Ehcache Guide & Reference

Version 1.7.1

Contents

Chapter 1

Preface

This is a book about Ehcache, a widely used open source Java cache. Ehcache has grown in size and scope since it was introduced in October 2003. As people used it they often noticed it was missing a feature they wanted. Over time, the features that were repeatedly asked for, and make sense for a Cache, have been added.

Ehcache is now used for Hibernate caching, data access object caching, security credential caching, web caching, SOAP and RESTful server caching, application persistence and distributed caching.

In August 2009, Ehcache was acquired by Terracotta, Inc.

1.1 Version

This book is for Ehcache version 1.7.1.

1.2 Audience

The intended audience for this book is developers who use ehcache. It should be able to be used to start from scratch, get up and running quickly, and also be useful for the more complex options.

Ehcache is about performance and load reduction of underlying resources. Another natural audience is performance specialists.

It is also intended for application and enterprise architects. Some of the features of ehcache, such as distributed caching and Java EE caching, are alternatives to be considered along with other ways of solving those problems. This book discusses the trade-offs in Ehcache's approach to help make a decision about appropriateness of use.

1.3 Acknowledgements

Ehcache has had many contributions in the form of forum discussions, feature requests, bug reports, patches and code commits.

Rather than try and list the many hundreds of people who have contributed to Ehcache in some way it is better to link to the web site where contributions are acknowledged in the following ways:

- Bug reports and features requests appear in the changes report here:

- Patch contributors generally end up with an author tag in the source they contributed to
- Team members appear on the team list page here:

  Thanks to Denis Orlov for suggesting the need for a book in the first place.

1.4 About the Ehcache name and logo

Adam Murdoch (an all round top Java coder) came up with the name in a moment of inspiration while we were stuck on the SourceForge project create page. Ehcache is a palindrome. He thought the name was wicked cool and we agreed.

The logo is similarly symmetrical, and is evocative of the diagram symbol for a doubly-linked list. That structure lies at the heart of ehcache.

Chapter 2

Introduction

Ehcache is a cache library. Before getting into ehcache, it is worth stepping back and thinking about caching generally.

2.1 About Caches

Wiktionary defines a cache as *A store of things that will be required in future, and can be retrieved rapidly*. That is the nub of it.

In computer science terms, a cache is a collection of temporary data which either duplicates data located elsewhere or is the result of a computation. Once in the cache, the data can be repeatedly accessed inexpensively.

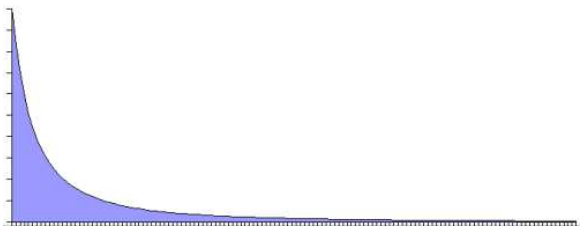
2.2 Why caching works

2.2.1 Locality of Reference

While Ehcache concerns itself with Java objects, caching is used throughout computing, from CPU caches to the DNS system. Why? Because many computer systems exhibit *locality of reference*. Data that is near other data or has just been used is more likely to be used again.

2.2.2 The Long Tail

Chris Anderson, of Wired Magazine, coined the term *The Long Tail* to refer to Ecommerce systems. The idea that a small number of items may make up the bulk of sales, a small number of blogs might get the most hits and so on. While there is a small list of popular items, there is a long tail of less popular ones.



The Long Tail

The Long Tail is itself a vernacular term for a Power Law probability distribution. They don't just appear in ecommerce, but throughout nature. One form of a Power Law distribution is the Pareto distribution, commonly know as the 80:20 rule.

This phenomenon is useful for caching. If 20% of objects are used 80% of the time and a way can be found to reduce the cost of obtaining that 20%, then the system performance will improve.

2.3 Will an Application Benefit from Caching?

The short answer is that it often does, due to the effects noted above.

