beans.xml`** file. If you look in the **`server/all/deploy`** directory of a JBoss AS 5 installation, you can find several more examples.

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

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- First we create a Configuration object for the cache -->
   <bean name="ExampleCacheConfig"
      class="org.jboss.cache.config.Configuration">

      <!-- Externally injected services -->
      <property name="runtimeConfig">
```

```
         <bean class="org.jboss.cache.config.RuntimeConfig">
            <property name="transactionManager">
               <inject bean="jboss:service=TransactionManager"
                       property="TransactionManager"/>
            </property>
            <property name="muxChannelFactory"><inject
bean="JChannelFactory"/></property>
         </bean>
      </property>

      <property name="multiplexerStack">udp</property>

      <property name="clusterName">Example-EntityCache</property>

      <property name="isolationLevel">REPEATABLE_READ</property>

      <property name="cacheMode">REPL_SYNC</property>

      <property name="stateRetrievalTimeout">15000</property>

      <property name="syncReplTimeout">20000</property>

      <property name="lockAcquisitionTimeout">15000</property>

      <property name="exposeManagementStatistics">true</property>
   </bean>

   <!-- Factory to build the Cache. -->
   <bean name="DefaultCacheFactory"
class="org.jboss.cache.DefaultCacheFactory">
      <constructor factoryClass="org.jboss.cache.DefaultCacheFactory"
                   factoryMethod="getInstance" />
   </bean>

   <!-- The cache itself -->
   <bean name="ExampleCache" class="org.jboss.cache.Cache">

      <constructor factoryMethod="createCache">
          <factory bean="DefaultCacheFactory"/>
          <parameter class="org.jboss.cache.config.Configuration"><inject
bean="ExampleCacheConfig"/></parameter>
          <parameter class="boolean">false</parameter>
      </constructor>

   </bean>

</deployment>
```

See the JBoss Microcontainer documentation for details on the above syntax. Basically, each **bean** element represents an object and is used to create a **Configuration** and its Section 3.3, “Composition of a **Configuration** Object ” The **DefaultCacheFactory** bean constructs the cache, conceptually doing the same thing as is shown in the Section 2.2, “Instantiating and Starting the Cache” chapter.

An interesting thing to note in the above example is the use of the **RuntimeConfig** object. External resources like a **TransactionManager** and a JGroups **ChannelFactory** that are visible to the microcontainer are dependency injected into the **RuntimeConfig**. The assumption here is that in some

other deployment descriptor in the AS, the referenced beans have already been described.

5.3. AUTOMATIC BINDING TO JNDI IN JBOSS AS

This feature is not available as of the time of this writing. We will add a wiki page describing how to use it once it becomes available.

5.4. RUNTIME MANAGEMENT INFORMATION

JBoss Cache includes JMX MBeans to expose cache functionality and provide statistics that can be used to analyze cache operations. JBoss Cache can also broadcast cache events as MBean notifications for handling via JMX monitoring tools.

5.4.1. JBoss Cache MBeans

JBoss Cache provides an MBean that can be registered with your environments JMX server to allow access to the cache instance via JMX. This MBean is the **`org.jboss.cache.jmx.CacheJmxWrapper`**. It is a StandardMBean, so its MBean interface is **`org.jboss.cache.jmx.CacheJmxWrapperMBean`**. This MBean can be used to:

- Get a reference to the underlying **`Cache`**.
- Invoke create/start/stop/destroy lifecycle operations on the underlying **`Cache`**.
- Inspect various details about the cache's current state (number of nodes, lock information, etc.)
- See numerous details about the cache's configuration, and change those configuration items that can be changed when the cache has already been started.

See the **`CacheJmxWrapperMBean`** Javadoc for more details.

If a **`CacheJmxWrapper`** is registered, JBoss Cache also provides MBeans for several other internal components and subsystems. These MBeans are used to capture and expose statistics related to the subsystems they represent. They are hierarchically associated with the **`CacheJmxWrapper`** MBean and have service names that reflect this relationship. For example, a replication interceptor MBean for the **`jboss.cache:service=TomcatClusteringCache`** instance will be accessible through the service named **`jboss.cache:service=TomcatClusteringCache,cache-interceptor=ReplicationInterceptor`**.

5.4.2. Registering the CacheJmxWrapper with the MBeanServer

The best way to ensure the **`CacheJmxWrapper`** is registered in JMX depends on how you are deploying your cache.

5.4.2.1. Programatic Registration with a Cache instance

Simplest way to do this is to create your **`Cache`** and pass it to the **`JmxRegistrationManager`** constructor.

```
CacheFactory factory = new DefaultCacheFactory();
// Build but don't start the cache
// (although it would work OK if we started it)
Cache cache = factory.createCache("cache-configuration.xml");
```

```
   MBeanServer server = getMBeanServer(); // however you do it
   ObjectName on = new ObjectName("jboss.cache:service=Cache");

   JmxRegistrationManager jmxManager = new JmxRegistrationManager(server,
cache, on);
   jmxManager.registerAllMBeans();

   ... use the cache

   ... on application shutdown

   jmxManager.unregisterAllMBeans();
   cache.stop();
```

5.4.2.2. Programatic Registration with a Configuration instance

Alternatively, build a **`Configuration`** object and pass it to the **`CacheJmxWrapper`**. The wrapper will construct the **`Cache`** on your behalf.

```
   Configuration config = buildConfiguration(); // whatever it does

   CacheJmxWrapperMBean wrapper = new CacheJmxWrapper(config);
   MBeanServer server = getMBeanServer(); // however you do it
   ObjectName on = new ObjectName("jboss.cache:service=TreeCache");
   server.registerMBean(wrapper, on);

   // Call to wrapper.create() will build the Cache if one wasn't injected
   wrapper.create();
   wrapper.start();

   // Now that it's built, created and started, get the cache from the
wrapper
   Cache cache = wrapper.getCache();

   ... use the cache

   ... on application shutdown

   wrapper.stop();
   wrapper.destroy();
```

5.4.2.3. JMX-Based Deployment in JBoss AS (JBoss AS 5.x)

`CacheJmxWrapper` is a POJO, so the microcontainer has no problem creating one. The trick is getting it to register your bean in JMX. This can be done by specifying the **`org.jboss.aop.microcontainer.aspects.jmx.JMX`** annotation on the **`CacheJmxWrapper`** bean:

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

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- First we create a Configuration object for the cache -->
```

```
   <bean name="ExampleCacheConfig"
      class="org.jboss.cache.config.Configuration">

      ... build up the Configuration

   </bean>

   <!-- Factory to build the Cache. -->
   <bean name="DefaultCacheFactory"
class="org.jboss.cache.DefaultCacheFactory">
      <constructor factoryClass="org.jboss.cache.DefaultCacheFactory"
                   factoryMethod="getInstance" />
   </bean>

   <!-- The cache itself -->
   <bean name="ExampleCache" class="org.jboss.cache.CacheImpl">

      <constructor factoryMethod="createnewInstance">
          <factory bean="DefaultCacheFactory"/>
          <parameter><inject bean="ExampleCacheConfig"/></parameter>
          <parameter>false</parameter>
      </constructor>

   </bean>

   <!-- JMX Management -->
   <bean name="ExampleCacheJmxWrapper"
class="org.jboss.cache.jmx.CacheJmxWrapper">


<annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(name="jboss.cach
e:service=ExampleTreeCache",

exposedInterface=org.jboss.cache.jmx.CacheJmxWrapperMBean.class,
                         registerDirectly=true)</annotation>

      <constructor>
          <parameter><inject bean="ExampleCache"/></parameter>
      </constructor>

   </bean>

</deployment>
```

As discussed in the Section 5.4.2, “Registering the CacheJmxWrapper with the MBeanServer” section, **`CacheJmxWrapper`** can do the work of building, creating and starting the **`Cache`** if it is provided with a **`Configuration`**. With the microcontainer, this is the preferred approach, as it saves the boilerplate XML needed to create the **`CacheFactory`**.

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

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- First we create a Configuration object for the cache -->
   <bean name="ExampleCacheConfig"
      class="org.jboss.cache.config.Configuration">
```

```
      ... build up the Configuration

   </bean>

   <bean name="ExampleCache" class="org.jboss.cache.jmx.CacheJmxWrapper">

      <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX
        (name="jboss.cache:service=ExampleTreeCache",
        exposedInterface=org.jboss.cache.jmx.CacheJmxWrapperMBean.class,
        registerDirectly=true)</annotation>

      <constructor>
          <parameter><inject bean="ExampleCacheConfig"/></parameter>
      </constructor>

   </bean>

</deployment>
```

5.4.3. JBoss Cache Statistics

JBoss Cache captures statistics in its interceptors and various other components, and exposes these statistics through a set of MBeans. Gathering of statistics is enabled by default; this can be disabled for a specific cache instance through the **`Configuration.setExposeManagementStatistics()`** setter. Note that the majority of the statistics are provided by the **`CacheMgmtInterceptor`**, so this MBean is the most significant in this regard. If you want to disable all statistics for performance reasons, you set **`Configuration.setExposeManagementStatistics(false)`** and this will prevent the **`CacheMgmtInterceptor`** from being included in the cache's interceptor stack when the cache is started.

If a **`CacheJmxWrapper`** is registered with JMX, the wrapper also ensures that an MBean is registered in JMX for each interceptor and component that exposes statistics.

NOTE

Note that if the **`CacheJmxWrapper`** is not registered in JMX, the interceptor MBeans will not be registered either. The JBoss Cache 1.4 releases included code that would try to "discover" an **`MBeanServer`** and automatically register the interceptor MBeans with it. For JBoss Cache 2.x we decided that this sort of "discovery" of the JMX environment was beyond the proper scope of a caching library, so we removed this functionality.

. Management tools can then access those MBeans to examine the statistics. See the section in the Section 13.1, “JBoss Cache Statistics” pertaining to the statistics that are made available via JMX.

5.4.4. Receiving JMX Notifications

JBoss Cache users can register a listener to receive cache events described earlier in the Section 2.7, “ Adding a Cache Listener - registering for cache events ” chapter. Users can alternatively utilize the cache's management information infrastructure to receive these events via JMX notifications. Cache events are accessible as notifications by registering a **`NotificationListener`** for the **`CacheJmxWrapper`**.

See the section in the Section 13.2, “JMX MBean Notifications” pertaining to JMX notifications for a list of notifications that can be received through the **`CacheJmxWrapper`**.

The following is an example of how to programmatically receive cache notifications when running in a JBoss AS environment. In this example, the client uses a filter to specify which events are of interest.

```
   MyListener listener = new MyListener();
   NotificationFilterSupport filter = null;

   // get reference to MBean server
   Context ic = new InitialContext();
   MBeanServerConnection server =
(MBeanServerConnection)ic.lookup("jmx/invoker/RMIAdaptor");

   // get reference to CacheMgmtInterceptor MBean
   String cache_service = "jboss.cache:service=TomcatClusteringCache";
   ObjectName mgmt_name = new ObjectName(cache_service);

   // configure a filter to only receive node created and removed events
   filter = new NotificationFilterSupport();
   filter.disableAllTypes();
   filter.enableType(CacheNotificationBroadcaster.NOTIF_NODE_CREATED);
   filter.enableType(CacheNotificationBroadcaster.NOTIF_NODE_REMOVED);

   // register the listener with a filter
   // leave the filter null to receive all cache events
   server.addNotificationListener(mgmt_name, listener, filter, null);

   // ...

   // on completion of processing, unregister the listener
   server.removeNotificationListener(mgmt_name, listener, filter, null);
```

The following is the simple notification listener implementation used in the previous example.

```
   private class MyListener implements NotificationListener, Serializable
   {
      public void handleNotification(Notification notification, Object
handback)
      {
         String message = notification.getMessage();
         String type = notification.getType();
         Object userData = notification.getUserData();

         System.out.println(type + ": " + message);

         if (userData == null)
         {
            System.out.println("notification data is null");
         }
         else if (userData instanceof String)
         {
            System.out.println("notification data: " + (String) userData);
         }
         else if (userData instanceof Object[])
         {
```