The medium answer is that it often depends on whether it is CPU bound or I/O bound. If an application is I/O bound then then the time taken to complete a computation depends principally on the rate at which data can be obtained. If it is CPU bound, then the time taken principally depends on the speed of the CPU and main memory.

While the focus for caching is on improving performance, it it also worth realizing that it reduces load. The time it takes something to complete is usually related to the expense of it. So, caching often reduces load on scarce resources.

2.3.1 Speeding up CPU bound Applications

CPU bound applications are often sped up by:

- improving algorithm performance
- parallelizing the computations across multiple CPUs (SMP) or multiple machines (Clusters).
- upgrading the CPU speed.

  The role of caching, if there is one, is to temporarily store computations that may be reused again.

  An example from Ehcache would be large web pages that have a high rendering cost. Another caching of authentication status, where authentication requires cryptographic transforms.

2.3.2 Speeding up I/O bound Applications

Many applications are I/O bound, either by disk or network operations. In the case of databases they can be limited by both.

There is no Moore's law for hard disks. A 10,000 RPM disk was fast 10 years ago and is still fast. Hard disks are speeding up by using their own caching of blocks into memory.

Network operations can be bound by a number of factors:

- time to set up and tear down connections
- latency, or the minimum round trip time
- throughput limits
- marshalling and unmarhshalling overhead

  The caching of data can often help a lot with I/O bound applications. Some examples of Ehcache uses are:

- Data Access Object caching for Hibernate
- Web page caching, for pages generated from databases.

2.3.3 Increased Application Scalability

The flip side of increased performance is increased scalability. Say you have a database which can do 100 expensive queries per second. After that it backs up and if connections are added to it it slowly dies.

In this case, caching may be able to reduce the workload required. If caching can cause 90 of that 100 to be cache hits and not even get to the database, then the database can scale 10 times higher than otherwise.

2.4 How much will an application speed up with Caching?

2.4.1 The short answer

The short answer is that it depends on a multitude of factors being:

- how many times a cached piece of data can and is reused by the application
- the proportion of the response time that is alleviated by caching

  In applications that are I/O bound, which is most business applications, most of the response time is getting data from a database. Therefore the speed up mostly depends on how much reuse a piece of data gets.

  In a system where each piece of data is used just once, it is zero. In a system where data is reused a lot, the speed up is large.

The long answer, unfortunately, is complicated and mathematical. It is considered next.

2.4.2 Applying Amdahl's Law

Amdahl's law, after Gene Amdahl, is used to find the system speed up from a speed up in part of the system.

```
1 / ((1 - Proportion Sped Up) + Proportion Sped Up / Speed up)
```

The following examples show how to apply Amdahl's law to common situations. In the interests of simplicity, we assume:

- a single server
- a system with a single thing in it, which when cached, gets 100% cache hits and lives forever.

Persistent Object Relational Caching

A Hibernate Session.load() for a single object is about 1000 times faster from cache than from a database.

A typical Hibernate query will return a list of IDs from the database, and then attempt to load each. If Session.iterate() is used Hibernate goes back to the database to load each object.

Imagine a scenario where we execute a query against the database which returns a hundred IDs and then load each one.

The query takes 20% of the time and the roundtrip loading takes the rest (80%). The database query itself is 75% of the time that the operation takes. The proportion being sped up is thus 60% (75% * 80%).

The expected system speedup is thus:

```
1 / ((1 - .6) + .6 / 1000)

= 1 / (.4 + .006)

= 2.5 times system speedup
```

Web Page Caching

An observed speed up from caching a web page is 1000 times. Ehcache can retrieve a page from its SimplePageCachingFilter in a few ms.

Because the web page is the end result of a computation, it has a proportion of 100%.

The expected system speedup is thus:

```
1 / ((1 - 1) + 1 / 1000)

= 1 / (0 + .001)

= 1000 times system speedup
```

Web Page Fragment Caching

Caching the entire page is a big win. Sometimes the liveness requirements vary in different parts of the page. Here the SimplePageFragmentCachingFilter can be used.

Let's say we have a 1000 fold improvement on a page fragment that taking 40% of the page render time.

The expected system speedup is thus:

```
1 / ((1 - .4) + .4 / 1000)

= 1 / (6 + .004)

= 1.6 times system speedup
```

2.4.3 Cache Efficiency

In real life cache entrie do not live forever. Some examples that come close are "static" web pages or fragments of same, like page footers, and in the database realm, reference data, such as the currencies in the world.

Factors which affect the efficiency of a cache are:

liveness how live the data needs to be. The less live the more it can be cached

proportion of data cached what proportion of the data can fit into the resource limits of the machine. For 32 bit Java systems, there was a hard limit of 2GB of address space. While now relaxed, garbage collection issues make it harder to go a lot large. Various eviction algorithms are used to evict excess entries.

Shape of the usage distribution If only 300 out of 3000 entries can be cached, but the Pareto distribution applies, it may be that 80% of the time, those 300 will be the ones requested. This drives up the average request lifespan.

Read/Write ratio The proportion of times data is read compared with how often it is written. Things such as the number of rooms left in a hotel will be written to quite a lot. However the details of a room

sold are immutable once created so have a maximum write of 1 with a potentially large number of reads.

Ehcache keeps these statistics for each Cache and each element, so they can be measured directly rather than estimated.

2.4.4 Cluster Efficiency

Also in real life, we generally do not find a single server?

Assume a round robin load balancer where each hit goes to the next server.

The cache has one entry which has a variable lifespan of requests, say caused by a time to live. The following table shows how that lifespan can affect hits and misses.

```
 Server 1    Server 2   Server 3    Server 4

  M             M          M           M
  H             H          H           H
  H             H          H           H
  H             H          H           H
  H             H          H           H
  ...           ...        ...         ...
```

The cache hit ratios for the system as a whole are as follows:

```
Entry
Lifespan  Hit Ratio   Hit Ratio  Hit Ratio   Hit Ratio
in Hits   1 Server    2 Servers  3 Servers   4 Servers

2          1/2           0/2         0/2         0/2
4          3/4           2/4         1/4         0/4
10         9/10          8/10        7/10        6/10
20         19/20         18/20       17/20       16/10
50         49/50         48/50       47/20       46/50
```

The efficiency of a cluster of standalone caches is generally:

```
(Lifespan in requests - Number of Standalone Caches) / Lifespan in requests
```

Where the lifespan is large relative to the number of standalone caches, cache efficiency is not much affected.

However when the lifespan is short, cache efficiency is dramatically affected.

(To solve this problem, Ehcache supports distributed caching, where an entry put in a local cache is also propagated to other servers in the cluster.)

2.4.5 A cache version of Amdahl's law

From the above we now have:

```
1 / ((1 - Proportion Sped Up * effective cache efficiency) +
(Proportion Sped Up  * effective cache efficiency)/ Speed up)
```

effective cache efficiency = cache efficiency * cluster efficiency

2.4.6 Web Page example

Applying this to the earlier web page cache example where we have cache efficiency of 35% and average request lifespan of 10 requests and two servers:

```
cache efficiency = .35

cluster efficiency = .(10 - 1) / 10
                   = .9

effective cache efficiency = .35 * .9
                           = .315

      1 / ((1 - 1 * .315) + 1 * .315 / 1000)

      = 1 / (.685 + .000315)

      = 1.45 times system speedup
```

What if, instead the cache efficiency is 70%; a doubling of efficiency. We keep to two servers.

```
cache efficiency = .70

cluster efficiency = .(10 - 1) / 10
                   = .9

effective cache efficiency = .70 * .9
                           = .63

      1 / ((1 - 1 * .63) + 1 * .63 / 1000)

      = 1 / (.37 + .00063)

      = 2.69 times system speedup
```

What if, instead the cache efficiency is 90%; a doubling of efficiency. We keep to two servers.

```
cache efficiency = .90

cluster efficiency = .(10 - 1) / 10
                   = .9

effective cache efficiency = .9 * .9
                           = .81

      1 / ((1 - 1 * .81) + 1 * .81 / 1000)

      = 1 / (.19 + .00081)

      = 5.24 times system speedup
```

Why is the reduction so dramatic? Because Amdahl's law is most sensitive to the proportion of the system that is sped up.