```
            Object[] ud = (Object[]) userData;
            for (Object data : ud)
            {
               System.out.println("notification data: " +
data.toString());
            }
         }
         else
         {
            System.out.println("notification data class: " +
userData.getClass().getName());
         }
      }
   }
```

Note that the JBoss Cache management implementation only listens to cache events after a client registers to receive MBean notifications. As soon as no clients are registered for notifications, the MBean will remove itself as a cache listener.

5.4.5. Accessing Cache MBeans in a Standalone Environment using the `jconsole` Utility

JBoss Cache MBeans are easily accessed when running cache instances in an application server that provides an MBean server interface such as JBoss JMX Console. Refer to your server documentation for instructions on how to access MBeans running in a server's MBean container.

In addition, though, JBoss Cache MBeans are also accessible when running in a non-server environment using your JDK's **`jconsole`** tool. When running a standalone cache outside of an application server, you can access the cache's MBeans as follows.

1. Set the system property **`-Dcom.sun.management.jmxremote`** when starting the JVM where the cache will run.

2. Once the JVM is running, start the **`jconsole`** utility, located in your JDK's **`/bin`** directory.

3. When the utility loads, you will be able to select your running JVM and connect to it. The JBoss Cache MBeans will be available on the MBeans panel.

Note that the **`jconsole`** utility will automatically register as a listener for cache notifications when connected to a JVM running JBoss Cache instances.

CHAPTER 6. VERSION COMPATIBILITY AND INTEROPERABILITY

6.1. API COMPATIBILITY

Within a major version, releases of JBoss Cache are meant to be compatible and interoperable. Compatible in the sense that it should be possible to upgrade an application from one version to another by simply replacing jars. Interoperable in the sense that if two different versions of JBoss Cache are used in the same cluster, they should be able to exchange replication and state transfer messages. Note however that interoperability requires use of the same JGroups version in all nodes in the cluster. In most cases, the version of JGroups used by a version of JBoss Cache can be upgraded.

As such, JBoss Cache 2.x.x is not API or binary compatible with prior 1.x.x versions. On the other hand, JBoss Cache 2.1.x will be API and binary compatible with 2.0.x.

We have made best efforts, however, to keep JBoss Cache 3.x both binary and API compatible with 2.x. Still, it is recommended that client code is updated not to use deprecated methods, classes and configuration files.

6.2. WIRE-LEVEL INTEROPERABILITY

A configuration parameter, **`Configuration.setReplicationVersion()`**, is available and is used to control the wire format of inter-cache communications. They can be wound back from more efficient and newer protocols to "compatible" versions when talking to older releases. This mechanism allows us to improve JBoss Cache by using more efficient wire formats while still providing a means to preserve interoperability.

6.3. COMPATIBILITY MATRIX

A compatibility matrix is maintained on the JBoss Cache website, which contains information on different versions of JBoss Cache, JGroups and JBoss Application Server.

PART II. JBOSS CACHE ARCHITECTURE

This section digs deeper into the JBoss Cache architecture, and is meant for developers wishing to use the more advanced cache features,extend or enhance the cache, write plugins, or are just looking for detailed knowledge of how things work under the hood.

CHAPTER 7. ARCHITECTURE

7.1. DATA STRUCTURES WITHIN THE CACHE

A **Cache** consists of a collection of **Node** instances, organised in a tree structure. Each **Node** contains a **Map** which holds the data objects to be cached. It is important to note that the structure is a mathematical tree, and not a graph; each **Node** has one and only one parent, and the root node is denoted by the constant fully qualified name, **Fqn.ROOT**.

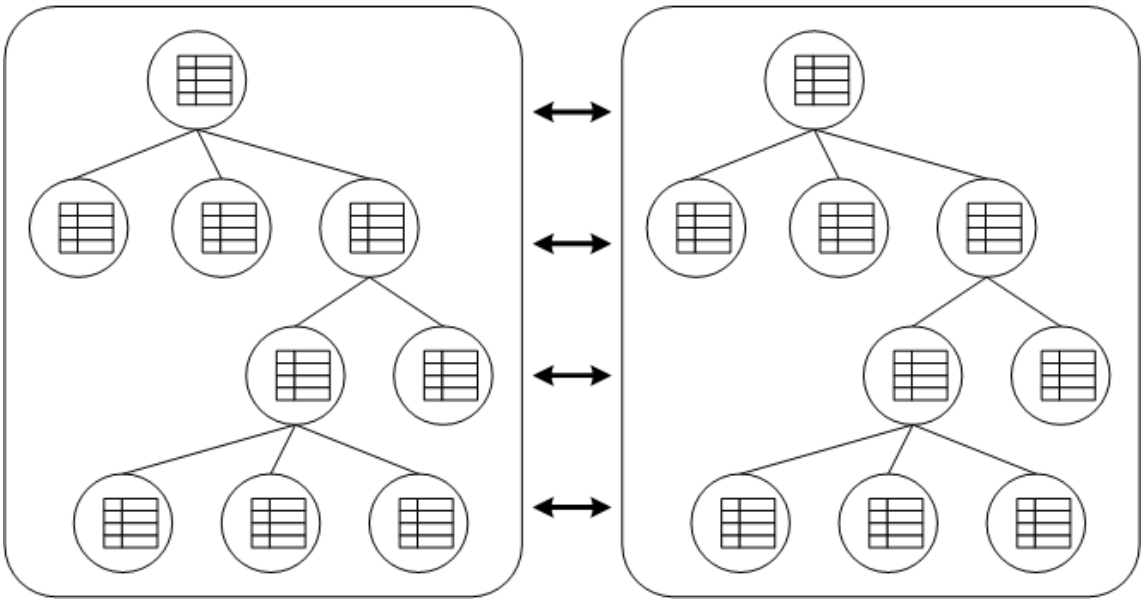


Figure 7.1. Data structured as a tree

In the diagram above, each box represents a JVM. You see 2 caches in separate JVMs, replicating data to each other.

Any modifications (see Chapter 2, *User API*) in one cache instance will be replicated to the other cache. Naturally, you can have more than 2 caches in a cluster. Depending on the transactional settings, this replication will occur either after each modification or at the end of a transaction, at commit time. When a new cache is created, it can optionally acquire the contents from one of the existing caches on startup.

7.2. SPI INTERFACES

In addition to **Cache** and **Node** interfaces, JBoss Cache exposes more powerful **CacheSPI** and **NodeSPI** interfaces, which offer more control over the internals of JBoss Cache. These interfaces are not intended for general use, but are designed for people who wish to extend and enhance JBoss Cache, or write custom **Interceptor** or **CacheLoader** instances.

Figure 7.2. SPI Interfaces

The **`CacheSPI`** interface cannot be created, but is injected into **`Interceptor`** and **`CacheLoader`** implementations by the **`setCache(CacheSPI cache)`** methods on these interfaces. **`CacheSPI`** extends **`Cache`** so all the functionality of the basic API is also available.

Similarly, a **`NodeSPI`** interface cannot be created. Instead, one is obtained by performing operations on **`CacheSPI`**, obtained as above. For example, **`Cache.getRoot() : Node`** is overridden as **`CacheSPI.getRoot() : NodeSPI`**.

It is important to note that directly casting a **`Cache`** or **`Node`** to its SPI counterpart is not recommended and is bad practice, since the inheritace of interfaces it is not a contract that is guaranteed to be upheld moving forward. The exposed public APIs, on the other hand, is guaranteed to be upheld.

7.3. METHOD INVOCATIONS ON NODES

Since the cache is essentially a collection of nodes, aspects such as clustering, persistence, eviction, etc. need to be applied to these nodes when operations are invoked on the cache as a whole or on individual nodes. To achieve this in a clean, modular and extensible manner, an interceptor chain is used. The chain is built up of a series of interceptors, each one adding an aspect or particular functionality. The chain is built when the cache is created, based on the configuration used.

It is important to note that the **`NodeSPI`** offers some methods (such as the **`xxxDirect()`** method

family) that operate on a node directly without passing through the interceptor stack. Plugin authors should note that using such methods will affect the aspects of the cache that may need to be applied, such as locking, replication, etc. To put it simply, don't use such methods unless you *really* know what you're doing!

7.3.1. Interceptors

JBoss Cache essentially is a core data structure - an implementation of **`DataContainer`** - and aspects and features are implemented using interceptors in front of this data structure. A **`CommandInterceptor`** is an abstract class, interceptor implementations extend this.

`CommandInterceptor` implements the **`Visitor`** interface so it is able to alter commands in a strongly typed manner as the command makes its way to the data structure. More on visitors and commands in the next section.

Interceptor implementations are chained together in the **`InterceptorChain`** class, which dispatches a command across the chain of interceptors. A special interceptor, the **`CallInterceptor`**, always sits at the end of this chain to invoke the command being passed up the chain by calling the command's **`process()`** method.

JBoss Cache ships with several interceptors, representing different behavioral aspects, some of which are:

- **`TxInterceptor`** - looks for ongoing transactions and registers with transaction managers to participate in synchronization events
- **`ReplicationInterceptor`** - replicates state across a cluster using the RpcManager class
- **`CacheLoaderInterceptor`** - loads data from a persistent store if the data requested is not available in memory

The interceptor chain configured for your cache instance can be obtained and inspected by calling **`CacheSPI.getInterceptorChain()`**, which returns an ordered **`List`** of interceptors in the order in which they would be encountered by a command.

7.3.1.1. Writing Custom Interceptors

Custom interceptors to add specific aspects or features can be written by extending **`CommandInterceptor`** and overriding the relevant **`visitXXX()`** methods based on the commands you are interested in intercepting. There are other abstract interceptors you could extend instead, such as the **`PrePostProcessingCommandInterceptor`** and the **`SkipCheckChainedInterceptor`**. Please see their respective javadocs for details on the extra features provided.

The custom interceptor will need to be added to the interceptor chain by using the **`Cache.addInterceptor()`** methods. See the javadocs on these methods for details.

Adding custom interceptors via XML is also supported, please see the Chapter 12, *Configuration References* for details.

7.3.2. Commands and Visitors

Internally, JBoss Cache uses a command/visitor pattern to execute API calls. Whenever a method is called on the cache interface, the **`CacheInvocationDelegate`**, which implements the **`Cache`** interface, creates an instance of **`VisitableCommand`** and dispatches this command up a chain of

interceptors. Interceptors, which implement the **`Visitor`** interface, are able to handle **`VisitableCommand`**s they are interested in, and add behavior to the command.

Each command contains all knowledge of the command being executed such as parameters used and processing behavior, encapsulated in a **`process()`** method. For example, the **`RemoveNodeCommand`** is created and passed up the interceptor chain when **`Cache.removeNode()`** is called, and **`RemoveNodeCommand.process()`** has the necessary knowledge of how to remove a node from the data structure.

In addition to being visitable, commands are also replicable. The JBoss Cache marshallers know how to efficiently marshal commands and invoke them on remote cache instances using an internal RPC mechanism based on JGroups.

7.3.3. InvocationContexts

`InvocationContext` holds intermediate state for the duration of a single invocation, and is set up and destroyed by the **`InvocationContextInterceptor`** which sits at the start of the interceptor chain.

`InvocationContext`, as its name implies, holds contextual information associated with a single cache method invocation. Contextual information includes associated **`javax.transaction.Transaction`** or **`org.jboss.cache.transaction.GlobalTransaction`**, method invocation origin (**`InvocationContext.isOriginLocal()`**) as well as Section 3.4.1, “Overriding the Configuration via the Option API”, and information around which nodes have been locked, etc.

The **`InvocationContext`** can be obtained by calling **`Cache.getInvocationContext()`**.

7.4. MANAGERS FOR SUBSYSTEMS

Some aspects and functionality is shared by more than a single interceptor. Some of these have been encapsulated into managers, for use by various interceptors, and are made available by the **`CacheSPI`** interface.

7.4.1. RpcManager

This class is responsible for calls made via the JGroups channel for all RPC calls to remote caches, and encapsulates the JGroups channel used.

7.4.2. BuddyManager

This class manages buddy groups and invokes group organization remote calls to organize a cluster of caches into smaller sub-groups.

7.4.3. CacheLoaderManager

Sets up and configures cache loaders. This class wraps individual **`CacheLoader`** instances in delegating classes, such as **`SingletonStoreCacheLoader`** or **`AsyncCacheLoader`**, or may add the **`CacheLoader`** to a chain using the **`ChainingCacheLoader`**.

7.5. MARSHALLING AND WIRE FORMATS

Early versions of JBoss Cache simply wrote cached data to the network by writing to an **`ObjectOutputStream`** during replication. Over various releases in the JBoss Cache 1.x.x series this approach was gradually deprecated in favor of a more mature marshalling framework. In the JBoss

Cache 2.x.x series, this is the only officially supported and recommended mechanism for writing objects to datastreams.

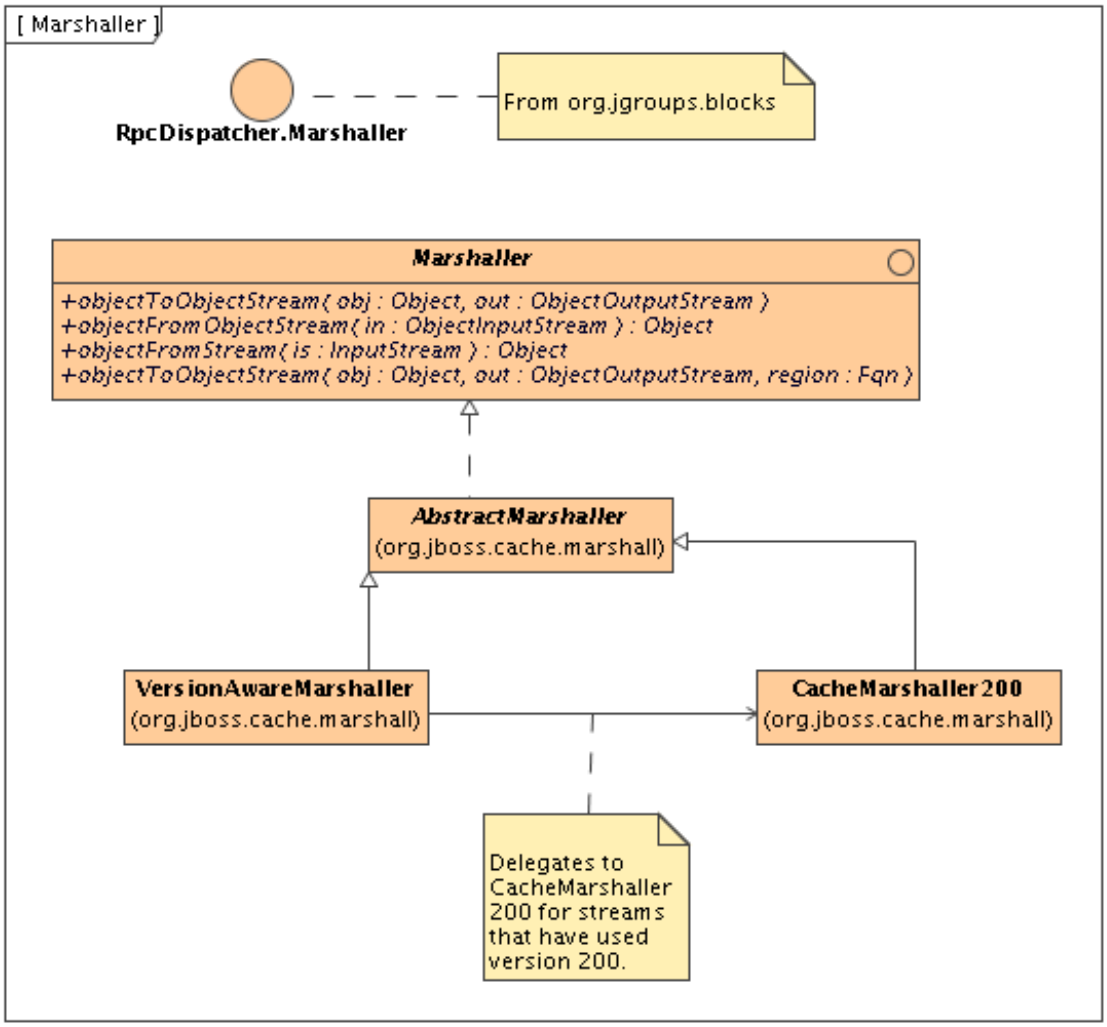


Figure 7.3. The Marshaller interface

7.5.1. The Marshaller Interface

The **`Marshaller`** interface extends **`RpcDispatcher.Marshaller`** from JGroups. This interface has two main implementations - a delegating **`VersionAwareMarshaller`** and a concrete **`CacheMarshaller300`** .

The marshaller can be obtained by calling **`CacheSPI.getMarshaller()`**, and defaults to the **`VersionAwareMarshaller`**. Users may also write their own marshallers by implementing the **`Marshaller`** interface or extending the **`AbstractMarshaller`** class, and adding it to their configuration by using the **`Configuration.setMarshallerClass()`** setter.

7.5.2. VersionAwareMarshaller

As the name suggests, this marshaller adds a version **`short`** to the start of any stream when writing, enabling similar **`VersionAwareMarshaller`** instances to read the version short and know which specific marshaller implementation to delegate the call to. For example, **`CacheMarshaller200`** is the marshaller for JBoss Cache 2.0.x. JBoss Cache 3.0.x ships with **`CacheMarshaller300`** with an

improved wire protocol. Using a **`VersionAwareMarshaller`** helps achieve wire protocol compatibility between minor releases but still affords us the flexibility to tweak and improve the wire protocol between minor or micro releases.

7.6. CLASS LOADING AND REGIONS

When used to cluster state of application servers, applications deployed in the application tend to put instances of objects specific to their application in the cache (or in an **`HttpSession`** object) which would require replication. It is common for application servers to assign separate **`ClassLoader`** instances to each application deployed, but have JBoss Cache libraries referenced by the application server's **`ClassLoader`**.

To enable us to successfully marshall and unmarshall objects from such class loaders, we use a concept called regions. A region is a portion of the cache which share a common class loader (a region also has other uses - see Chapter 10, *Eviction*).

A region is created by using the **`Cache.getRegion(Fqn fqn, boolean createIfNotExists)`** method, and returns an implementation of the **`Region`** interface. Once a region is obtained, a class loader for the region can be set or unset, and the region can be activated/deactivated. By default, regions are active unless the **`InactiveOnStartup`** configuration attribute is set to **`true`**.

CHAPTER 8. CACHE MODES AND CLUSTERING

This chapter talks about aspects around clustering JBoss Cache.

8.1. CACHE REPLICATION MODES

JBoss Cache can be configured to be either local (standalone) or clustered. If in a cluster, the cache can be configured to replicate changes, or to invalidate changes. A detailed discussion on this follows.

8.1.1. Local Mode

Local caches don't join a cluster and don't communicate with other caches in a cluster. The dependency on the JGroups library is still there, although a JGroups channel is not started.

8.1.2. Replicated Caches

Replicated caches replicate all changes to some or all of the other cache instances in the cluster. Replication can either happen after each modification (no transactions or batches), or at the end of a transaction or batch.

Replication can be synchronous or asynchronous. Use of either one of the options is application dependent. Synchronous replication blocks the caller (e.g. on a **`put()`**) until the modifications have been replicated successfully to all nodes in a cluster. Asynchronous replication performs replication in the background (the **`put()`** returns immediately). JBoss Cache also offers a replication queue, where modifications are replicated periodically (i.e. interval-based), or when the queue size exceeds a number of elements, or a combination thereof. A replication queue can therefore offer much higher performance as the actual replication is performed by a background thread.

Asynchronous replication is faster (no caller blocking), because synchronous replication requires acknowledgments from all nodes in a cluster that they received and applied the modification successfully (round-trip time). However, when a synchronous replication returns successfully, the caller knows for sure that all modifications have been applied to all cache instances, whereas this is not be the case with asynchronous replication. With asynchronous replication, errors are simply written to a log. Even when using transactions, a transaction may succeed but replication may not succeed on all cache instances.

8.1.2.1. Replicated Caches and Transactions

When using transactions, replication only occurs at the transaction boundary - i.e., when a transaction commits. This results in minimizing replication traffic since a single modification is broadcast rather than a series of individual modifications, and can be a lot more efficient than not using transactions. Another effect of this is that if a transaction were to roll back, nothing is broadcast across a cluster.

Depending on whether you are running your cluster in asynchronous or synchronous mode, JBoss Cache will use either a single phase or two-phase commit protocol, respectively.

8.1.2.1.1. One Phase Commits

Used when your cache mode is REPL_ASYNC. All modifications are replicated in a single call, which instructs remote caches to apply the changes to their local in-memory state and commit locally. Remote errors/rollbacks are never fed back to the originator of the transaction since the communication is asynchronous.

8.1.2.1.2. Two Phase Commits

Used when your cache mode is REPL_SYNC. Upon committing your transaction, JBoss Cache broadcasts a prepare call, which carries all modifications relevant to the transaction. Remote caches then acquire local locks on their in-memory state and apply the modifications. Once all remote caches respond to the prepare call, the originator of the transaction broadcasts a commit. This instructs all remote caches to commit their data. If any of the caches fail to respond to the prepare phase, the originator broadcasts a rollback.

Note that although the prepare phase is synchronous, the commit and rollback phases are asynchronous. This is because Sun's JTA specification does not specify how transactional resources should deal with failures at this stage of a transaction; and other resources participating in the transaction may have indeterminate state anyway. As such, we do away with the overhead of synchronous communication for this phase of the transaction. That said, they can be forced to be synchronous using the **`SyncCommitPhase`** and **`SyncRollbackPhase`** configuration attributes.

8.1.2.2. Buddy Replication

Buddy Replication allows you to suppress replicating your data to all instances in a cluster. Instead, each instance picks one or more 'buddies' in the cluster, and only replicates to these specific buddies. This greatly helps scalability as there is no longer a memory and network traffic impact every time another instance is added to a cluster.

One of the most common use cases of Buddy Replication is when a replicated cache is used by a servlet container to store HTTP session data. One of the pre-requisites to buddy replication working well and being a real benefit is the use of *session affinity* , more casually known as *sticky sessions* in HTTP session replication speak. What this means is that if certain data is frequently accessed, it is desirable that this is always accessed on one instance rather than in a round-robin fashion as this helps the cache cluster optimize how it chooses buddies, where it stores data, and minimizes replication traffic.

If this is not possible, Buddy Replication may prove to be more of an overhead than a benefit.

8.1.2.2.1. Selecting Buddies

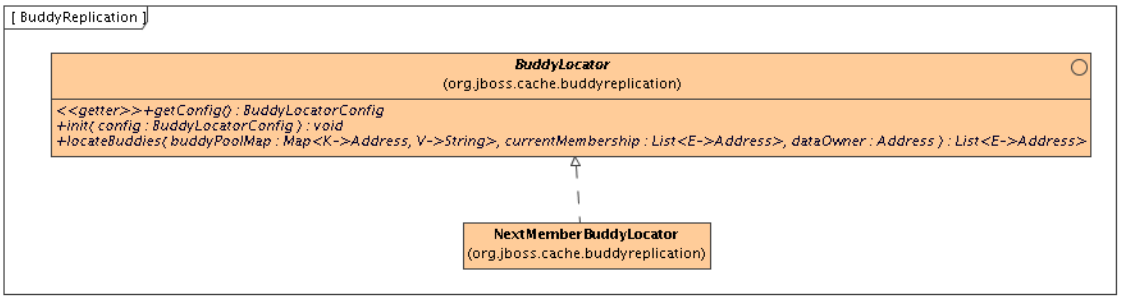


Figure 8.1. BuddyLocator

Buddy Replication uses an instance of a **`BuddyLocator`** which contains the logic used to select buddies in a network. JBoss Cache currently ships with a single implementation, **`NextMemberBuddyLocator`**, which is used as a default if no implementation is provided. The **`NextMemberBuddyLocator`** selects the next member in the cluster, as the name suggests, and guarantees an even spread of buddies for each instance.

The **`NextMemberBuddyLocator`** takes in 2 parameters, both optional.

- **`numBuddies`** - specifies how many buddies each instance should pick to back its data onto. This defaults to 1.

- **`ignoreColocatedBuddies`** - means that each instance will *try* to select a buddy on a different physical host. If not able to do so though, it will fall back to co-located instances. This defaults to **`true`** .

8.1.2.2.2. BuddyPools

Also known as *replication groups*, a buddy pool is an optional construct where each instance in a cluster may be configured with a buddy pool name. Think of this as an 'exclusive club membership' where when selecting buddies, **`BuddyLocator`** s that support buddy pools would try and select buddies sharing the same buddy pool name. This allows system administrators a degree of flexibility and control over how buddies are selected. For example, a sysadmin may put two instances on two separate physical servers that may be on two separate physical racks in the same buddy pool. So rather than picking an instance on a different host on the same rack, **`BuddyLocator`** s would rather pick the instance in the same buddy pool, on a separate rack which may add a degree of redundancy.

8.1.2.2.3. Failover

In the unfortunate event of an instance crashing, it is assumed that the client connecting to the cache (directly or indirectly, via some other service such as HTTP session replication) is able to redirect the request to any other random cache instance in the cluster. This is where a concept of Data Gravitation comes in.

Data Gravitation is a concept where if a request is made on a cache in the cluster and the cache does not contain this information, it asks other instances in the cluster for the data. In other words, data is lazily transferred, migrating *only* when other nodes ask for it. This strategy prevents a network storm effect where lots of data is pushed around healthy nodes because only one (or a few) of them die.

If the data is not found in the primary section of some node, it would (optionally) ask other instances to check in the backup data they store for other caches. This means that even if a cache containing your session dies, other instances will still be able to access this data by asking the cluster to search through their backups for this data.

Once located, this data is transferred to the instance which requested it and is added to this instance's data tree. The data is then (optionally) removed from all other instances (and backups) so that if session affinity is used, the affinity should now be to this new cache instance which has just *taken ownership* of this data.

Data Gravitation is implemented as an interceptor. The following (all optional) configuration properties pertain to data gravitation.

- **`dataGravitationRemoveOnFind`** - forces all remote caches that own the data or hold backups for the data to remove that data, thereby making the requesting cache the new data owner. This removal, of course, only happens after the new owner finishes replicating data to its buddy. If set to **`false`** an evict is broadcast instead of a remove, so any state persisted in cache loaders will remain. This is useful if you have a shared cache loader configured. Defaults to **`true`** .

- **`dataGravitationSearchBackupTrees`** - Asks remote instances to search through their backups as well as main data trees. Defaults to **`true`** . The resulting effect is that if this is **`true`** then backup nodes can respond to data gravitation requests in addition to data owners.

- **`autoDataGravitation`** - Whether data gravitation occurs for every cache miss. By default this is set to **`false`** to prevent unnecessary network calls. Most use cases will know when it may need to gravitate data and will pass in an **`Option`** to enable data gravitation on a per-invocation basis. If **`autoDataGravitation`** is **`true`** this **`Option`** is unnecessary.

8.1.2.2.4. Configuration

See the Chapter 12, *Configuration References* for details on configuring buddy replication.

8.2. INVALIDATION

If a cache is configured for invalidation rather than replication, every time data is changed in a cache other caches in the cluster receive a message informing them that their data is now stale and should be evicted from memory. Invalidation, when used with a shared cache loader (see chapter on Chapter 9, *Cache Loaders*) would cause remote caches to refer to the shared cache loader to retrieve modified data. The benefit of this is twofold: network traffic is minimized as invalidation messages are very small compared to replicating updated data, and also that other caches in the cluster look up modified data in a lazy manner, only when needed.

Invalidation messages are sent after each modification (no transactions or batches), or at the end of a transaction or batch, upon successful commit. This is usually more efficient as invalidation messages can be optimized for the transaction as a whole rather than on a per-modification basis.

Invalidation too can be synchronous or asynchronous, and just as in the case of replication, synchronous invalidation blocks until all caches in the cluster receive invalidation messages and have evicted stale data while asynchronous invalidation works in a 'fire-and-forget' mode, where invalidation messages are broadcast but doesn't block and wait for responses.

8.3. STATE TRANSFER

State Transfer refers to the process by which a JBoss Cache instance prepares itself to begin providing a service by acquiring the current state from another cache instance and integrating that state into its own state.

8.3.1. State Transfer Types

There are three divisions of state transfer types depending on a point of view related to state transfer. First, in the context of particular state transfer implementation, the underlying plumbing, there are two starkly different state transfer types: byte array and streaming based state transfer. Second, state transfer can be full or partial state transfer depending on a subtree being transferred. Entire cache tree transfer represents full transfer while transfer of a particular subtree represents partial state transfer. And finally state transfer can be "in-memory" and "persistent" transfer depending on a particular use of cache.

8.3.2. Byte array and streaming based state transfer

Byte array based transfer was a default and only transfer methodology for cache in all previous releases up to 2.0. Byte array based transfer loads entire state transferred into a byte array and sends it to a state receiving member. Major limitation of this approach is that the state transfer that is very large (>1GB) would likely result in OutOfMemoryException. Streaming state transfer provides an InputStream to a state reader and an OutputStream to a state writer. OutputStream and InputStream abstractions enable state transfer in byte chunks thus resulting in smaller memory requirements. For example, if application state is represented as a tree whose aggregate size is 1GB, rather than having to provide a 1GB byte array streaming state transfer transfers the state in chunks of N bytes where N is user configurable.

Byte array and streaming based state transfer are completely API transparent, interchangeable, and statically configured through a standard cache configuration XML file. Refer to JGroups documentation on how to change from one type of transfer to another.

8.3.3. Full and partial state transfer

If either in-memory or persistent state transfer is enabled, a full or partial state transfer will be done at various times, depending on how the cache is used. "Full" state transfer refers to the transfer of the state related to the entire tree -- i.e. the root node and all nodes below it. A "partial" state transfer is one where just a portion of the tree is transferred -- i.e. a node at a given Fqn and all nodes below it.

If either in-memory or persistent state transfer is enabled, state transfer will occur at the following times:

1. Initial state transfer. This occurs when the cache is first started (as part of the processing of the **`start()`** method). This is a full state transfer. The state is retrieved from the cache instance that has been operational the longest. [1] If there is any problem receiving or integrating the state, the cache will not start.

   Initial state transfer will occur unless:

   1. The cache's **`InactiveOnStartup`** property is **`true`**. This property is used in conjunction with region-based marshalling.

   2. Buddy replication is used. See below for more on state transfer with buddy replication.

2. Partial state transfer following region activation. When region-based marshalling is used, the application needs to register a specific class loader with the cache. This class loader is used to unmarshall the state for a specific region (subtree) of the cache.

   After registration, the application calls **`cache.getRegion(fqn, true).activate()`**, which initiates a partial state transfer of the relevant subtree's state. The request is first made to the oldest cache instance in the cluster. However, if that instance responds with no state, it is then requested from each instance in turn until one either provides state or all instances have been queried.

   Typically when region-based marshalling is used, the cache's **`InactiveOnStartup`** property is set to **`true`**. This suppresses initial state transfer, which would fail due to the inability to deserialize the transferred state.

3. Buddy replication. When buddy replication is used, initial state transfer is disabled. Instead, when a cache instance joins the cluster, it becomes the buddy of one or more other instances, and one or more other instances become its buddy. Each time an instance determines it has a new buddy providing backup for it, it pushes its current state to the new buddy. This "pushing" of state to the new buddy is slightly different from other forms of state transfer, which are based on a "pull" approach (i.e. recipient asks for and receives state). However, the process of preparing and integrating the state is the same.

   This "push" of state upon buddy group formation only occurs if the **`InactiveOnStartup`** property is set to **`false`**. If it is **`true`**, state transfer amongst the buddies only occurs when the application activates the region on the various members of the group.

   Partial state transfer following a region activation call is slightly different in the buddy replication case as well. Instead of requesting the partial state from one cache instance, and trying all instances until one responds, with buddy replication the instance that is activating a region will request partial state from each instance for which it is serving as a backup.

8.3.4. Transient ("in-memory") and persistent state transfer

The state that is acquired and integrated can consist of two basic types:

1. "Transient" or "in-memory" state. This consists of the actual in-memory state of another cache instance - the contents of the various in-memory nodes in the cache that is providing state are

serialized and transferred; the recipient deserializes the data, creates corresponding nodes in its own in-memory tree, and populates them with the transferred data.

"In-memory" state transfer is enabled by setting the cache's **`FetchInMemoryState`** configuration attribute to **`true`**.

2. "Persistent" state. Only applicable if a non-shared cache loader is used. The state stored in the state-provider cache's persistent store is deserialized and transferred; the recipient passes the data to its own cache loader, which persists it to the recipient's persistent store.

   "Persistent" state transfer is enabled by setting a cache loader's **`fetchPersistentState`** attribute to **`true`**. If multiple cache loaders are configured in a chain, only one can have this property set to true; otherwise you will get an exception at startup.

   Persistent state transfer with a shared cache loader does not make sense, as the same persistent store that provides the data will just end up receiving it. Therefore, if a shared cache loader is used, the cache will not allow a persistent state transfer even if a cache loader has **`fetchPersistentState`** set to **`true`**.

Which of these types of state transfer is appropriate depends on the usage of the cache.

1. If a write-through cache loader is used, the current cache state is fully represented by the persistent state. Data may have been evicted from the in-memory state, but it will still be in the persistent store. In this case, if the cache loader is not shared, persistent state transfer is used to ensure the new cache has the correct state. In-memory state can be transferred as well if the desire is to have a "hot" cache -- one that has all relevant data in memory when the cache begins providing service. (Note that the **`<preload>`** element in the **`<loaders>`** configuration element can be used as well to provide a "warm" or "hot" cache without requiring an in-memory state transfer. This approach somewhat reduces the burden on the cache instance providing state, but increases the load on the persistent store on the recipient side.)

2. If a cache loader is used with passivation, the full representation of the state can only be obtained by combining the in-memory (i.e. non-passivated) and persistent (i.e. passivated) states. Therefore an in-memory state transfer is necessary. A persistent state transfer is necessary if the cache loader is not shared.

3. If no cache loader is used and the cache is solely a write-aside cache (i.e. one that is used to cache data that can also be found in a persistent store, e.g. a database), whether or not in-memory state should be transferred depends on whether or not a "hot" cache is desired.

8.3.5. Configuring State Transfer

To ensure state transfer behaves as expected, it is important that all nodes in the cluster are configured with the same settings for persistent and transient state. This is because byte array based transfers, when requested, rely only on the requester's configuration while stream based transfers rely on both the requester and sender's configuration, and this is expected to be identical.

---

[1] The longest operating cache instance is always, in JGroups terms, the coordinator.

CHAPTER 9. CACHE LOADERS

JBoss Cache can use a **`CacheLoader`** to back up the in-memory cache to a backend datastore. If JBoss Cache is configured with a cache loader, then the following features are provided:

- Whenever a cache element is accessed, and that element is not in the cache (e.g. due to eviction or due to server restart), then the cache loader transparently loads the element into the cache if found in the backend store.
- Whenever an element is modified, added or removed, then that modification is persisted in the backend store via the cache loader. If transactions are used, all modifications created within a transaction are persisted. To this end, the **`CacheLoader`** takes part in the two phase commit protocol run by the transaction manager, although it does not do so explicitly.

9.1. THE CACHELOADER INTERFACE AND LIFECYCLE

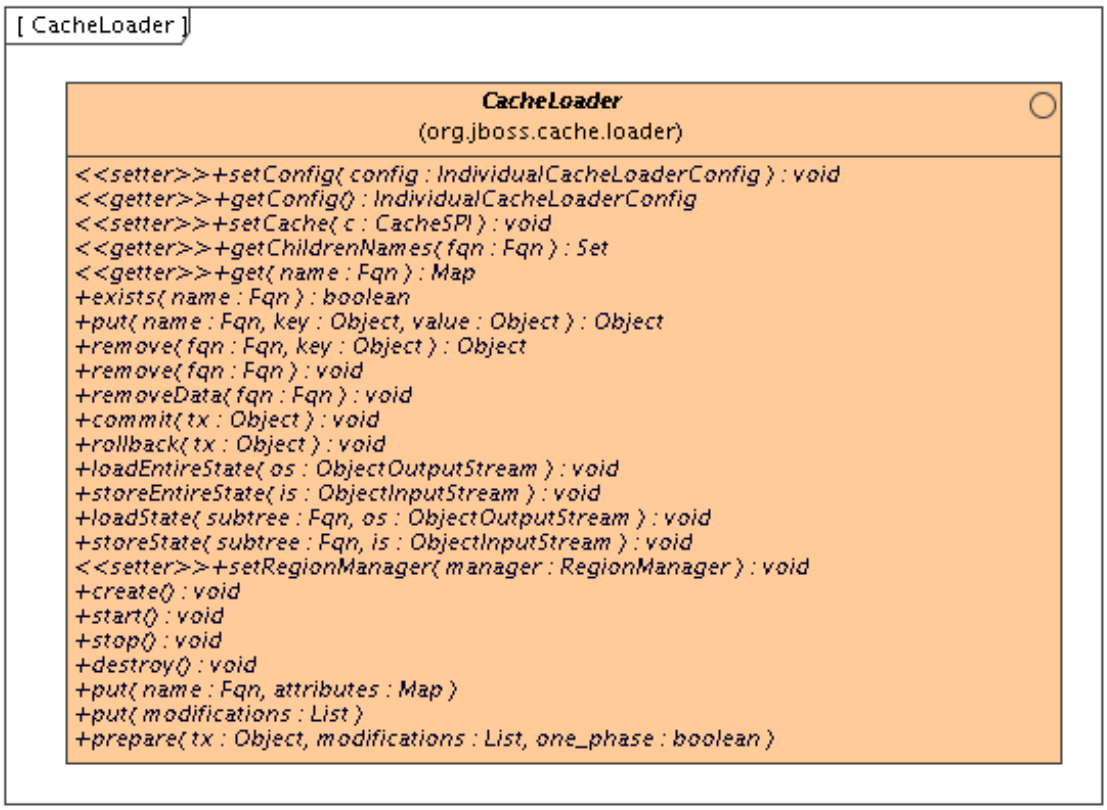


Figure 9.1. The CacheLoader interface

The interaction between JBoss Cache and a **`CacheLoader`** implementation is as follows. When **`CacheLoaderConfiguration`** (see below) is non-null, an instance of each configured **`CacheLoader`** is created when the cache is created, and started when the cache is started.

`CacheLoader.create()` and **`CacheLoader.start()`** are called when the cache is started. Correspondingly, **`stop()`** and **`destroy()`** are called when the cache is stopped.

Next, **`setConfig()`** and **`setCache()`** are called. The latter can be used to store a reference to the cache, the former is used to configure this instance of the **`CacheLoader`** . For example, here a database cache loader could establish a connection to the database.

The **CacheLoader** interface has a set of methods that are called when no transactions are used: **get()** , **put()** , **remove()** and **removeData()** : they get/set/remove the value immediately. These methods are described as javadoc comments in the interface.

Then there are three methods that are used with transactions: **prepare()** , **commit()** and **rollback()** . The **prepare()** method is called when a transaction is to be committed. It has a transaction object and a list of modfications as argument. The transaction object can be used as a key into a hashmap of transactions, where the values are the lists of modifications. Each modification list has a number of **Modification** elements, which represent the changes made to a cache for a given transaction. When **prepare()** returns successfully, then the cache loader *must* be able to commit (or rollback) the transaction successfully.

JBoss Cache takes care of calling prepare(), commit() and rollback() on the cache loaders at the right time.

The **commit()** method tells the cache loader to commit the transaction, and the **rollback()** method tells the cache loader to discard the changes associated with that transaction.

See the javadocs on this interface for a detailed explanation on each method and the contract implementations would need to fulfill.

9.2. CONFIGURATION

Cache loaders are configured as follows in the JBoss Cache XML file. Note that you can define several cache loaders, in a chain. The impact is that the cache will look at all of the cache loaders in the order they've been configured, until it finds a valid, non-null element of data. When performing writes, all cache loaders are written to, except if the **ignoreModifications** element has been set to **true** for a specific cache loader. See the configuration section below for details.