Chapter 3

Getting Started

Ehcache can be used directly. It can also be used with the popular Hibernate Object/Relational tool. Finally, it can be used for Java EE Servlet Caching.

This quick guide gets you started on each of these. The rest of the documentation can be explored for a deeper understanding.

3.1 General Purpose Caching

- Make sure you are using a supported Java version.
- Place the Ehcache jar into your classpath.
- Ensure that any libraries required to satisfy dependencies are also in the classpath.
- Configure ehcache.xml and place it in your classpath.
- Optionally, configure an appropriate logging level.

  See the Code Samples chapter for more information on direct interaction with ehcache.

3.2 Hibernate

- Perform the same steps as for General Purpose Caching.
- Create caches in ehcache.xml.

  See the Hibernate Caching chapter for more information.

3.3 Java EE Servlet Caching

- Perform the same steps as for General Purpose Caching.
- Configure a cache for your web page in ehcache.xml.
- To cache an entire web page, either use SimplePageCachingFilter or create your own subclass of CachingFilter
- To cache a jsp:Include or anything callable from a RequestDispatcher, either use SimplePageFragmentCachingFilter or create a subclass of PageFragmentCachingFilter.

- Configure the web.xml. Declare the filters created above and create filter mapping associating the filter with a URL.

  See the Web Caching chapter for more information.

3.4 RESTful and SOAP Caching with the Cache Server

- Download the standalone cache server from http://sourceforge.net/project/showfiles.php?group_id=93232
- cd to the bin directory
- Type `startup.sh` to start the server with the log in the foreground.

  By default it will listen on port 8080, will have both RESTful and SOAP web services enabled, and will use a sample Ehcache configuration from the WAR module.
- See the code samples in the Cache Server chapter. You can use Java or any other programming language to the use the Cache Server.

  See the Cache Server chapter for more information.

3.5 JCache style caching

Ehcache contains an early draft implementation of JCache contained in the net.sf.ehcache.jcache package.

See the JSR107 chapter for usage.

3.6 Spring, Cocoon, Acegi and other frameworks

Usually, with these, you are using Ehcache without even realising it. The first steps in getting more control over what is happening are:

- discover the cache names used by the framework
- create your own ehcache.xml with settings for the caches and place it in the application classpath.

Chapter 4

Features

- Fast and Light Weight
  - Fast
  - Simple
  - Small foot print
  - Minimal dependencies
- Scalable
  - Provides Memory and Disk stores for scalabilty into gigabytes
  - Scalable to hundreds of caches
  - Tuned for high concurrent load on large multi-cpu servers
  - Multiple CacheManagers per virtual machine
- Flexible
  - Supports Object or Serializable caching
  - Support cache-wide or Element-based expiry policies
  - Provides LRU, LFU and FIFO cache eviction policies
  - Provides Memory and Disk stores
  - Distributed Caching
- Standards Based
  - Full implementation of JSR107 JCACHE API
- Extensible
  - Listeners may be plugged in
  - Peer Discovery, Replicators and Listeners may be plugged in
  - Cache Extensions may be plugged in
  - Cache Loaders may be plugged in
  - Cache Exception Handlers may be plugged in
- Application Persistence

  - Persistent disk store which stores data between VM restarts
  - Flush to disk on demand
- Supports Listeners
  - CacheManager listeners
  - Cache event listeners
- JMX Enabled
- Distributed
  - Support for replication via RMI, JGroups, JMS or Terracotta
  - Peer Discovery
  - Reliable Delivery
  - Synchronous Or Asynchronous Replication
  - Copy Or Invalidate Replication
  - Transparent Replication
  - Extensible
  - Bootstrapping from Peers
- Cache Server
  - #RESTful cache server
  - #SOAP cache server
  - #comes as a WAR or as a complete server
- Java EE and Applied Caching
  - Blocking Cache to avoid duplicate processing for concurrent operations
  - SelfPopulating Cache for pull through caching of expensive operations
  - Java EE Gzipping Servlet Filter
  - Cacheable Commands
  - Works with Hibernate
  - Works with Google App Engine
- High Quality
  - High Test Coverage
  - Automated Load, Limit and Performance System Tests
  - Production tested
  - Fully documented
  - Trusted by Popular Frameworks
  - Conservative Commit policy
  - Full public information on the history of every bug
  - Responsiveness to serious bugs
- Open Source Licensing
  - Apache 2.0 license