```
...

<!-- Cache loader config block -->
<!-- if passivation is true, only the first cache loader is used; the rest
are ignored -->
<loaders passivation="false" shared="false">
      <preload>
         <!-- Fqns to preload -->
         <node fqn="/some/stuff"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used;
the rest are ignored -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="true"
               ignoreModifications="false" purgeOnStartup="false">
         <properties>
            cache.jdbc.driver=com.mysql.jdbc.Driver
            cache.jdbc.url=jdbc:mysql://localhost:3306/jbossdb
            cache.jdbc.user=root
            cache.jdbc.password=
         </properties>
      </loader>
  </loaders>
```

The **class** element defines the class of the cache loader implementation. (Note that, because of a bug in the properties editor in JBoss AS, backslashes in variables for Windows filenames might not get

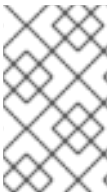
expanded correctly, so replace="false" may be necessary). Note that an implementation of cache loader has to have an empty constructor.

The **properties** element defines a configuration specific to the given implementation. The filesystem-based implementation for example defines the root directory to be used, whereas a database implementation might define the database URL, name and password to establish a database connection. This configuration is passed to the cache loader implementation via **CacheLoader.setConfig(Properties)**. Note that backspaces may have to be escaped.

preload allows us to define a list of nodes, or even entire subtrees, that are visited by the cache on startup, in order to preload the data associated with those nodes. The default ("/") loads the entire data available in the backend store into the cache, which is probably not a good idea given that the data in the backend store might be large. As an example, **/a, /product/catalogue** loads the subtrees **/a** and **/product/catalogue** into the cache, but nothing else. Anything else is loaded lazily when accessed. Preloading makes sense when one anticipates using elements under a given subtree frequently. .

fetchPersistentState determines whether or not to fetch the persistent state of a cache when joining a cluster. Only one configured cache loader may set this property to true; if more than one cache loader does so, a configuration exception will be thrown when starting your cache service.

async determines whether writes to the cache loader block until completed, or are run on a separate thread so writes return immediately. If this is set to true, an instance of **org.jboss.cache.loader.AsyncCacheLoader** is constructed with an instance of the actual cache loader to be used. The **AsyncCacheLoader** then delegates all requests to the underlying cache loader, using a separate thread if necessary. See the Javadocs on **AsyncCacheLoader** for more details. If unspecified, the **async** element defaults to **false**.



NOTE

There is always the possibility of dirty reads since all writes are performed asynchronously, and it is thus impossible to guarantee when (and even if) a write succeeds. This needs to be kept in mind when setting the **async** element to true.

ignoreModifications determines whether write methods are pushed down to the specific cache loader. Situations may arise where transient application data should only reside in a file based cache loader on the same server as the in-memory cache, for example, with a further shared **JDBCCacheLoader** used by all servers in the network. This feature allows you to write to the 'local' file cache loader but not the shared **JDBCCacheLoader**. This property defaults to **false**, so writes are propagated to all cache loaders configured.

purgeOnStatup empties the specified cache loader (if **ignoreModifications** is **false**) when the cache loader starts up.

shared indicates that the cache loader is shared among different cache instances, for example where all instances in a cluster use the same JDBC settings t talk to the same remote, shared database. Setting this to **true** prevents repeated and unnecessary writes of the same data to the cache loader by different cache instances. Default value is **false** .

9.2.1. Singleton Store Configuration

```
<loaders passivation="false" shared="true">
   <preload>
      <node fqn="/a/b/c"/>
      <node fqn="/f/r/s"/>
```

```
    </preload>

    <!-- we can now have multiple cache loaders, which get chained -->
    <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="false"
         ignoreModifications="false" purgeOnStartup="false">
       <properties>
         cache.jdbc.datasource=java:/DefaultDS
       </properties>
       <singletonStore enabled="true"
class="org.jboss.cache.loader.SingletonStoreCacheLoader">
          <properties>
             pushStateWhenCoordinator=true
             pushStateWhenCoordinatorTimeout=20000
          </properties>
       </singletonStore>
    </loader>
 </loaders>
```

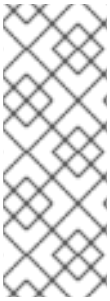
singletonStore element enables modifications to be stored by only one node in the cluster, the coordinator. Essentially, whenever any data comes in to some node it is always replicated so as to keep the caches' in-memory states in sync; the coordinator, though, has the sole responsibility of pushing that state to disk. This functionality can be activated setting the **enabled** subelement to true in all nodes, but again only the coordinator of the cluster will store the modifications in the underlying cache loader as defined in **loader** element. You cannot define a cache loader as **shared** and with **singletonStore** enabled at the same time. Default value for **enabled** is **false**.

Optionally, within the **singletonStore** element, you can define a **class** element that specifies the implementation class that provides the singleton store functionality. This class must extend **org.jboss.cache.loader.AbstractDelegatingCacheLoader**, and if absent, it defaults to **org.jboss.cache.loader.SingletonStoreCacheLoader**.

The **properties** subelement defines properties that allow changing the behavior of the class providing the singleton store functionality. By default, **pushStateWhenCoordinator** and **pushStateWhenCoordinatorTimeout** properties have been defined, but more could be added as required by the user-defined class providing singleton store functionality.

pushStateWhenCoordinator allows the in-memory state to be pushed to the cache store when a node becomes the coordinator, as a result of the new election of coordinator due to a cluster topology change. This can be very useful in situations where the coordinator crashes and there's a gap in time until the new coordinator is elected. During this time, if this property was set to **false** and the cache was updated, these changes would never be persisted. Setting this property to **true** would ensure that any changes during this process also get stored in the cache loader. You would also want to set this property to **true** if each node's cache loader is configured with a different location. Default value is **true**.

pushStateWhenCoordinatorTimeout is only relevant if **pushStateWhenCoordinator** is **true** in which case, sets the maximum number of milliseconds that the process of pushing the in-memory state to the underlying cache loader should take, reporting a **PushStateException** if exceeded. Default value is 20000.



NOTE

Setting up a cache loader as a singleton and using cache passivation (via evictions) can lead to undesired effects. If a node is to be passivated as a result of an eviction, while the cluster is in the process of electing a new coordinator, the data will be lost. This is because no coordinator is active at that time and therefore, none of the nodes in the cluster will store the passivated node. A new coordinator is elected in the cluster when either, the coordinator leaves the cluster, the coordinator crashes or stops responding.

9.3. SHIPPED IMPLEMENTATIONS

The currently available implementations shipped with JBoss Cache are as follows.

9.3.1. File system based cache loaders

JBoss Cache ships with several cache loaders that utilize the file system as a data store. They all require that the **<loader><properties>** configuration element contains a **location** property, which maps to a directory to be used as a persistent store (e.g., **location=/tmp/myDataStore**). Used mainly for testing and not recommended for production use.

- **FileCacheLoader**, which is a simple filesystem-based implementation. By default, this cache loader checks for any potential character portability issues in the location or tree node names, for example invalid characters, producing warning messages. These checks can be disabled adding **check.character.portability** property to the **<properties>** element and setting it to **false** (e.g., **check.character.portability=false**).

  The FileCacheLoader has some severe limitations which restrict its use in a production environment, or if used in such an environment, it should be used with due care and sufficient understanding of these limitations.

  - Due to the way the FileCacheLoader represents a tree structure on disk (directories and files) traversal is inefficient for deep trees.
  - Usage on shared filesystems like NFS, Windows shares, etc. should be avoided as these do not implement proper file locking and can cause data corruption.
  - Usage with an isolation level of NONE can cause corrupt writes as multiple threads attempt to write to the same file.
  - File systems are inherently not transactional, so when attempting to use your cache in a transactional context, failures when writing to the file (which happens during the commit phase) cannot be recovered.

  As a rule of thumb, it is recommended that the FileCacheLoader not be used in a highly concurrent, transactional or stressful environment, and its use is restricted to testing.

- **BdbjeCacheLoader** , which is a cache loader implementation based on the Oracle/Sleepycat's BerkeleyDB Java Edition.

- **JdbmCacheLoader**, which is a cache loader implementation based on the JDBM engine, a fast and free alternative to BerkeleyDB.

Note that the BerkeleyDB implementation is much more efficient than the filesystem-based implementation, and provides transactional guarantees, but requires a commercial license if distributed with an application (see http://www.oracle.com/database/berkeley-db/index.html for details).

9.3.2. Cache loaders that delegate to other caches

- **`LocalDelegatingCacheLoader`** , which enables loading from and storing to another local (same JVM) cache.
- **`ClusteredCacheLoader`**, which allows querying of other caches in the same cluster for in-memory data via the same clustering protocols used to replicate data. Writes are *not* 'stored' though, as replication would take care of any updates needed. You need to specify a property called **`timeout`**, a long value telling the cache loader how many milliseconds to wait for responses from the cluster before assuming a null value. For example, **`timeout = 3000`** would use a timeout value of 3 seconds.

9.3.3. JDBCCacheLoader

JBossCache is distributed with a JDBC-based cache loader implementation that stores/loads nodes' state into a relational database. The implementing class is **`org.jboss.cache.loader.JDBCCacheLoader`**.

The current implementation uses just one table. Each row in the table represents one node and contains three columns:

- column for **`Fqn`** (which is also a primary key column)
- column for node contents (attribute/value pairs)
- column for parent **`Fqn`**

`Fqn`s are stored as strings. Node content is stored as a BLOB.

WARNING

JBoss Cache does not impose any limitations on the types of objects used in **`Fqn`** but this implementation of cache loader requires **`Fqn`** to contain only objects of type **`java.lang.String`**. Another limitation for **`Fqn`** is its length. Since **`Fqn`** is a primary key, its default column type is **`VARCHAR`** which can store text values up to some maximum length determined by the database in use.

See this wiki page for configuration tips with specific database systems.

9.3.3.1. JDBCCacheLoader configuration

9.3.3.1.1. Table configuration

Table and column names as well as column types are configurable with the following properties.

- *cache.jdbc.table.name* - the name of the table. Can be prepended with schema name for the given table: **`{schema_name}.{table_name}`**. The default value is 'jbosscache'.
- *cache.jdbc.table.primarykey* - the name of the primary key for the table. The default value is 'jbosscache_pk'.

- *cache.jdbc.table.create* - can be true or false. Indicates whether to create the table during startup. If true, the table is created if it doesn't already exist. The default value is true.
- *cache.jdbc.table.drop* - can be true or false. Indicates whether to drop the table during shutdown. The default value is true.
- *cache.jdbc.fqn.column* - FQN column name. The default value is 'fqn'.
- *cache.jdbc.fqn.type* - FQN column type. The default value is 'varchar(255)'.
- *cache.jdbc.node.column* - node contents column name. The default value is 'node'.
- *cache.jdbc.node.type* - node contents column type. The default value is 'blob'. This type must specify a valid binary data type for the database being used.

9.3.3.1.2. DataSource

If you are using JBossCache in a managed environment (e.g., an application server) you can specify the JNDI name of the DataSource you want to use.

- *cache.jdbc.datasource* - JNDI name of the DataSource. The default value is **`java:/DefaultDS`** .

9.3.3.1.3. JDBC driver

If you are *not* using DataSource you have the following properties to configure database access using a JDBC driver.

- *cache.jdbc.driver* - fully qualified JDBC driver name.
- *cache.jdbc.url* - URL to connect to the database.
- *cache.jdbc.user* - user name to connect to the database.
- *cache.jdbc.password* - password to connect to the database.

9.3.3.1.4. c3p0 connection pooling

JBoss Cache implements JDBC connection pooling when running outside of an application server standalone using the c3p0:JDBC DataSources/Resource Pools library. In order to enable it, just edit the following property:

- *cache.jdbc.connection.factory* - Connection factory class name. If not set, it defaults to standard non-pooled implementation. To enable c3p0 pooling, just set the connection factory class for c3p0. See example below.

You can also set any c3p0 parameters in the same cache loader properties section but don't forget to start the property name with 'c3p0.'. To find a list of available properties, please check the c3p0 documentation for the c3p0 library version distributed in c3p0:JDBC DataSources/Resource Pools . Also, in order to provide quick and easy way to try out different pooling parameters, any of these properties can be set via a System property overriding any values these properties might have in the JBoss Cache XML configuration file, for example: **`-Dc3p0.maxPoolSize=20`** . If a c3p0 property is not defined in either the configuration file or as a System property, default value, as indicated in the c3p0 documentation, will apply.

9.3.3.1.5. Configuration example

Below is an example of a JDBCCacheLoader using Oracle as database. The CacheLoaderConfiguration XML element contains an arbitrary set of properties which define the database-related configuration.