4.1 Fast and Light Weight

4.1.1 Fast

Over the years, various performance tests have shown Ehcache to be one of the fastest Java caches. Ehcache's threading is designed for large, high concurrency systems.

Extensive performance tests in the test suite keep ehcache's performance consistent between releases.

As an example, some guys have created a java cache test tool called cache4j_perfomance_tester.

The results for ehcache-1.1 and ehcache-1.2 follow.

```
ehcache-1.1

[java] ------------------------------------------------------------
[java] java.version=1.4.2_09
[java] java.vm.name=Java HotSpot(TM) Client VM
[java] java.vm.version=1.4.2-54
[java] java.vm.info=mixed mode
[java] java.vm.vendor="Apple Computer, Inc."
[java] os.name=Mac OS X
[java] os.version=10.4.5
[java] os.arch=ppc
[java] ------------------------------------------------------------
[java] This test can take about 5-10 minutes. Please wait ...
[java] ------------------------------------------------------------
[java] |GetPutRemoveT |GetPutRemove |Get |
[java] ------------------------------------------------------------
[java] cache4j 0.4 |9240 |9116 |5556 |
[java] oscache 2.2 |33577 |30803 |8350 |
[java] Ehcache 1.1 |7697 |6145 |3395 |
[java] jcs 1.2.7.0 |8966 |9455 |4072 |
[java] ------------------------------------------------------------

ehcache-1.2
[java] ------------------------------------------------------------
[java] java.version=1.4.2_09
[java] java.vm.name=Java HotSpot(TM) Client VM
[java] java.vm.version=1.4.2-54
[java] java.vm.info=mixed mode
[java] java.vm.vendor="Apple Computer, Inc."
[java] os.name=Mac OS X
[java] os.version=10.4.5
[java] os.arch=ppc
[java] ------------------------------------------------------------
[java] This test can take about 5-10 minutes. Please wait ...
[java] ------------------------------------------------------------
[java] |GetPutRemoveT |GetPutRemove |Get |
[java] ------------------------------------------------------------
[java] cache4j 0.4 |9410 |9053 |5865 |
[java] oscache 2.2 |28076 |30833 |8031 |
[java] Ehcache 1.2 |8753 |7072 |3479 |
[java] jcs 1.2.7.0 |8806 |9522 |4097 |
[java] ------------------------------------------------------------
```

4.1.2 Simple

Many users of Ehcache hardly know they are using it. Sensible defaults require no initial configuration.

The API is very simple and easy to use, making it possible to get up and running in minutes. See the Code Samples for details.

4.1.3 Small foot print

Ehcache 1.2 is 110KB making it convenient to package.

4.1.4 Minimal dependencies

The only dependency for core use is the JCACHE API.

4.2 Scalable

4.2.1 Provides Memory and Disk stores for scalabilty into gigabytes

The largest Ehcache installations use memory and disk stores in the gigabyte range. Ehcache is tuned for these large sizes.

4.2.2 Scalable to hundreds of caches

The largest Ehcache installations use hundreds of caches.

4.2.3 Tuned for high concurrent load on large multi-cpu servers

There is a tension between thread safety and performance. Ehcache's threading started off coarse-grained, but has increasingly made use of ideas from Doug Lea to achieve greater performance. Over the years there have been a number of scalability bottlenecks identified and fixed.

4.2.4 Multiple CacheManagers per virtual machine

Ehcache 1.2 introduced multiple CacheManagers per virtual machine. This enables completely difference ehcache.xml configurations to be applied.