```
<loaders passivation="false" shared="false">
      <preload>
         <node fqn="/some/stuff"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used;
the rest are ignored -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="true"
              ignoreModifications="false" purgeOnStartup="false">
         <properties>
            cache.jdbc.table.name=jbosscache
            cache.jdbc.table.create=true
            cache.jdbc.table.drop=true
            cache.jdbc.table.primarykey=jbosscache_pk
            cache.jdbc.fqn.column=fqn
            cache.jdbc.fqn.type=VARCHAR(255)
            cache.jdbc.node.column=node
            cache.jdbc.node.type=BLOB
            cache.jdbc.parent.column=parent
            cache.jdbc.driver=oracle.jdbc.OracleDriver
            cache.jdbc.url=jdbc:oracle:thin:@localhost:1521:JBOSSDB
            cache.jdbc.user=SCOTT
            cache.jdbc.password=TIGER
          </properties>
      </loader>
  </loaders>
```

As an alternative to configuring the entire JDBC connection, the name of an existing data source can be given:

```
 <loaders passivation="false" shared="false">
      <preload>
         <node fqn="/some/stuff"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used;
the rest are ignored -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="true"
              ignoreModifications="false" purgeOnStartup="false">
         <properties>
            cache.jdbc.datasource=java:/DefaultDS
         </properties>
      </loader>
  </loaders>
```

Configuration example for a cache loader using c3p0 JDBC connection pooling:

```
 <loaders passivation="false" shared="false">
      <preload>
         <node fqn="/some/stuff"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used;
```

```
the rest are ignored -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="false"
fetchPersistentState="true"
                ignoreModifications="false" purgeOnStartup="false">
         <properties>
            cache.jdbc.table.name=jbosscache
            cache.jdbc.table.create=true
            cache.jdbc.table.drop=true
            cache.jdbc.table.primarykey=jbosscache_pk
            cache.jdbc.fqn.column=fqn
            cache.jdbc.fqn.type=VARCHAR(255)
            cache.jdbc.node.column=node
            cache.jdbc.node.type=BLOB
            cache.jdbc.parent.column=parent
            cache.jdbc.driver=oracle.jdbc.OracleDriver
            cache.jdbc.url=jdbc:oracle:thin:@localhost:1521:JBOSSDB
            cache.jdbc.user=SCOTT
            cache.jdbc.password=TIGER

cache.jdbc.connection.factory=org.jboss.cache.loader.C3p0ConnectionFactory
            c3p0.maxPoolSize=20
            c3p0.checkoutTimeout=5000
         </properties>
      </loader>
   </loaders>
```

9.3.4. S3CacheLoader

The **`S3CacheLoader`** uses the Amazon S3 (Simple Storage Solution) for storing cache data. Since Amazon S3 is remote network storage and has fairly high latency, it is really best for caches that store large pieces of data, such as media or files. But consider this cache loader over the JDBC or file system based cache loaders if you want remotely managed, highly reliable storage. Or, use it for applications running on Amazon's EC2 (Elastic Compute Cloud).

If you're planning to use Amazon S3 for storage, consider using it with JBoss Cache. JBoss Cache itself provides in-memory caching for your data to minimize the amount of remote access calls, thus reducing the latency and cost of fetching your Amazon S3 data. With cache replication, you are also able to load data from your local cluster without having to remotely access it every time.

Note that Amazon S3 does not support transactions. If transactions are used in your application then there is some possibility of state inconsistency when using this cache loader. However, writes are atomic, in that if a write fails nothing is considered written and data is never corrupted.

Data is stored in keys based on the Fqn of the Node and Node data is serialized as a java.util.Map using the **`CacheSPI.getMarshaller()`** instance. Read the javadoc on how data is structured and stored. Data is stored using Java serialization. Be aware this means data is not readily accessible over HTTP to non-JBoss Cache clients. Your feedback and help would be appreciated to extend this cache loader for that purpose.

With this cache loader, single-key operations such as **`Node.remove(Object)`** and **`Node.put(Object, Object)`** are the slowest as data is stored in a single Map instance. Use bulk operations such as **`Node.replaceAll(Map)`** and **`Node.clearData()`** for more efficiency. Try the **`cache.s3.optimize`** option as well.

9.3.4.1. Amazon S3 Library

The S3 cache loader is provided with the default distribution but requires a library to access the service at runtime. This runtime library may be obtained through a Sourceforge Maven Repository. Include the following sections in your pom.xml file:

```
<repository>
   <id>e-xml.sourceforge.net</id>
   <url>http://e-xml.sourceforge.net/maven2/repository</url>
</repository>
  ...
<dependency>
   <groupId>net.noderunner</groupId>
   <artifactId>amazon-s3</artifactId>
   <version>1.0.0.0</version>
   <scope>runtime</scope>
</dependency>
```

If you do not use Maven, you can still download the amazon-s3 library by navigating the repository or through this URL.

9.3.4.2. Configuration

At a minimum, you must configure your Amazon S3 access key and secret access key. The following configuration keys are listed in general order of utility.

- **`cache.s3.accessKeyId`** - Amazon S3 Access Key, available from your account profile.
- **`cache.s3.secretAccessKey`** - Amazon S3 Secret Access Key, available from your account profile. As this is a password, be careful not to distribute it or include this secret key in built software.
- **`cache.s3.secure`** - The default is **`false`**: Traffic is sent unencrypted over the public Internet. Set to **`true`** to use HTTPS. Note that unencrypted uploads and downloads use less CPU.
- **`cache.s3.bucket`** - Name of the bucket to store data. For different caches using the same access key, use a different bucket name. Read the S3 documentation on the definition of a bucket. The default value is **`jboss-cache`**.
- **`cache.s3.callingFormat`** - One of**`PATH`**, **`SUBDOMAIN`**, or **`VANITY`**. Read the S3 documentation on the use of calling domains. The default value is **`SUBDOMAIN`**.
- **`cache.s3.optimize`** - The default is **`false`**. If true, **`put(Map)`** operations replace the data stored at an Fqn rather than attempt to fetch and merge. (This option is fairly experimental at the moment.)
- **`cache.s3.parentCache`** - The default is **`true`**. Set this value to **`false`** if you are using multiple caches sharing the same S3 bucket, that remove parent nodes of nodes being created in other caches. (This is not a common use case.)

  JBoss Cache stores nodes in a tree format and automatically creates intermediate parent nodes as necessary. The S3 cache loader must also create these parent nodes as well to allow for operations such as **`getChildrenNames`** to work properly. Checking if all parent nodes exists for every **`put`** operation is fairly expensive, so by default the cache loader caches the existence of these parent nodes.

- **`cache.s3.location`** - This choses a primary storage location for your data to reduce loading and retrieval latency. Set to **`EU`** to store data in Europe. The default is **`null`**, to store data in the United States.

9.3.5. TcpDelegatingCacheLoader

This cache loader allows to delegate loads and stores to another instance of JBoss Cache, which could reside (a) in the same address space, (b) in a different process on the same host, or (c) in a different process on a different host.

A TcpDelegatingCacheLoader talks to a remote **`org.jboss.cache.loader.tcp.TcpCacheServer`**, which can be a standalone process started on the command line, or embedded as an MBean inside JBoss AS. The **`TcpCacheServer`** has a reference to another JBoss Cache instance, which it can create itself, or which is given to it (e.g. by JBoss, using dependency injection).

As of JBoss Cache 2.1.0, the **`TcpDelegatingCacheLoader`** transparently handles reconnects if the connection to the TcpCacheServer is lost.

The TcpDelegatingCacheLoader is configured with the host and port of the remote TcpCacheServer, and uses this to communicate to it. In addition, 2 new optional parameters are used to control transparent reconnecting to the TcpCacheServer. The **`timeout`** property (defaults to 5000) specifies the length of time the cache loader must continue retrying to connect to the TcpCacheServer before giving up and throwing an exception. The **`reconnectWaitTime`** (defaults to 500) is how long the cache loader should wait before attempting a reconnect if it detects a communication failure. The last two parameters can be used to add a level of fault tolerance to the cache loader, do deal with TcpCacheServer restarts.

The configuration looks as follows:

```
 <loaders passivation="false" shared="false">
      <preload>
         <node fqn="/"/>
      </preload>
      <!-- if passivation is true, only the first cache loader is used;
the rest are ignored -->
      <loader class="org.jboss.cache.loader.TcpDelegatingCacheLoader">
         <properties>
            host=myRemoteServer
            port=7500
            timeout=10000
            reconnectWaitTime=250
         </properties>
      </loader>
   </loaders>
```

This means this instance of JBoss Cache will delegate all load and store requests to the remote TcpCacheServer running on **`myRemoteServer:7500`**.

A typical use case could be multiple replicated instances of JBoss Cache in the same cluster, all delegating to the same TcpCacheServer instance. The TcpCacheServer might itself delegate to a database via JDBCCacheLoader, but the point here is that - if we have 5 nodes all accessing the same dataset - they will load the data from the TcpCacheServer, which has do execute one SQL statement per unloaded data set. If the nodes went directly to the database, then we'd have the same SQL executed multiple times. So TcpCacheServer serves as a natural cache in front of the DB (assuming that a network round trip is faster than a DB access (which usually also include a network round trip)).

To alleviate single point of failure, we could configure several cache loaders. The first cache loader is a ClusteredCacheLoader, the second a TcpDelegatingCacheLoader, and the last a JDBCacheLoader, effectively defining our cost of access to a cache in increasing order.

9.3.6. Transforming Cache Loaders

The way cached data is written to **`FileCacheLoader`** and **`JDBCCacheLoader`** based cache stores has changed in JBoss Cache 2.0 in such way that these cache loaders now write and read data using the same marhalling framework used to replicate data across the network. Such change is trivial for replication purposes as it just requires the rest of the nodes to understand this format. However, changing the format of the data in cache stores brings up a new problem: how do users, which have their data stored in JBoss Cache 1.x.x format, migrate their stores to JBoss Cache 2.0 format?

With this in mind, JBoss Cache 2.0 comes with two cache loader implementations called **`org.jboss.cache.loader.TransformingFileCacheLoader`** and **`org.jboss.cache.loader.TransformingJDBCCacheLoader`** located within the optional jbosscache-cacheloader-migration.jar file. These are one-off cache loaders that read data from the cache store in JBoss Cache 1.x.x format and write data to cache stores in JBoss Cache 2.0 format.

The idea is for users to modify their existing cache configuration file(s) momentarily to use these cache loaders and for them to create a small Java application that creates an instance of this cache, recursively reads the entire cache and writes the data read back into the cache. Once the data is transformed, users can revert back to their original cache configuration file(s). In order to help the users with this task, a cache loader migration example has been constructed which can be located under the **`examples/cacheloader-migration`** directory within the JBoss Cache distribution. This example, called **`examples.TransformStore`** , is independent of the actual data stored in the cache as it writes back whatever it was read recursively. It is highly recommended that anyone interested in porting their data run this example first, which contains a **`readme.txt`** file with detailed information about the example itself, and also use it as base for their own application.

9.4. CACHE PASSIVATION

A cache loader can be used to enforce node passivation and activation on eviction in a cache.

Cache Passivation is the process of removing an object from in-memory cache and writing it to a secondary data store (e.g., file system, database) on eviction. *Cache Activation* is the process of restoring an object from the data store into the in-memory cache when it's needed to be used. In both cases, the configured cache loader will be used to read from the data store and write to the data store.

When an eviction policy in effect evicts a node from the cache, if passivation is enabled, a notification that the node is being passivated will be emitted to the cache listeners and the node and its children will be stored in the cache loader store. When a user attempts to retrieve a node that was evicted earlier, the node is loaded (lazy loaded) from the cache loader store into memory. When the node and its children have been loaded, they're removed from the cache loader and a notification is emitted to the cache listeners that the node has been activated.

To enable cache passivation/activation, you can set **`passivation`** to true. The default is **`false`** . When passivation is used, only the first cache loader configured is used and all others are ignored.

9.4.1. Cache Loader Behavior with Passivation Disabled vs. Enabled

When passivation is disabled, whenever an element is modified, added or removed, then that modification is persisted in the backend store via the cache loader. There is no direct relationship between eviction and cache loading. If you don't use eviction, what's in the persistent store is basically a

copy of what's in memory. If you do use eviction, what's in the persistent store is basically a superset of what's in memory (i.e. it includes nodes that have been evicted from memory).

When passivation is enabled, there is a direct relationship between eviction and the cache loader. Writes to the persistent store via the cache loader only occur as part of the eviction process. Data is deleted from the persistent store when the application reads it back into memory. In this case, what's in memory and what's in the persistent store are two subsets of the total information set, with no intersection between the subsets.

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

1. Insert /A
2. Insert /B
3. Eviction thread runs, evicts /A
4. Read /A
5. Eviction thread runs, evicts /B
6. Remove /B

When passivation is disabled:

```
1) Memory: /A Disk: /A
2) Memory: /A, /B Disk: /A, /B
3) Memory: /B Disk: /A, /B
4) Memory: /A, /B Disk: /A, /B
5) Memory: /A Disk: /A, /B
6) Memory: /A Disk: /A
```

When passivation is enabled:

```
1) Memory: /A Disk:
2) Memory: /A, /B Disk:
3) Memory: /B Disk: /A
4) Memory: /A, /B Disk:
5) Memory: /A Disk: /B
6) Memory: /A Disk:
```

9.5. STRATEGIES

This section discusses different patterns of combining different cache loader types and configuration options to achieve specific outcomes.

9.5.1. Local Cache With Store

This is the simplest case. We have a JBoss Cache instance, whose cache mode is **`LOCAL`** , therefore no replication is going on. The cache loader simply loads non-existing elements from the store and stores modifications back to the store. When the cache is started, depending on the **`preload`** element, certain data can be preloaded, so that the cache is partly warmed up.

9.5.2. Replicated Caches With All Caches Sharing The Same Store

The following figure shows 2 JBoss Cache instances sharing the same backend store:

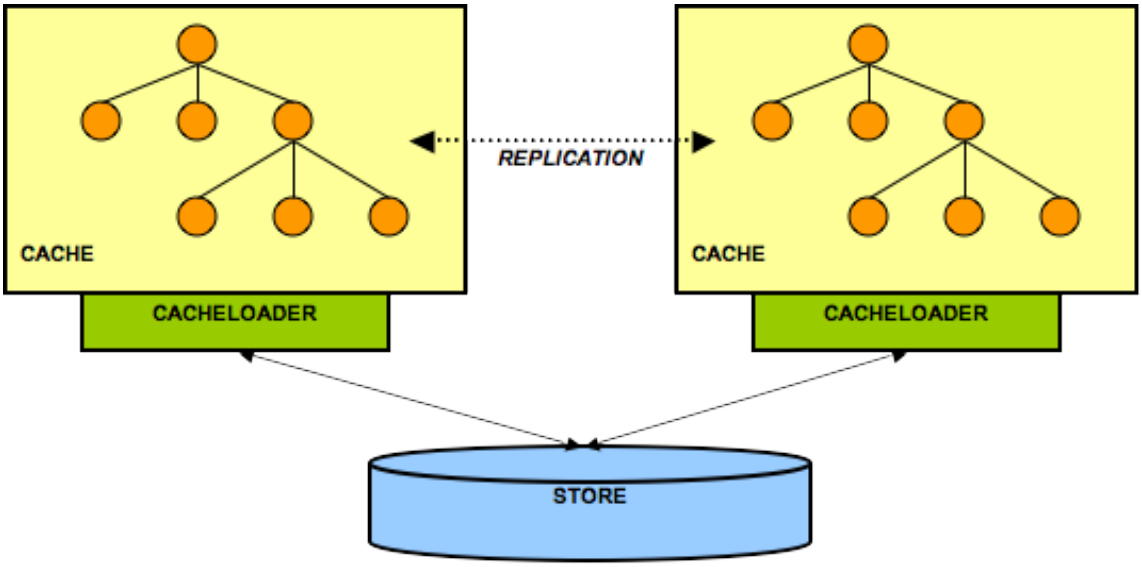


Figure 9.2. 2 nodes sharing a backend store

Both nodes have a cache loader that accesses a common shared backend store. This could for example be a shared filesystem (using the FileCacheLoader), or a shared database. Because both nodes access the same store, they don't necessarily need state transfer on startup.

NOTE

Of course they can enable state transfer, if they want to have a warm or hot cache after startup.

Rather, the `FetchInMemoryState` attribute could be set to false, resulting in a 'cold' cache, that gradually warms up as elements are accessed and loaded for the first time. This would mean that individual caches in a cluster might have different in-memory state at any given time (largely depending on their preloading and eviction strategies).

When storing a value, the writer takes care of storing the change in the backend store. For example, if node1 made change C1 and node2 C2, then node1 would tell its cache loader to store C1, and node2 would tell its cache loader to store C2.

9.5.3. Replicated Caches With Only One Cache Having A Store

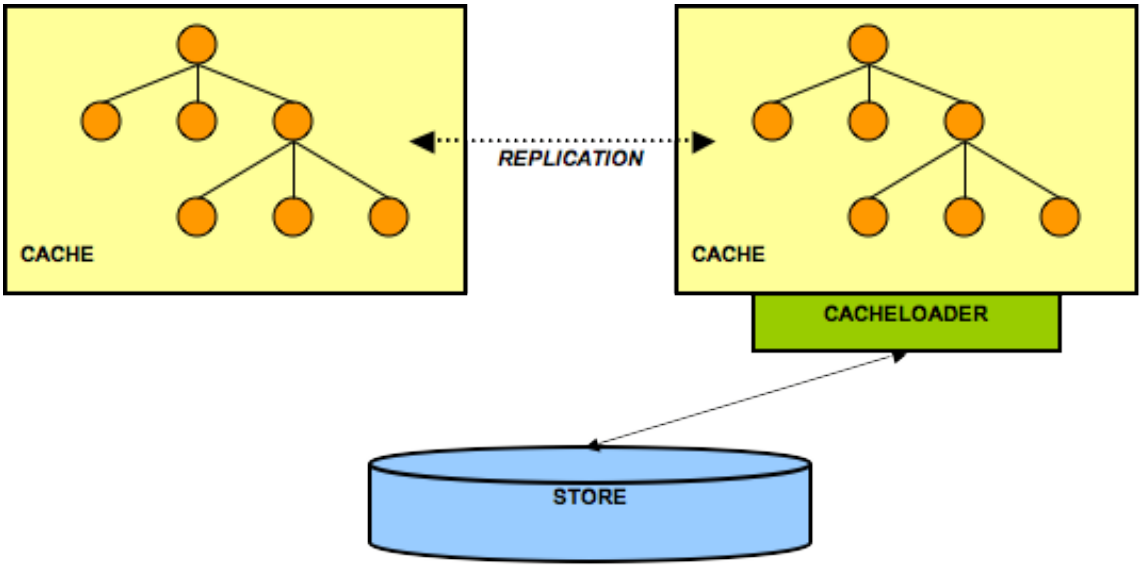


Figure 9.3. 2 nodes but only one accesses the backend store

This is a similar case to the previous one, but here only one node in the cluster interacts with a backend store via its cache loader. All other nodes perform in-memory replication. The idea here is all application state is kept in memory in each node, with the existence of multiple caches making the data highly available. (This assumes that a client that needs the data is able to somehow fail over from one cache to another.) The single persistent backend store then provides a backup copy of the data in case all caches in the cluster fail or need to be restarted.

Note that here it may make sense for the cache loader to store changes asynchronously, that is *not* on the caller's thread, in order not to slow down the cluster by accessing (for example) a database. This is a non-issue when using asynchronous replication.

A weakness with this architecture is that the cache with access to the cache loader becomes a single point of failure. Furthermore, if the cluster is restarted, the cache with the cache loader must be started first (easy to forget). A solution to the first problem is to configure a cache loader on each node, but set the **`singletonStore`** configuration to **`true`**. With this kind of setup, one but only one node will always be writing to a persistent store. However, this complicates the restart problem, as before restarting you need to determine which cache was writing before the shutdown/failure and then start that cache first.

9.5.4. Replicated Caches With Each Cache Having Its Own Store

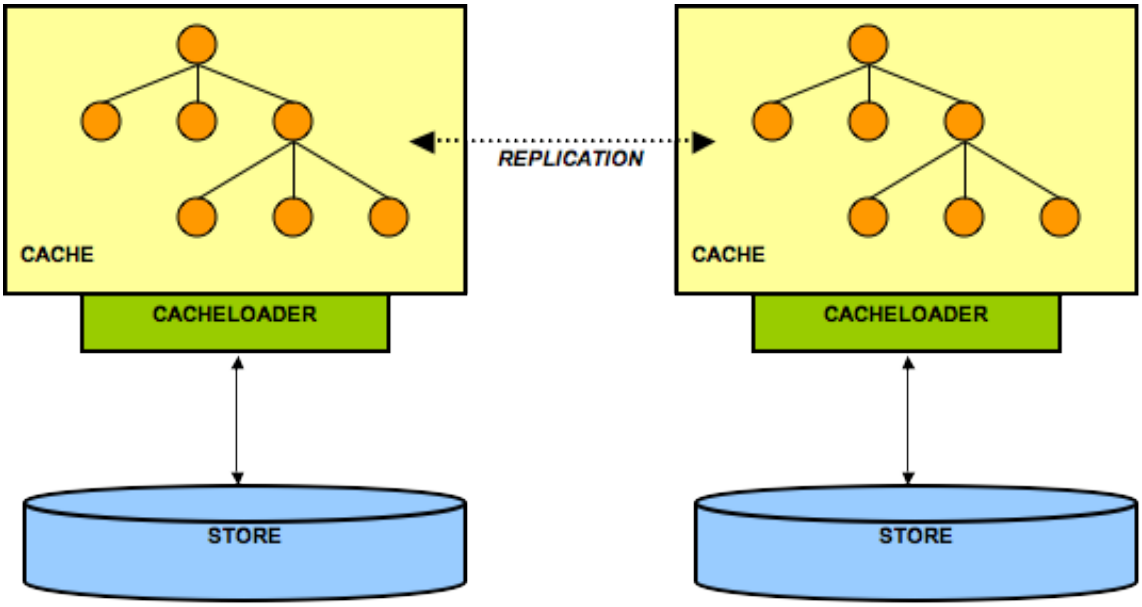


Figure 9.4. 2 nodes each having its own backend store

Here, each node has its own datastore. Modifications to the cache are (a) replicated across the cluster and (b) persisted using the cache loader. This means that all datastores have exactly the same state. When replicating changes synchronously and in a transaction, the two-phase commit protocol takes care that all modifications are replicated and persisted in each datastore, or none is replicated and persisted (atomic updates).

Note that JBoss Cache is *not* an XA Resource, that means it doesn't implement recovery. When used with a transaction manager that supports recovery, this functionality is not available.

The challenge here is state transfer: when a new node starts it needs to do the following:

1. Tell the coordinator (oldest node in a cluster) to send it the state. This is always a full state transfer, overwriting any state that may already be present.
2. The coordinator then needs to wait until all in-flight transactions have completed. During this time, it will not allow for new transactions to be started.
3. Then the coordinator asks its cache loader for the entire state using **`loadEntireState()`** . It then sends back that state to the new node.
4. The new node then tells its cache loader to store that state in its store, overwriting the old state. This is the **`CacheLoader.storeEntireState()`** method
5. As an option, the transient (in-memory) state can be transferred as well during the state transfer.
6. The new node now has the same state in its backend store as everyone else in the cluster, and modifications received from other nodes will now be persisted using the local cache loader.

9.5.5. Hierarchical Caches

If you need to set up a hierarchy within a single JVM, you can use the **`LocalDelegatingCacheLoader`** . This type of hierarchy can currently only be set up programmatically.

Hierarchical caches could also be set up spanning more than one JVM or server, using the **`TcpDelegatingCacheLoader`** .

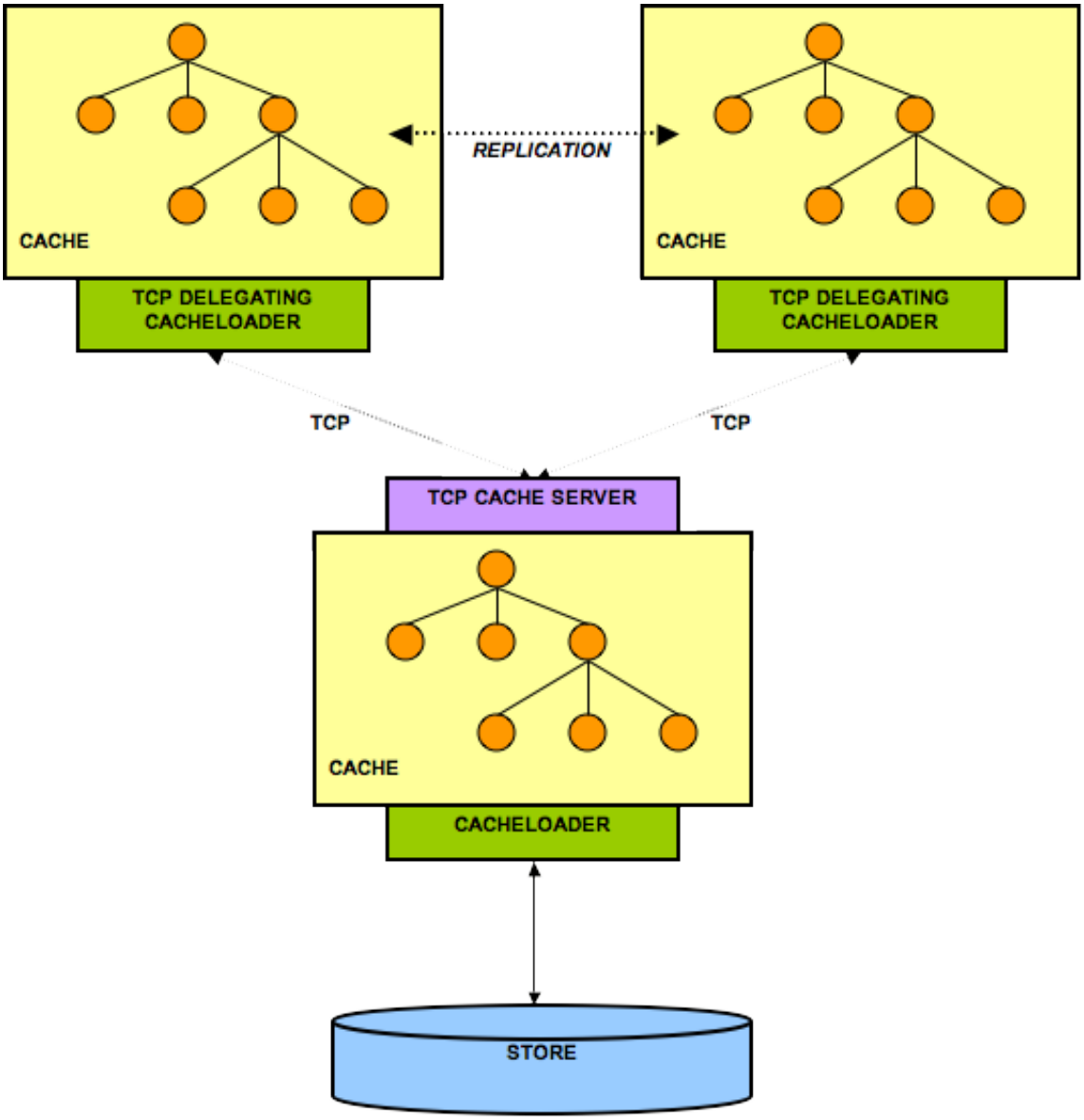


Figure 9.5. TCP delegating cache loader

9.5.6. Multiple Cache Loaders

You can set up more than one cache loader in a chain. Internally, a delegating **`ChainingCacheLoader`** is used, with references to each cache loader you have configured. Use cases vary depending on the type of cache loaders used in the chain. One example is using a filesystem based cache loader, co-located on the same host as the JVM, used as an overflow for memory. This ensures data is available relatively easily and with low cost. An additional remote cache loader, such as a **`TcpDelegatingCacheLoader`** provides resilience between server restarts.

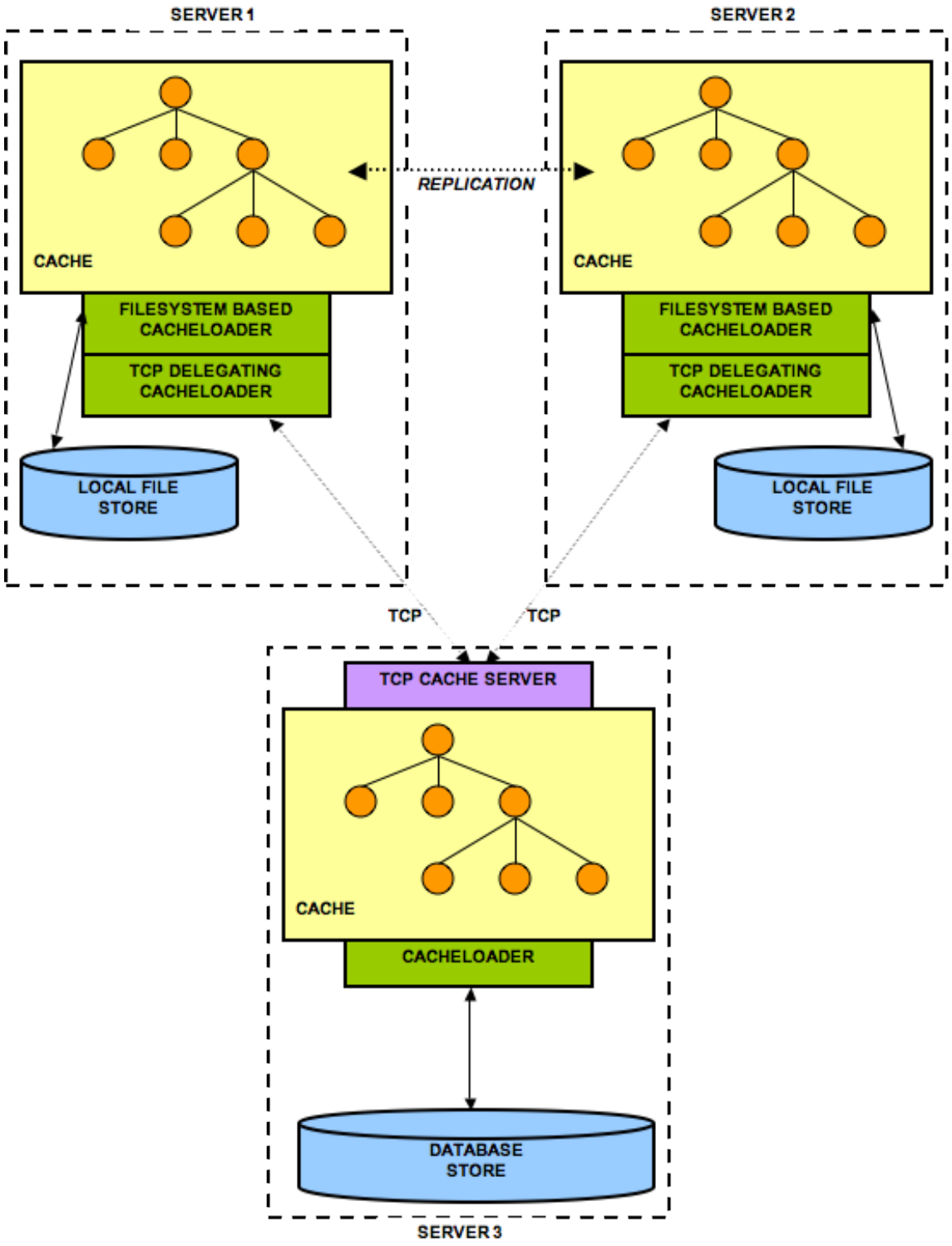


Figure 9.6. Multiple cache loaders in a chain

CHAPTER 10. EVICTION

Eviction controls JBoss Cache's memory management by restricting how many nodes are allowed to be stored in memory, and for how long. Memory constraints on servers mean caches cannot grow indefinitely, so eviction needs to occur to prevent out of memory errors. Eviction is most often used alongside Chapter 9, *Cache Loaders*.

10.1. DESIGN

Eviction in JBoss Cache is designed around four concepts:

- 1. Collecting statistics
- 2. Determining which nodes to evict
- 3. How nodes are evicted
- 4. Eviction threads.

In addition, Regions play a key role in eviction, as eviction is always configured on a per-region basis so that different subtrees in the cache can have different eviction characteristics.

10.1.1. Collecting Statistics

This is done on the caller's thread whenever anyone interacts with the cache. If eviction is enabled, an **`EvictionInterceptor`** is added to the interceptor chain and events are recorded in an event queue. Events are denoted by the **`EvictionEvent`** class. Event queues are held on specific Regions so each region has its own event queue.

This aspect of eviction is not configurable, except that the **`EvictionInterceptor`** is either added to the interceptor chain or not, depending on whether eviction is enabled.

10.1.2. Determining Which Nodes to Evict

An **`EvictionAlgorithm`** implementation processes the eviction queue to decide which nodes to evict. JBoss Cache ships with a number of implementations, including **`FIFOAlgorithm`**, **`LRUAlgorithm`**, **`LFUAlgorithm`**, etc. Each implementation has a corresponding **`EvictionAlgorithmConfig`** implementation with configuration details for the algorithm.

Custom **`EvictionAlgorithm`** implementations can be provided by implementing the interface or extending one of the provided implementations.

Algorithms are executed by calling its **`process()`** method and passing in the event queue to process. This is typically done by calling **`Region.processEvictionQueues()`**, which will locate the Algorithm assigned to the region.

10.1.3. How Nodes are Evicted

Once the **`EvictionAlgorithm`** decides which nodes to evict, it uses an implementation of **`EvictionActionPolicy`** to determine how to evict nodes. This is configurable on a per-region basis, and defaults to **`DefaultEvictionActionPolicy`**, which invokes **`Cache.evict()`** for each node that needs to be evicted.

JBoss Cache also ships with **`RemoveOnEvictActionPolicy`**, which calls **`Cache.removeNode()`** for each node that needs to be evicted, instead of **`Cache.evict()`**.

Custom **`EvictionActionPolicy`** implementations can be used as well.

10.1.4. Eviction threads

By default, a single cache-wide eviction thread is used to periodically iterate through registered regions and call **`Region.processEvictionQueues()`** on each region. The frequency with which this thread runs can be configured using the **`wakeUpInterval`** attribute in the **`eviction`** configuration element, and defaults to 5000 milliseconds if not specified.

The eviction thread can be disabled by setting **`wakeUpInterval`** to **`0`**. This can be useful if you have your own periodic maintenance thread running and would like to iterate through regions and call **`Region.processEvictionQueues()`** yourself.

10.2. EVICTION REGIONS

The concept of regions and the **`Region`** class were Section 7.6, “Class Loading and Regions” when talking about marshalling. Regions are also used to define the eviction behavior for nodes within that region. In addition to using a region-specific configuration, you can also configure default, cache-wide eviction behavior for nodes that do not fall into predefined regions or if you do not wish to define specific regions. It is important to note that when defining regions using the configuration XML file, all elements of the **`Fqn`** that defines the region are **`String`** objects.

For each region, you can define eviction parameters.

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, in order to handle scenarios like this consistently, will always choose the first region it encounters. In this way, if the algorithm needed to decide how to handle node **`/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`**.

10.2.1. Resident Nodes

Nodes marked as resident (using **`Node.setResident()`** API) will be ignored by the eviction policies both when checking whether to trigger the eviction and when proceeding with the actual eviction of nodes. E.g. if a region is configured to have a maximum of 10 nodes, resident nodes won't be counted when deciding whether to evict nodes in that region. In addition, resident nodes will not be considered for eviction when the region's eviction threshold is reached.

In order to mark a node as resident the **`Node.setResident()`** API should be used. By default, the newly created nodes are not resident. The **`resident`** attribute of a node is neither replicated, persisted nor transaction-aware.

A sample use case for resident nodes would be ensuring "path" nodes don't add "noise" to an eviction policy. E.g.,:

```
...
   Map lotsOfData = generateData();
   cache.put("/a/b/c", lotsOfData);
   cache.getRoot().getChild("/a").setResident(true);
   cache.getRoot().getChild("/a/b").setResident(true);
...
```

In this example, the nodes **/a** and **/a/b** are paths which exist solely to support the existence of node **/a/b/c** and don't hold any data themselves. As such, they are good candidates for being marked as resident. This would lead to better memory management as no eviction events would be generated when accessing **/a** and**/a/b**.

NOTE

When adding attributes to a resident node, e.g. **cache.put("/a", "k", "v")** in the above example, it would make sense to mark the nodes as non-resident again and let them be considered for eviction.

10.3. CONFIGURING EVICTION

10.3.1. Basic Configuration

The basic eviction configuration element looks like:

```
...
<eviction wakeUpInterval="500" eventQueueSize="100000">
   <default algorithmClass="org.jboss.cache.eviction.LRUAlgorithm">
      <property name="maxNodes" value="5000" />
      <property name="timeToLive" value="1000" />
   </default>
</eviction>
...
```

- **wakeUpInterval** - this required parameter defines how often the eviction thread runs, in milliseconds.
- **eventQueueSize** - this optional parameter defines the size of the bounded queue which holds eviction events. If your eviction thread does not run often enough, you may find that the event queue fills up. It may then be necessary to get your eviction thread to run more frequently, or increase the size of your event queue. This configuration is just the *default* event queue size, and can be overridden in specific eviction regions. If not specified, this defaults to **200000**.
- **algorithmClass** - this is required, unless you set individual **algorithmClass** attributes on each and every region. This defines the default eviction algorithm to use if one is not defined for a region.
- Algorithm configuration attributes - these are specific to the algorithm specified in **algorithmClass**. See the section specific to the algorithm you are interested in for details.

10.3.2. Programmatic Configuration

Configuring eviction using the **Configuration** object entails the use of the **org.jboss.cache.config.EvictionConfig** bean, which is passed into **Configuration.setEvictionConfig()**. See the Chapter 3, *Configuration* for more on building a **Configuration** programatically.

The use of simple POJO beans to represent all elements in a cache's configuration also makes it fairly easy to programatically add eviction regions after the cache is started. For example, assume we had an existing cache configured via XML with the EvictionConfig element shown above. Now at runtime we

wished to add a new eviction region named "/org/jboss/fifo", using **`LRUAlgorithm`** but a different number of **`maxNodes`**:

```
Fqn fqn = Fqn.fromString("/org/jboss/fifo");

// Create a configuration for an LRUPolicy
LRUAlgorithmConfig lruc = new LRUAlgorithmConfig();
lruc.setMaxNodes(10000);

// Create an eviction region config
EvictionRegionConfig erc = new EvictionRegionConfig(fqn, lruc);

// Create the region and set the config
Region region = cache.getRegion(fqn, true);
region.setEvictionRegionConfig(erc);
```

10.4. SHIPPED EVICTION POLICIES

This section details the different algorithms shipped with JBoss Cache, and the various configuration parameters used for each algorithm.

10.4.1. LRUAlgorithm - Least Recently Used

`org.jboss.cache.eviction.LRUAlgorithm` controls both the node lifetime and age. This policy guarantees a constant order (**`O (1)`**) for adds, removals and lookups (visits). It has the following configuration parameters:

- **`maxNodes`** - This is the maximum number of nodes allowed in this region. 0 denotes immediate expiry, -1 denotes no limit.
- **`timeToLive`** - The amount of time a node is not written to or read (in milliseconds) before the node is swept away. 0 denotes immediate expiry, -1 denotes no limit.
- **`maxAge`** - Lifespan of a node (in milliseconds) regardless of idle time before the node is swept away. 0 denotes immediate expiry, -1 denotes no limit.
- **`minTimeToLive`** - 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.

10.4.2. FIFOAlgorithm - First In, First Out

`org.jboss.cache.eviction.FIFOAlgorithm` controls the eviction in a proper first in first out order. This policy guarantees a constant order (**`O (1)`**) for adds, removals and lookups (visits). It has the following configuration parameters:

- **`maxNodes`** - This is the maximum number of nodes allowed in this region. 0 denotes immediate expiry, -1 denotes no limit.
- **`minTimeToLive`** - 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.

10.4.3. MRUAlgorithm - Most Recently Used

`org.jboss.cache.eviction.MRUAlgorithm` controls the eviction in based on most recently used algorithm. The most recently used nodes will be the first to evict with this policy. This policy guarantees a constant order (**`O (1)`**) for adds, removals and lookups (visits). It has the following configuration parameters:

- **`maxNodes`** - This is the maximum number of nodes allowed in this region. 0 denotes immediate expiry, -1 denotes no limit.
- **`minTimeToLive`** - 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.

10.4.4. LFUAlgorithm - Least Frequently Used

`org.jboss.cache.eviction.LFUAlgorithm` controls the eviction in based on least frequently used algorithm. The least frequently used nodes will be the first to evict with this policy. Node usage starts at 1 when a node is first added. Each time it is visited, the node usage counter increments by 1. This number is used to determine which nodes are least frequently used. LFU is also a sorted eviction algorithm. The underlying EvictionQueue implementation and algorithm is sorted in ascending order of the node visits counter. This class guarantees a constant order (**`O (1)`**) for adds, removal and searches. However, when any number of nodes are added/visited to the queue for a given processing pass, a single quasilinear (**`O (n * log n)`**) operation is used to resort the queue in proper LFU order. Similarly if any nodes are removed or evicted, a single linear (**`O (n)`**) pruning operation is necessary to clean up the EvictionQueue. LFU has the following configuration parameters:

- **`maxNodes`** - This is the maximum number of nodes allowed in this region. 0 denotes immediate expiry, -1 denotes no limit.
- **`minNodes`** - This is the minimum number of nodes allowed in this region. This value determines what the eviction queue should prune down to per pass. e.g. If minNodes is 10 and the cache grows to 100 nodes, the cache is pruned down to the 10 most frequently used nodes when the eviction timer makes a pass through the eviction algorithm.
- **`minTimeToLive`** - 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.

10.4.5. ExpirationAlgorithm

`org.jboss.cache.eviction.ExpirationAlgorithm` is a policy that evicts nodes based on an absolute expiration time. The expiration time is indicated using the **`org.jboss.cache.Node.put()`** method, using a String key **`expiration`** and the absolute time as a **`java.lang.Long`** object, with a value indicated as milliseconds past midnight January 1st, 1970 UTC (the same relative time as provided by **`java.lang.System.currentTimeMillis()`**).

This policy guarantees a constant order (**`O (1)`**) for adds and removals. Internally, a sorted set (TreeSet) containing the expiration time and Fqn of the nodes is stored, which essentially functions as a heap.

This policy has the following configuration parameters:

- **`expirationKeyName`** - This is the Node key name used in the eviction algorithm. The configuration default is **`expiration`** .

- **maxNodes** - This is the maximum number of nodes allowed in this region. 0 denotes immediate expiry, -1 denotes no limit.

The following listing shows how the expiration date is indicated and how the policy is applied:

```
   Cache cache = DefaultCacheFactory.createCache();
   Fqn fqn1 = Fqn.fromString("/node/1");
   Long future = new Long(System.currentTimeMillis() + 2000);

   // sets the expiry time for a node

cache.getRoot().addChild(fqn1).put(ExpirationConfiguration.EXPIRATION_KEY,
future);

   assertTrue(cache.getRoot().hasChild(fqn1));
   Thread.sleep(5000);

   // after 5 seconds, expiration completes
   assertFalse(cache.getRoot().hasChild(fqn1));
```

Note that the expiration time of nodes is only checked when the region manager wakes up every **wakeUpIntervalSeconds** , so eviction may happen a few seconds later than indicated.

10.4.6. ElementSizeAlgorithm - Eviction based on number of key/value pairs in a node

org.jboss.cache.eviction.ElementSizeAlgorithm controls the eviction in based on the number of key/value pairs in the node. Nodes The most recently used nodes will be the first to evict with this policy. It has the following configuration parameters:

- **maxNodes** - This is the maximum number of nodes allowed in this region. 0 denotes immediate expiry, -1 denotes no limit.
- **maxElementsPerNode** - This is the trigger number of attributes per node for the node to be selected for eviction. 0 denotes immediate expiry, -1 denotes no limit.
- **minTimeToLive** - 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.

CHAPTER 11. TRANSACTIONS AND CONCURRENCY

11.1. CONCURRENT ACCESS

JBoss Cache is a thread safe caching API, and uses its own efficient mechanisms of controlling concurrent access. It uses an innovative implementation of multi-versioned concurrency control (MVCC) as the default locking scheme. Versions of JBoss Cache prior to 3.x offered Optimistic and Pessimistic Locking schemes, both of which are now deprecated in favor of MVCC.

11.1.1. Multi-Version Concurrency Control (MVCC)

MVCC is a locking scheme commonly used by modern database implementations to control fast, safe concurrent access to shared data.

11.1.1.1. MVCC Concepts

MVCC is designed to provide the following features for concurrent access:

- Readers that don't block writers
- Writers that fail fast

and achieves this by using data versioning and copying for concurrent writers. The theory is that readers continue reading shared state, while writers copy the shared state, increment a version id, and write that shared state back after verifying that the version is still valid (i.e., another concurrent writer has not changed this state first).

This allows readers to continue reading while not preventing writers from writing, and repeatable read semantics are maintained by allowing readers to read off the old version of the state.

11.1.1.2. MVCC Implementation

JBoss Cache's implementation of MVCC is based on a few features:

- Readers don't acquire any locks
- Only one additional version is maintained for shared state, for a single writer
- All writes happen sequentially, to provide fail-fast semantics

The extremely high performance of JBoss Cache's MVCC implementation for reading threads is achieved by not requiring any synchronization or locking for readers. For each reader thread, the **`MVCCLockingInterceptor`** wraps state in a lightweight container object, which is placed in the thread's **`InvocationContext`** (or **`TransactionContext`** if running in a transaction). All subsequent operations on the state happens via the container object. This use of Java references allows for repeatable read semantics even if the actual state changes simultaneously.

Writer threads, on the other hand, need to acquire a lock before any writing can commence. Currently, we use lock striping to improve the memory performance of the cache, and the size of the shared lock pool can be tuned using the **`concurrencyLevel`** attribute of the **`locking`** element. See the Chapter 12, *Configuration References* for details. After acquiring an exclusive lock on an Fqn, the writer thread then wraps the state to be modified in a container as well, just like with reader threads, and then copies this state for writing. When copying, a reference to the original version is still maintained in the container (for rollbacks). Changes are then made to the copy and the copy is finally written to the data structure when the write completes.

This way, subsequent readers see the new version while existing readers still hold a reference to the original version in their context.

If a writer is unable to acquire the write lock after some time, a **`TimeoutException`** is thrown. This lock acquisition timeout defaults to 10000 millis and can be configured using the **`lockAcquisitionTimeout`** attribute of the **`locking`** element. See the Chapter 12, *Configuration References* for details.

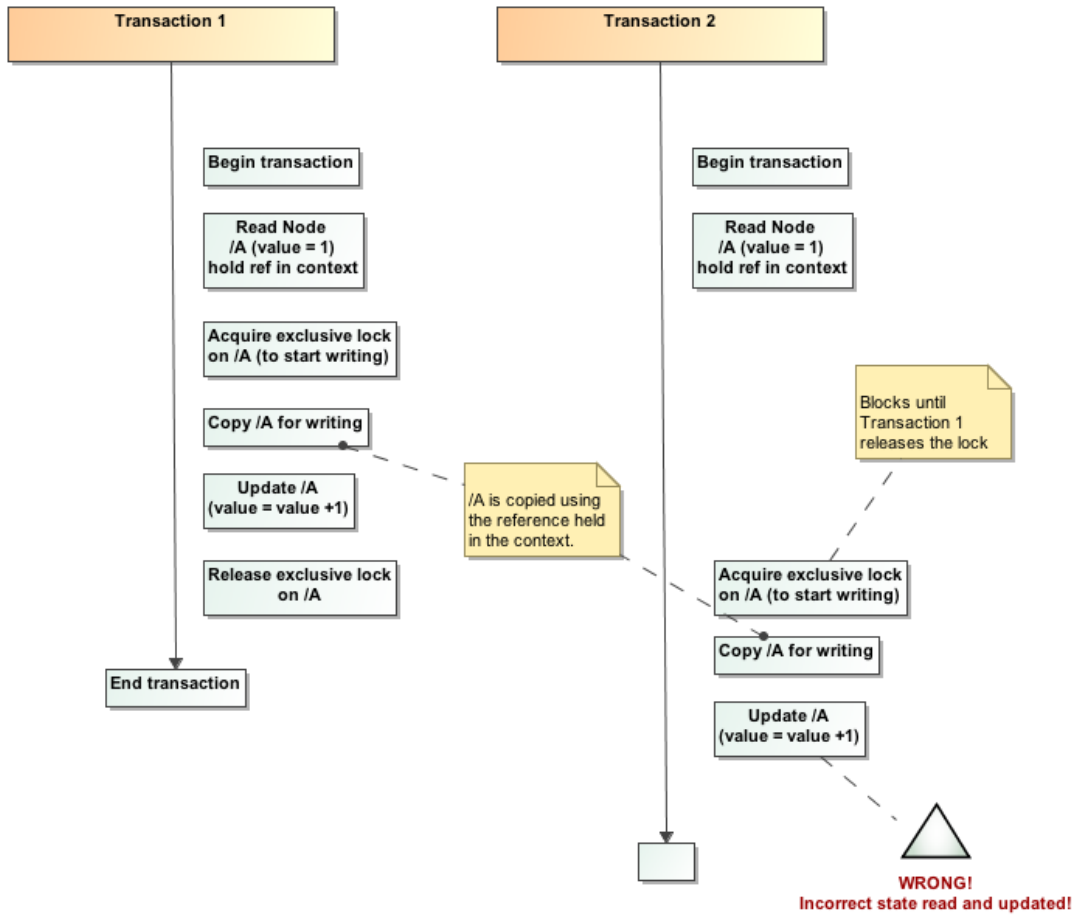
11.1.1.2.1. Isolation Levels

JBoss Cache 3.x supports two isolation levels: REPEATABLE_READ and READ_COMMITTED, which correspond in semantic to database-style isolation levels. Previous versions of JBoss Cache supported all 5 database isolation levels, and if an unsupported isolation level is configured, it is either upgraded or downgraded to the closest supported level.

REPEATABLE_READ is the default isolation level, to maintain compatibility with previous versions of JBoss Cache. READ_COMMITTED, while providing a slightly weaker isolation, has a significant performance benefit over REPEATABLE_READ.

11.1.1.2.2. Concurrent Writers and Write-Skews

Although MVCC forces writers to obtain a write lock, a phenomenon known as write skews may occur when using REPEATABLE_READ:



This happens when concurrent transactions performing a read and then a write, based on the value that was read. Since reads involve holding on to the reference to the state in the transaction context, a subsequent write would work off that original state read, which may now be stale.

The default behavior with dealing with a write skew is to throw a **`DataVersioningException`**, when it is detected when copying state for writing. However, in most applications, a write skew may not be an issue (for example, if the state written has no relationship to the state originally read) and should be allowed. If your application does not care about write skews, you can allow them to happen by setting the **`writeSkewCheck`** configuration attribute to **`false`**. See the Chapter 12, *Configuration References* for details.

Note that write skews cannot happen when using READ_COMMITTED since threads always work off committed state.

11.1.1.3. Configuring Locking

Configuring MVCC involves using the **`<locking />`** configuration tag, as follows:

```
<locking
   isolationLevel="REPEATABLE_READ"
   lockAcquisitionTimeout="10234"
   nodeLockingScheme="mvcc"
   writeSkewCheck="false"
   concurrencyLevel="1000" />
```

- **`nodeLockingScheme`** - the node locking scheme used. Defaults to MVCC if not provided, deprecated schemes such as **`pessimistic`** or **`optimistic`** may be used but is not encouraged.
- **`isolationLevel`** - transaction isolation level. Defaults to REPEATABLE_READ if not provided.
- **`writeSkewCheck`** - defaults to **`true`** if not provided.
- **`concurrencyLevel`** - defaults to 500 if not provided.
- **`lockAcquisitionTimeout`** - only applies to writers when using MVCC. Defaults to 10000 if not provided.

11.1.2. Pessimistic and Optimistic Locking Schemes

From JBoss Cache 3.x onwards, pessimistic and optimistic locking schemes are deprecated in favor of Section 11.1.1, “Multi-Version Concurrency Control (MVCC)”. It is recommended that existing applications move off these legacy locking schemes as support for them will eventually be dropped altogether in future releases.

Documentation for legacy locking schemes are not included in this user guide, and if necessary, can be referenced in previous versions of this document, which can be found on the JBoss Cache website.

11.2. JTA SUPPORT

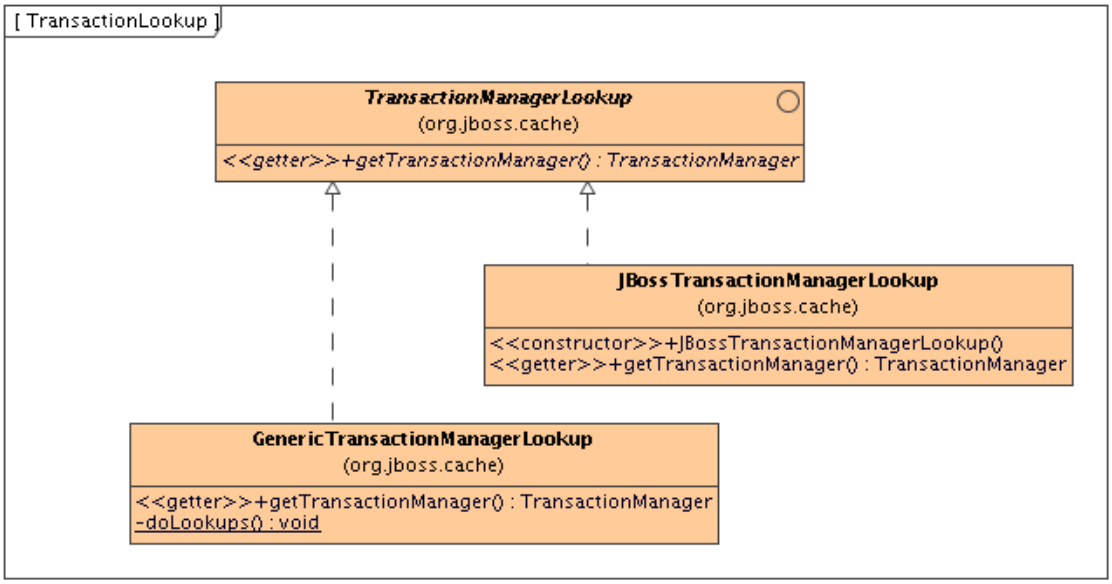
JBoss Cache can be configured to use and participate in JTA compliant transactions. Alternatively, if transaction support is disabled, it is equivalent to using autocommit in JDBC calls, where modifications are potentially replicated after every change (if replication is enabled).

What JBoss Cache does on every incoming call is:

1. Retrieve the current **`javax.transaction.Transaction`** associated with the thread

2. If not already done, register a **`javax.transaction.Synchronization`** with the transaction manager to be notified when a transaction commits or is rolled back.

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



JBoss Cache ships with **`JBossTransactionManagerLookup`** and **`GenericTransactionManagerLookup`**. The **`JBossTransactionManagerLookup`** is able to bind to a running JBoss AS instance and retrieve a **`TransactionManager`** while the **`GenericTransactionManagerLookup`** is able to bind to most popular Java EE application servers and provide the same functionality. A dummy implementation - **`DummyTransactionManagerLookup`** - is also provided for unit tests. Being a dummy, this is not recommended for production use a it has some severe limitations to do with concurrent transactions and recovery.

An alternative to configuring a **`TransactionManagerLookup`** is to programatically inject a reference to the **`TransactionManager`** into the **`Configuration`** object's **`RuntimeConfig`** element:

```
TransactionManager tm = getTransactionManager(); // magic method
cache.getConfiguration().getRuntimeConfig().setTransactionManager(tm);
```

Injecting the **`TransactionManager`** is the recommended approach when the **`Configuration`** is built by some sort of IOC container that already has a reference to the **`TransactionManager`**.

When the transaction commits, we initiate either a one-phase or two-phase commit protocol. See Section 8.1.2.1, “Replicated Caches and Transactions” for details.

PART III. JBOSS CACHE CONFIGURATION REFERENCES

This section contains technical references for easy looking up.

CHAPTER 12. CONFIGURATION REFERENCES

12.1. SAMPLE XML CONFIGURATION FILE

This is what a typical XML configuration file looks like. It is recommended that you use one of the configurations shipped with the JBoss Cache distribution and tweak according to your needs rather than write one from scratch.

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

<jbosscache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="urn:jboss:jbosscache-core:config:3.1">


   <!--
      isolation levels supported: READ_COMMITTED and REPEATABLE_READ
      nodeLockingSchemes: mvcc, pessimistic (deprecated), optimistic
(deprecated)
   -->
   <locking
         isolationLevel="REPEATABLE_READ"
         lockParentForChildInsertRemove="false"
         lockAcquisitionTimeout="20000"
         nodeLockingScheme="mvcc"
         writeSkewCheck="false"
         useLockStriping="true"
         concurrencyLevel="500"/>

   <!--
      Used to register a transaction manager and participate in ongoing
transactions.
      -->
   <transaction

transactionManagerLookupClass="org.jboss.cache.transaction.GenericTransact
ionManagerLookup"
         syncRollbackPhase="false"
         syncCommitPhase="false"/>

   <!--
      Used to register JMX statistics in any available MBean server
      -->
   <jmxStatistics
         enabled="false"/>

   <!--
      If region based marshalling is used, defines whether new regions are
inactive on startup.
   -->
   <startup
         regionsInactiveOnStartup="true"/>

   <!--
      Used to register JVM shutdown hooks.
      hookBehavior: DEFAULT, REGISTER, DONT_REGISTER
```

```
   -->
   <shutdown
         hookBehavior="DEFAULT"/>

   <!--
      Used to define async listener notification thread pool size
   -->
   <listeners
         asyncPoolSize="1"
         asyncQueueSize="100000"/>

   <!--
      Used to enable invocation batching and allow the use of
Cache.startBatch()/endBatch() methods.
   -->
   <invocationBatching
         enabled="false"/>

   <!--
      serialization related configuration, used for replication and cache
loading
   -->
   <serialization
         objectInputStreamPoolSize="12"
         objectOutputStreamPoolSize="14"
         version="3.0.0"
         marshallerClass="org.jboss.cache.marshall.VersionAwareMarshaller"
         useLazyDeserialization="false"
         useRegionBasedMarshalling="false"/>

   <!--
      This element specifies that the cache is clustered.
      modes supported: replication (r) or invalidation (i).
   -->
   <clustering mode="replication" clusterName="JBossCache-cluster">

      <!--
         Defines whether to retrieve state on startup
      -->
      <stateRetrieval timeout="20000" fetchInMemoryState="false"/>

      <!--
         Network calls are synchronous.
      -->
      <sync replTimeout="20000"/>
      <!--
         Uncomment this for async replication.
      -->
      <!--<async useReplQueue="true" replQueueInterval="10000"
replQueueMaxElements="500" serializationExecutorPoolSize="20"
serializationExecutorQueueSize="5000000"/>-->

      <!-- Uncomment to use Buddy Replication -->
      <!--
      <buddy enabled="true" poolName="myBuddyPoolReplicationGroup"
communicationTimeout="2000">
```

```
         <dataGravitation auto="true" removeOnFind="true"
searchBackupTrees="true"/>
         <locator
class="org.jboss.cache.buddyreplication.NextMemberBuddyLocator">
            <properties>
               numBuddies = 1
               ignoreColocatedBuddies = true
            </properties>
         </locator>
      </buddy>
      -->

      <!--
         Configures the JGroups channel.  Looks up a JGroups config file
on the classpath or filesystem.  udp.xml
         ships with jgroups.jar and will be picked up by the class loader.
      -->
      <jgroupsConfig configFile="udp.xml">
         <!-- uncomment to define a JGroups stack here

         <PING timeout="2000" num_initial_members="3"/>
         <MERGE2 max_interval="30000" min_interval="10000"/>
         <FD_SOCK/>
         <FD timeout="10000" max_tries="5" shun="true"/>
         <VERIFY_SUSPECT timeout="1500"/>
         <pbcast.NAKACK use_mcast_xmit="false" gc_lag="0"
                        retransmit_timeout="300,600,1200,2400,4800"
                        discard_delivered_msgs="true"/>
         <UNICAST timeout="300,600,1200,2400,3600"/>
         <pbcast.STABLE stability_delay="1000" desired_avg_gossip="50000"
                        max_bytes="400000"/>
         <pbcast.GMS print_local_addr="true" join_timeout="5000"
shun="false"
                     view_bundling="true"
view_ack_collection_timeout="5000"/>
         <FRAG2 frag_size="60000"/>
         <pbcast.STREAMING_STATE_TRANSFER use_reading_thread="true"/>
         <pbcast.FLUSH timeout="0"/>
          -->
      </jgroupsConfig>
   </clustering>

   <!--
      Eviction configuration.  WakeupInterval defines how often the
eviction thread runs, in milliseconds.  0 means
      the eviction thread will never run.
   -->
   <eviction wakeUpInterval="500">
      <default algorithmClass="org.jboss.cache.eviction.LRUAlgorithm"
eventQueueSize="200000">
         <property name="maxNodes" value="5000" />
         <property name="timeToLive" value="1000" />
      </default>
      <region name="/org/jboss/data1">
         <property name="timeToLive" value="2000" />
      </region>
```

```
      <region name="/org/jboss/data2"
algorithmClass="org.jboss.cache.eviction.FIFOAlgorithm"
eventQueueSize="100000">
         <property name="maxNodes" value="3000" />
         <property name="minTimeToLive" value="4000" />
      </region>
   </eviction>

   <!--
      Cache loaders.

      If passivation is enabled, state is offloaded to the cache loaders
ONLY when evicted.  Similarly, when the state
      is accessed again, it is removed from the cache loader and loaded
into memory.

      Otherwise, state is always maintained in the cache loader as well as
in memory.

      Set 'shared' to true if all instances in the cluster use the same
cache loader instance, e.g., are talking to the
      same database.
   -->
   <loaders passivation="false" shared="false">
      <preload>
         <node fqn="/org/jboss"/>
         <node fqn="/org/tempdata"/>
      </preload>

      <!--
         we can have multiple cache loaders, which get chained
      -->
      <loader class="org.jboss.cache.loader.JDBCCacheLoader" async="true"
fetchPersistentState="true"
               ignoreModifications="true" purgeOnStartup="true">
         <properties>
            cache.jdbc.table.name=jbosscache
            cache.jdbc.table.create=true
            cache.jdbc.table.drop=true
         </properties>
         <singletonStore enabled="true"
class="org.jboss.cache.loader.SingletonStoreCacheLoader">
            <properties>
               pushStateWhenCoordinator=true
               pushStateWhenCoordinatorTimeout=20000
            </properties>
         </singletonStore>
      </loader>
   </loaders>

   <!--
      Define custom interceptors.  All custom interceptors need to extend
org.jboss.cache.interceptors.base.CommandInterceptor
   -->
   <!--
   <customInterceptors>
```

```
      <interceptor position="first"

class="org.jboss.cache.config.parsing.custominterceptors.AaaCustomIntercep
tor">
         <property name="attrOne" value="value1" />
         <property name="attrTwo" value="value2" />
      </interceptor>
      <interceptor position="last"

class="org.jboss.cache.config.parsing.custominterceptors.BbbCustomIntercep
tor"/>
      <interceptor index="3"

class="org.jboss.cache.config.parsing.custominterceptors.AaaCustomIntercep
tor"/>
      <interceptor before="org.jboss.cache.interceptors.CallInterceptor"

class="org.jboss.cache.config.parsing.custominterceptors.BbbCustomIntercep
tor"/>
      <interceptor after="org.jboss.cache.interceptors.CallInterceptor"

class="org.jboss.cache.config.parsing.custominterceptors.AaaCustomIntercep
tor"/>
   </customInterceptors>
   -->
</jbosscache>
```

12.1.1. XML validation

Configuration XML files are validated using an XSD schema. This schema is included in **`jbosscache-core.jar`** and is also available online: **`http://www.jboss.org/jbosscache/jbosscache-config-3.0.xsd`**. Most IDEs and XML authoring tools will be able to use this schema to validate your configuration file as you write it.

JBoss Cache also validates your configuration file when you start up, and will throw an exception if it encounters an invalid file. You can suppress this behavior by passing in **`-Djbosscache.config.validate=false`** to your JVM when you start up. Alternatively, you can point the validator to a different schema by passing in **`-Djbosscache.config.schemaLocation=url`**.

12.2. CONFIGURATION FILE QUICK REFERENCE

A list of definitions of each of the XML elements attributes used above, and their bean counterparts for programmatic configuration. If the description of an attribute states that it is *dynamic*, that means it can be changed after the cache is created and started.

Table 12.1. The `<jbosscache />` Element

The `<jbosscache />` Element	
Description	This is the root element for the JBoss Cache configuration file. This is the only mandatory element in a valid JBoss Cache configuration file.

The <jbosscache /> Element	
Parent	none (is root element)
Children	Table 12.40, "The <clustering /> Element", Table 12.37, "The <customInterceptors /> Element", Table 12.19, "The <eviction /> Element", Table 12.15, "The <invocationBatching /> Element", Table 12.7, "The <jmxStatistics /> Element", Table 12.13, "The <listeners /> Element", Table 12.27, "The <loaders /> Element", Table 12.3, "The <locking /> Element", Table 12.17, "The <serialization /> Element", Table 12.11, "The <shutdown /> Element", Table 12.9, "The <startup /> Element", Table 12.5, "The <transaction /> Element"
Bean Equivalent	`Configuration`

Table 12.2. <jbosscache /> Attributes

<jbosscache /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
xmlns	-	urn:jboss:jbosscache-core:config:3.1	urn:jboss:jbosscache-core:config:3.1	Defines the XML namespace for all configuration entries.
xmlns:xsi	-	http://www.w3.org/2001/XMLSchema-instance	http://www.w3.org/2001/XMLSchema-instance	Defines the XML schema instance for the configuration.

Table 12.3. The <locking /> Element

The <locking /> Element	
Description	This element specifies locking behavior on the cache.
Parent	Table 12.1, "The <jbosscache /> Element"
Children	
Bean equivalent	`Configuration`

Table 12.4. `<locking />` Attributes

`<locking />` Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
isolationLevel	isolationLevel	READ_COMMITTED, REPEATABLE_READ	REPEATABLE_READ	The isolation level used for transactions.
lockParentForChildInsertRemove	lockParentForChildInsertRemove	true, false	false	Specifies whether parent nodes are locked when inserting or removing children. This can also be configured on a per-node basis (see **`Node.setLockForChildInsertRemove()`**
lockAcquisitionTimeout	lockAcquisitionTimeout (*dynamic*)	Any positive long value	10000	Length of time, in milliseconds, that a thread will try and acquire a lock. A **`TimeoutException`** is usually thrown if a lock cannot be acquired in this given timeframe. Can be overridden on a per-invocation basis using **`Option.setLockAcquisitionTimeout()`**
nodeLockingScheme (*deprecated*)	nodeLockingScheme	mvcc, pessimistic, optimistic	mvcc	Specifies the node locking scheme to be used.

`<locking />` Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
writeSkewCheck	writeSkewCheck	true, false	false	Specifies whether to check for write skews. Only used if **`nodeLockingScheme`** is **`mvcc`** and **`isolationLevel`** is **`REPEATABLE_READ`**. See the Section 11.1.1.2.2, "Concurrent Writers and Write-Skews" for a more detailed discussion.
useLockStriping	useLockStriping	true, false	true	Specifies whether lock striping is used. Only used if **`nodeLockingScheme`** is **`mvcc`**. Lock striping usually offers greater performance and better memory usage, although in certain cases deadlocks may occur where several Fqns map to the same shared lock. This can be mitigated by increasing your concurrency level, though the only concrete solution is to disable lock striping altogether.

<locking /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
concurrencyLevel	concurrencyLevel	Any positive integer; 0 not allowed.	500	Specifies the number of shared locks to use for write locks acquired. Only used if **nodeLockingScheme** is **mvcc**. See the Section 11.1.1.2, "MVCC Implementation" for a more detailed discussion.

Table 12.5. The <transaction /> Element

The <transaction /> Element	
Description	This element specifies transactional behavior on the cache.
Parent	Table 12.1, "The **<jbosscache />** Element"
Children	
Bean equivalent	**Configuration**

Table 12.6. <transaction /> Attributes

<transaction /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**

`<transaction />` Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
transactionManagerLookupClass	transactionManagerLookupClass	A valid class that is available on the classpath	none	Specifies the **`TransactionManagerLookupClass`** implementation to use to obtain a transaction manager. If not specified (and a **`TransactionManager`** is not injected using **`RuntimeConfig.setTransactionManager()`**), the cache will not be able to participate in any transactions.
syncCommitPhase	syncCommitPhase (*dynamic*)	true, false	false	If enabled, commit messages that are broadcast around a cluster are done so synchronously. This is usually of little value since detecting a failure in broadcasting a commit means little else can be done except log a message, since some nodes in a cluster may have already committed and cannot rollback.

<transaction /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
syncRollbackPhase	syncRollbackPhase (*dynamic*)	true, false	false	If enabled, rollback messages that are broadcast around a cluster are done so synchronously. This is usually of little value since detecting a failure in broadcasting a rollback means little else can be done except log a message, since some nodes in a cluster may have already committed and cannot rollback.

Table 12.7. The <jmxStatistics /> Element

The <jmxStatistics /> Element	
Description	This element specifies whether cache statistics are gathered and reported via JMX.
Parent	Table 12.1, "The <jbosscache /> Element"
Children	
Bean equivalent	`Configuration`

Table 12.8. <jmxStatistics /> Attributes

<jmxStatistics /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
enabled	exposeManagementStatistics	true, false	true	Controls whether cache statistics are gathered and exposed via JMX.

Table 12.9. The <startup /> Element

The <startup /> Element	
Description	This element specifies behavior when the cache starts up.
Parent	Table 12.1, "The <jbosscache /> Element"
Children	
Bean equivalent	`Configuration`

Table 12.10. <startup /> Attributes

<startup /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
regionsInactiveOnStartup	inactiveOnStartup	true, false	false	If Section 7.6, "Class Loading and Regions" is enabled, this attribute controls whether new regions created are inactive on startup.

Table 12.11. The <shutdown /> Element

The <shutdown /> Element	
Description	This element specifies behavior when the cache shuts down.
Parent	Table 12.1, "The <jbosscache /> Element"
Children	
Bean equivalent	`Configuration`

Table 12.12. <shutdown /> Attributes

<shutdown /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
hookBehavior	shutdownHookBehavior	DEFAULT, DONT_REGISTER, REGISTER	DEFAULT	This attribute determines whether the cache registers a JVM shutdown hook so that it can clean up resources if the JVM is receives a shutdown signal. By default a shutdown hook is registered if no MBean server (apart from the JDK default) is detected. REGSTER forces the cache to register a shutdown hook even if an MBean server is detected, and DONT_REGISTER forces the cache NOT to register a shutdown hook, even if no MBean server is detected.

Table 12.13. The <listeners /> Element

The <listeners /> Element	
Description	This element specifies behavior of registered cache listeners.
Parent	Table 12.1, "The <jbosscache /> Element"
Children	
Bean equivalent	`Configuration`

Table 12.14. <listeners /> Attributes

<listeners /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
asyncPoolSize	listenerAsyncPoolSize	integer	1	The size of the thread pool used to dispatch events to cache listeners that have registered as asynchronous listeners. If this number is less than 1, all asynchronous listeners will be treated as synchronous listeners and notified synchronously.
asyncQueueSize	listenerAsyncQueueSize	positive integer	50000	The size of the bounded queue used by the async listener thread pool. Only considered if **asyncPoolSize** is greater than 0. Increase this if you see a lot of threads blocking trying to add events to this queue.

Table 12.15. The <invocationBatching /> Element

The <invocationBatching /> Element	
Description	This element specifies behavior of invocation batching.
Parent	Table 12.1, "The **<jbosscache />** Element"
Children	
Bean equivalent	**Configuration**

Table 12.16. <invocationBatching /> Attributes

<invocationBatching /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
enabled	invocationBatchingEnabled	true, false	false	Whether invocation batching is enabled or not. See the chapter on Chapter 4, *Batching API* for details.

Table 12.17. The <serialization /> Element

The <serialization /> Element	
Description	This element specifies behavior of object serialization in JBoss Cache.
Parent	Table 12.1, "The <jbosscache /> Element"
Children	
Bean equivalent	**Configuration**

Table 12.18. <serialization /> Attributes

<serialization /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
marshallerClass	marshallerClass	A valid class that is available on the classpath	VersionAwareMarshaller	Specifies the marshaller to use when serializing and deserializing objects, either for replication or persistence.

<serialization /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
useLazyDeserialization	useLazyDeserialization	true, false	false	A mechanism by which serialization and deserialization of objects is deferred till the point in time in which they are used and needed. This typically means that any deserialization happens using the thread context class loader of the invocation that requires deserialization, and is an effective mechanism to provide classloader isolation.
useRegionBasedMarshalling (*deprecated*)	useRegionBasedMarshalling	true, false	false	An older mechanism by which classloader isolation was achieved, by registering classloaders on specific regions.

<serialization /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
version	replicationVersion	Valid JBoss Cache version string	Current version	Used by the **VersionAwareMarshaller** in determining which version stream parser to use by default when initiating communications in a cluster. Useful when you need to run a newer version of JBoss Cache in a cluster containing older versions, and can be used to perform rolling upgrades.
objectInputStreamPoolSize	objectInputStreamPoolSize	Positive integer	50	Not used at the moment.
objectOutputStreamPoolSize	objectOutputStreamPoolSize	Positive integer	50	Not used at the moment.

Table 12.19. The <eviction /> Element

The <eviction /> Element	
Description	This element controls how eviction works in the cache.
Parent	Table 12.1, "The **<jbosscache />** Element"
Children	Table 12.21, "The **<default />** Element", Table 12.23, "The **<region />** Element"
Bean equivalent	**EvictionConfig**

Table 12.20. <eviction /> Attributes

<eviction /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
wakeUpInterval	wakeupInterval	integer	5000	The frequency with which the eviction thread runs, in milliseconds. If set to less than 1, the eviction thread never runs and is effectively disabled.

Table 12.21. The <default /> Element

The <default /> Element	
Description	This element defines the default eviction region.
Parent	Table 12.19, "The **<eviction />** Element"
Children	Table 12.25, "The **<property />** Element"
Bean equivalent	**EvictionRegionConfig**

Table 12.22. <default /> Attributes

<default /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**

<default /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
algorithmClass	evictionAlgorithmConfig	A valid class that is available on the classpath	none	This attribute needs to be specified if this tag is being used. Note that if being configured programmatically, the eviction algorithm's corresponding **EvictionAlgorithmConfig** file should be used instead. E.g., where you would use **LRUAlgorithm** in XML, you would use an instance of **LRUAlgorithmConfig** programmatically.
actionPolicyClass	evictionActionPolicyClassName	A valid class that is available on the classpath	DefaultEvictionActionPolicy	The eviction action policy class, defining what happens when a node needs to be evicted.
eventQueueSize	eventQueueSize (*dynamic*	integer	200000	The size of the bounded eviction event queue.

Table 12.23. The <region /> Element

The <region /> Element	
Description	This element defines an eviction region. Multiple instances of this tag can exist provided they have unique **name** attributes.
Parent	Table 12.19, "The **<eviction />** Element"
Children	Table 12.25, "The **<property />** Element"

The `<region />` Element	
Bean equivalent	**`EvictionRegionConfig`**

Table 12.24. `<region />` Attributes

`<region />` Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
name	regionFqn	A String that could be parsed using Fqn.fromString()	none	This should be a unique name that defines this region. See the Section 10.2, "Eviction Regions" for details of eviction regions.
algorithmClass	evictionAlgorithmConfig	A valid class that is available on the classpath	none	This attribute needs to be specified if this tag is being used. Note that if being configured programmatically, the eviction algorithm's corresponding **`EvictionAlgorithmConfig`** file should be used instead. E.g., where you would use **`LRUAlgorithm`** in XML, you would use an instance of **`LRUAlgorithmConfig`** programmatically.
actionPolicyClass	evictionActionPolicyClassName	A valid class that is available on the classpath	DefaultEvictionActionPolicy	The eviction action policy class, defining what happens when a node needs to be evicted.

<region /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
eventQueueSize	eventQueueSize (*dynamic*	integer	200000	The size of the bounded eviction event queue.

Table 12.25. The <property /> Element

The <property /> Element	
Description	A mechanism of passing in name-value properties to the enclosing configuration element.
Parent	Table 12.21, "The **<default />** Element", Table 12.23, "The **<region />** Element", Table 12.38, "The **<interceptor />** Element"
Children	
Bean equivalent	Either direct setters or **setProperties()** enclosing bean

Table 12.26. <property /> Attributes

<property /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
name	Either direct setters or **setProperties()** enclosing bean	String	none	Property name
value	Either direct setters or **setProperties()** enclosing bean	String	none	Property value

Table 12.27. The <loaders /> Element

The <loaders /> Element	
Description	Defines any cache loaders.

The <loaders /> Element	
Parent	Table 12.1, "The <jbosscache /> Element"
Children	Table 12.29, "The <preload /> Element", Table 12.32, "The <loader /> Element"
Bean equivalent	CacheLoaderConfig

Table 12.28. <loaders /> Attributes

<loaders /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
passivation	passivation	true, false	false	If true, cache loaders are used in passivation mode. See the Chapter 9, *Cache Loaders* for a detailed discussion on this.
shared	shared	true, false	false	If true, cache loaders are used in shared mode. See the Chapter 9, *Cache Loaders* for a detailed discussion on this.

Table 12.29. The <preload /> Element

The <preload /> Element	
Description	Defines preloading of Fqn subtrees when a cache starts up. This element has no attributes.
Parent	Table 12.27, "The <loaders /> Element"
Children	Table 12.30, "The <node /> Element"
Bean equivalent	CacheLoaderConfig

Table 12.30. The <node /> Element

The <node /> Element	
Description	This element defines a subtree under which all content will be preloaded from the cache loaders when the cache starts. Multiple subtrees can be preloaded, although it only makes sense to define more than one subtree if they do not overlap.
Parent	Table 12.29, "The **<preload />** Element"
Children	
Bean equivalent	CacheLoaderConfig

Table 12.31. <node /> Attributes

<node /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
fqn	preload	String	none	An Fqn to preload. This should be a String that can be parsed with Fqn.fromString(). When doing this programmatically, you should create a single String containing all of the Fqns you wish to preload, separated by spaces, and pass that into **CacheLoaderConfig.setPreload()**.

Table 12.32. The <loader /> Element

The <loader /> Element	
Description	This element defines a cache loader. Multiple elements may be used to create cache loader chains.
Parent	Table 12.27, "The **<loaders />** Element"

The <loader /> Element	
Children	Table 12.34, "The <properties /> Element", Table 12.35, "The <singletonStore /> Element"
Bean equivalent	IndividualCacheLoaderConfig

Table 12.33. <loader /> Attributes

<loader /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
class	className	A valid class that is available on the classpath	none	A cache loader implementation to use.
async	async	true, false	false	All modifications to this cache loader happen asynchronously, on a separate thread.
fetchPersistentState	fetchPersistentState	true, false	false	When a cache starts up, retrieve persistent state from the cache loaders in other caches in the cluster. Only *one* loader element may set this to true. Also, only makes sense if the Table 12.40, "The <clustering /> Element" tag is present.
purgeOnStartup	purgeOnStartup	true, false	false	Purges this cache loader when it starts up.

Table 12.34. The <properties /> Element

The `<properties />` Element	
Description	This element contains a set of properties that can be read by a **`java.util.Properties`** instance. This tag has no attributes, and the contents of this tag will be parsed by **`Properties.load()`**.
Parent	Table 12.32, "The **`<loader />`** Element", Table 12.35, "The **`<singletonStore />`** Element", Table 12.52, "The **`<locator />`** Element"
Children	
Bean equivalent	IndividualCacheLoaderConfig.setProperties()

Table 12.35. The `<singletonStore />` Element

The `<singletonStore />` Element	
Description	This element configures the enclosing cache loader as a Section 9.2.1, "Singleton Store Configuration".
Parent	Table 12.32, "The **`<loader />`** Element"
Children	Table 12.34, "The **`<properties />`** Element"
Bean equivalent	SingletonStoreConfig

Table 12.36. `<singletonStore />` Attributes

`<singletonStore />` Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
class	className	A valid class that is available on the classpath	SingletonStoreCacheLoader	A singleton store wrapper implementation to use.
enabled	enabled	true, false	false	If true, the singleton store cache loader is enabled.

Table 12.37. The `<customInterceptors />` Element

The `<customInterceptors />` Element	
Description	This element allows you to define custom interceptors for the cache. This tag has no attributes.
Parent	Table 12.1, "The **`<jbosscache />`** Element"
Children	Table 12.38, "The **`<interceptor />`** Element"
Bean equivalent	None. At runtime, instantiate your own interceptor and pass it in to the cache using **`Cache.addInterceptor()`**.

Table 12.38. The `<interceptor />` Element

The `<interceptor />` Element	
Description	This element allows you configure a custom interceptor. This tag may appear multiple times.
Parent	Table 12.37, "The **`<customInterceptors />`** Element"
Children	Table 12.25, "The **`<property />`** Element"
Bean equivalent	None. At runtime, instantiate your own interceptor and pass it in to the cache using **`Cache.addInterceptor()`**.

Table 12.39. `<interceptor />` Attributes

`<interceptor />` Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
class	-	A valid class that is available on the classpath	none	An implementation of **`CommandInterceptor`**.

<interceptor /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
position	-	first, last		A position at which to place this interceptor in the chain. First is the first interceptor encountered when an invocation is made on the cache, last is the last interceptor before the call is passed on to the data structure. *Note that this attribute is mutually exclusive with* ***before****,* ***after*** *and* ***index****.*
before	-	Fully qualified class name of an interceptor		Will place the new interceptor directly before the instance of the named interceptor. *Note that this attribute is mutually exclusive with* ***position****,* ***after*** *and* ***index****.*
after	-	Fully qualified class name of an interceptor		Will place the new interceptor directly after the instance of the named interceptor. *Note that this attribute is mutually exclusive with* ***position****,* ***before*** *and* ***index****.*

<interceptor /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
index	-	Positive integers		A position at which to place this interceptor in the chain, with 0 being the first position. *Note that this attribute is mutually exclusive with* ***position****,* ***before*** *and* ***after****.*

Table 12.40. The <clustering /> Element

The <clustering /> Element	
Description	If this element is present, the cache is started in clustered mode. Attributes and child elements define clustering characteristics.
Parent	Table 12.1, "The **<jbosscache />** Element"
Children	Table 12.46, "The **<stateRetrieval />** Element", Table 12.42, "The **<sync />** Element", Table 12.44, "The **<async />** Element", Table 12.48, "The **<buddy />** Element", Table 12.54, "The **<jgroupsConfig />** Element"
Bean equivalent	Configuration

Table 12.41. <clustering /> Attributes

<clustering /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**

<clustering /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
mode	cacheMode	replication, invalidation, r, i	replication	See the Chapter 8, *Cache Modes and Clustering* for the differences between replication and invalidation. When using the bean, synchronous and asynchronous communication is combined with clustering mode to give you the enumberation **Configuration.CacheMode**.
clusterName	clusterName	String	JBossCache-cluster	A cluster name which is used to identify the cluster to join.

Table 12.42. The <sync /> Element

The <sync /> Element	
Description	If this element is present, all communications are synchronous, in that whenever a thread sends a message sent over the wire, it blocks until it receives an acknowledgement from the recipient. This element is mutually exclusive with the Table 12.44, "The **<async />** Element" element.
Parent	Table 12.40, "The **<clustering />** Element"
Children	
Bean equivalent	Configuration.setCacheMode()

Table 12.43. <sync /> Attributes

<sync /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
replTimeout	syncReplTimeout (*dynamic*)	positive integer	15000	This is the timeout used to wait for an acknowledgement when making a remote call, after which an exception is thrown.

Table 12.44. The <async /> Element

The <async /> Element	
Description	If this element is present, all communications are asynchronous, in that whenever a thread sends a message sent over the wire, it does not wait for an acknowledgement before returning. This element is mutually exclusive with the Table 12.42, "The **<sync />** Element" element.
Parent	Table 12.40, "The **<clustering />** Element"
Children	
Bean equivalent	Configuration.setCacheMode()

Table 12.45. <async /> Attributes

<async /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**

<async /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
serializationExecutorPoolSize	serializationExecutorPoolSize	positive integer	25	In addition to replication happening asynchronously, even serialization of contents for replication happens in a separate thread to allow the caller to return as quickly as possible. This setting controls the size of the serializer thread pool. Setting this to any value less than 1 means serialization does not happen asynchronously.
serializationExecutorQueueSize	serializationExecutorQueueSize	positive integer	50000	This is used to define the size of the bounded queue that holds tasks for the serialization executor. This is ignored if a serialization executor is not used, such as when **`serializationExecutorPoolSize`** is less than 1.
useReplQueue	useReplQueue	true, false	false	If true, this forces all async communications to be queued up and sent out periodically as a batch.

<async /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
replQueueInterval	replQueueInterval	positive integer	5000	If **useReplQueue** is set to true, this attribute controls how often the asynchronous thread used to flush the replication queue runs. This should be a positive integer which represents thread wakeup time in milliseconds.
replQueueMaxElements	replQueueMaxElements	positive integer	1000	If **useReplQueue** is set to true, this attribute can be used to trigger flushing of the queue when it reaches a specific threshold.

Table 12.46. The <stateRetrieval /> Element

The <stateRetrieval /> Element	
Description	This tag controls ho state is retrieved from neighboring caches when this cache instance starts.
Parent	Table 12.40, "The **<clustering />** Element"
Children	
Bean equivalent	Configuration

Table 12.47. <stateRetrieval /> Attributes

<stateRetrieval /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
fetchInMemoryState	fetchInMemoryState	true, false	true	If true, this will cause the cache to ask neighboring caches for state when it starts up, so the cache starts "warm".
timeout	stateRetrievalTimeout	positive integer	10000	This is the maximum amount of time - in milliseconds - to wait for state from neighboring caches, before throwing an exception and aborting startup.
nonBlocking	useNonBlockingStateTransfer	true, false	false	This configuration switch enables the Non-Blocking State Transfer mechanism, new in 3.1.0. Note that this requires MVCC as a node locking scheme, and that STREAMING_STATE_TRANSFER is present in the JGroups stack used.

Table 12.48. The <buddy /> Element

The <buddy /> Element	
Description	If this tag is present, then state is not replicated across the entire cluster. Instead, buddy replication is used to select cache instances to maintain backups on. See Section 8.1.2.2, “Buddy Replication” for details. Note that this is only used if the clustering mode is **`replication`**, and not if it is **`invalidation`**.

The <buddy /> Element	
Parent	Table 12.40, "The <clustering /> Element"
Children	Table 12.50, "The <dataGravitation /> Element", Table 12.52, "The <locator /> Element",
Bean equivalent	BuddyReplicationConfig

Table 12.49. <buddy /> Attributes

<buddy /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
enabled	enabled	true, false	false	If true, buddy replication is enabled.
communicationTimeout	buddyCommunicationTimeout	positive integer	10000	This is the maximum amount of time - in milliseconds - to wait for buddy group organization communications from buddy caches.
poolName	buddyPoolName	String		This is used as a means to identify cache instances and provide hints to the buddy selection algorithms. More information on Section 8.1.2.2, "Buddy Replication".

Table 12.50. The <dataGravitation /> Element

The <dataGravitation /> Element	
Description	This tag configures how data gravitation is conducted. See Section 8.1.2.2, "Buddy Replication" for details.

The <dataGravitation /> Element	
Parent	Table 12.48, "The **<buddy />** Element"
Children	
Bean equivalent	BuddyReplicationConfig

Table 12.51. <dataGravitation /> Attributes

<dataGravitation /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
auto	autoDataGravitation	true, false	true	If true, when a get() is performed on a cache and nothing is found, a gravitation from neighboring caches is attempted. If this is false, then gravitations can only occur if the **Option.setForceDataGravitation()** option is provided.
removeOnFind	dataGravitationRemoveOnFind	true, false	true	If true, when gravitation occurs, the instance that requests the gravitation takes ownership of the state and requests that all other instances remove the gravitated state from memory.

<dataGravitation /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
searchBackupTrees	dataGravitationSearchBackupTrees	true, false	true	If true, incoming gravitation requests will cause the cache to search not just its primary data structure but its backup structure as well.

Table 12.52. The <locator /> Element

The <locator /> Element	
Description	This tag provides a pluggable mechanism for providing buddy location algorithms.
Parent	Table 12.48, "The **<buddy />** Element"
Children	Table 12.34, "The **<properties />** Element"
Bean equivalent	BuddyLocatorConfig

Table 12.53. <locator /> Attributes

<locator /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
class	className	A valid class that is available on the classpath	NextMemberBuddyLocator	A **BuddyLocator** implementation to use when selecting buddies from the cluster. Please refer to **BuddyLocator** javadocs for details.

Table 12.54. The <jgroupsConfig /> Element

The <jgroupsConfig /> Element	
Description	This tag provides a configuration which is used with JGroups to create a network communication channel.
Parent	Table 12.40, "The <clustering /> Element"
Children	A series of elements representing JGroups protocols (see JGroups documentation). Note that there are no child elements if any of the element attributes are used instead. See section on attributes.
Bean equivalent	Configuration

Table 12.55. <jgroupsConfig /> Attributes

<jgroupsConfig /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
configFile	clusterConfig	A JGroups configuration file on the classpath	udp.xml	If this attribute is used, then any JGroups elements representing protocols within this tag are ignored. Instead, JGroups settings are read from the file specified. Note that this cannot be used with the **multiplexerStack** attribute.

<jgroupsConfig /> Attributes				
Attribute	**Bean Field**	**Allowed**	**Default**	**Description**
multiplexerStack	muxStackName	A valid multiplexer stack name that exists in the channel factory passed in to the **RuntimeConfig**		This can only be used with the **RuntimeConfig**, where you pass in a JGroups **ChannelFactory** instance using **RuntimeConfig.setMuxChannelFactory()**. If this attribute is used, then any JGroups elements representing protocols within this tag are ignored. Instead, the JGroups channel is created using the factory passed in. Note that this cannot be used with the **configFile** attribute.

CHAPTER 13. JMX REFERENCES

13.1. JBOSS CACHE STATISTICS

There is a whole wealth of information being gathered and exposed on to JMX for monitoring the cache. Some of these are detailed below:

Table 13.1. JBoss Cache JMX MBeans

MBean	Attribute/Operation Name	Description
DataContainerImpl	getNumberOfAttributes()	Returns the number of attributes in all nodes in the data container
	getNumberOfNodes()	Returns the number of nodes in the data container
	printDetails()	Prints details of the data container
RPCManagerImpl	localAddressString	String representation of the local address
	membersString	String representation of the cluster view
	statisticsEnabled	Whether RPC statistics are being gathered
	replicationCount	Number of successful replications
	replicationFailures	Number of failed replications
	successRatio	RPC call success ratio
RegionManagerImpl	dumpRegions()	Dumps a String representation of all registered regions, including eviction regions depicting their event queue sizes
	numRegions	Number of registered regions
BuddyManager	buddyGroup	A String representation of the cache's buddy group
	buddyGroupsIParticipateIn	String representations of all buddy groups the cache participates in
TransactionTable	numberOfRegisteredTransactions	The number of registered, ongoing transactions

MBean	Attribute/Operation Name	Description
	transactionMap	A String representation of all currently registered transactions mapped to internal GlobalTransaction instances
MVCCLockManager	concurrencyLevel	The configured concurrency level
	numberOfLocksAvailable	Number of locks in the shared lock pool that are not used
	numberOfLocksHeld	Number of locks in the shared lock pool that are in use
	testHashing(String fqn)	Tests the spreading of locks across Fqns. For a given (String based) Fqn, this method returns the index in the lock array that it maps to.
ActivationInterceptor	Activations	Number of passivated nodes that have been activated.
CacheLoaderInterceptor	CacheLoaderLoads	Number of nodes loaded through a cache loader.
	CacheLoaderMisses	Number of unsuccessful attempts to load a node through a cache loader.
CacheMgmtInterceptor	Hits	Number of successful attribute retrievals.
	Misses	Number of unsuccessful attribute retrievals.
	Stores	Number of attribute store operations.
	Evictions	Number of node evictions.
	NumberOfAttributes	Number of attributes currently cached.
	NumberOfNodes	Number of nodes currently cached.
	ElapsedTime	Number of seconds that the cache has been running.

MBean	Attribute/Operation Name	Description
	TimeSinceReset	Number of seconds since the cache statistics have been reset.
	AverageReadTime	Average time in milliseconds to retrieve a cache attribute, including unsuccessful attribute retrievals.
	AverageWriteTime	Average time in milliseconds to write a cache attribute.
	HitMissRatio	Ratio of hits to hits and misses. A hit is a get attribute operation that results in an object being returned to the client. The retrieval may be from a cache loader if the entry isn't in the local cache.
	ReadWriteRatio	Ratio of read operations to write operations. This is the ratio of cache hits and misses to cache stores.
CacheStoreInterceptor	CacheLoaderStores	Number of nodes written to the cache loader.
InvalidationInterceptor	Invalidations	Number of cached nodes that have been invalidated.
PassivationInterceptor	Passivations	Number of cached nodes that have been passivated.
TxInterceptor	Prepares	Number of transaction prepare operations performed by this interceptor.
	Commits	Number of transaction commit operations performed by this interceptor.
	Rollbacks	Number of transaction rollbacks operations performed by this interceptor.

MBean	Attribute/Operation Name	Description
	numberOfSyncsRegistered	Number of synchronizations registered with the transaction manager pending completion and removal.

13.2. JMX MBEAN NOTIFICATIONS

The following table depicts the JMX notifications available for JBoss Cache as well as the cache events to which they correspond. These are the notifications that can be received through the **`CacheJmxWrapper`** MBean. Each notification represents a single event published by JBoss Cache and provides user data corresponding to the parameters of the event.

Table 13.2. JBoss Cache MBean Notifications

Notification Type	Notification Data	CacheListener Event
org.jboss.cache.CacheStarted	String: cache service name	@CacheStarted
org.jboss.cache.CacheStopped	String: cache service name	@CacheStopped
org.jboss.cache.NodeCreated	String: fqn, boolean: isPre, boolean: isOriginLocal	@NodeCreated
org.jboss.cache.NodeEvicted	String: fqn, boolean: isPre, boolean: isOriginLocal	@NodeEvicted
org.jboss.cache.NodeLoaded	String: fqn, boolean: isPre	@NodeLoaded
org.jboss.cache.NodeModifed	String: fqn, boolean: isPre, boolean: isOriginLocal	@NodeModifed
org.jboss.cache.NodeRemoved	String: fqn, boolean: isPre, boolean: isOriginLocal	@NodeRemoved
org.jboss.cache.NodeVisited	String: fqn, boolean: isPre	@NodeVisited
org.jboss.cache.ViewChanged	String: view	@ViewChanged
org.jboss.cache.NodeActivated	String: fqn	@NodeActivated
org.jboss.cache.NodeMoved	String: fromFqn, String: toFqn, boolean: isPre	@NodeMoved
org.jboss.cache.NodePassivated	String: fqn	@NodePassivated