4.3 Flexible

4.3.1 Supports Object or Serializable caching

As of ehcache-1.2 there is an API for Objects in addition to the one for Serializable. Non-serializable Objects can use all parts of Ehcache except for DiskStore and replication. If an attempt is made to persist or replicate them they are discarded and a WARNING level log message emitted.

The APIs are identical except for the return methods from Element. Two new methods on Element: getObjectValue and getKeyValue are the only API differences between the Serializable and Object APIs. This makes it very easy to start with caching Objects and then change your Objects to Seralizable to participate in the extra features when needed. Also a large number of Java classes are simply not Serializable.

4.3.2 Support cache-wide or Element-based expiry policies

Time to lives and time to idles are settable per cache. In addition, from ehcache-1.2.1, overrides to these can be set per Element.

4.3.3 Provides LRU, LFU and FIFO cache eviction policies

Ehcache 1.2 introduced Less Frequently Used and First In First Out caching eviction policies. These round out the eviction policies.

4.3.4 Provides Memory and Disk stores

Ehcache, like most of the cache solutions, provides high performance memory and disk stores.

4.3.5 Distributed

Flexible, extensible, high performance distributed caching. The default implementation supports cache discovery via multicast or manual configuration. Updates are delivered either asynchronously or synchronously via custom RMI connections. Additional discovery or delivery schemes can be plugged in by third parties.

See the Distributed Caching documentation for more feature details.

4.4 Standards Based

4.4.1 Full implementation of JSR107 JCACHE API

Ehcache offers the the most complete implementation of the JSR107 JCACHE to date.

Because JCACHE has not yet been released the JCACHE API that Ehcache implements has been released as net.sf.jsr107cache.

Implementers can code to the JCACHE API which will create portability to other caching solutions in the future.

The maintainer of ehcache, Greg Luck, is on the expert committee for JSR107.

4.5 Extensible

4.5.1 Listeners may be plugged in

Ehcache 1.2 provides `CacheManagerEventListener` and `CacheEventListener` interfaces. Implementations can be plugged in and configured in ehcache.xml.

4.5.2 Peer Discovery, Replicators and Listeners may be plugged in

Distributed caching, introduced in Ehcache 1.2 involves many choices and tradeoffs. The Ehcache team believe that one size will not fit all. Implementers can use built-in mechanisms or write their own. A plugin development guide is included for this purpose.

4.5.3 Cache Extensions may be plugged in

Create your own Cache Extensions, which hold a reference to a cache and are bound to its lifecycle.

4.5.4 Cache Loaders may be plugged in

Create your own Cache Loaders, which are general purpose asynchronous methods for loading data into caches, or use them in pull-through configuration.

4.5.5 Cache Exception Handlers may be plugged in

Create an Exception Handler which is invoked if any Exception occurs on a cache operation.

4.6 Application Persistence

4.6.1 Persistent disk store which stores data between VM restarts

With Ehcache 1.1 in 2004, Ehcache was the first open source Java cache to introduce persistent storage of cache data on disk on shutdown. The cached data is then accessible the next time the application runs.

4.6.2 Flush to disk on demand

With Ehcache 1.2, the flushing of entries to disk can be executed with a `cache.flush()` method whenever required, making it easier to use ehcache

4.7 Listeners

4.7.1 CacheManager listeners

Register Cache Manager listeners through the `CacheManagerEventListener` interface with the following event methods:

- `notifyCacheAdded()`
- `notifyCacheRemoved()`

4.7.2 Cache event listeners

Register Cache Event Listeners through the `CacheEventListener` interfaces, which provides a lot of flexibility for post-processing of cache events. The methods are:

- `notifyElementRemoved`
- `notifyElementPut`
- `notifyElementUpdated`
- `notifyElementExpired`

4.8 JMX Enabled

Ehcache is JMX enabled. You can monitor and manage the following MBeans:

- CacheManager
- Cache
- CacheConfiguration
- CacheStatistics

See the net.sf.ehcache.management package.

See http://weblogs.java.net/blog/maxpoon/archive/2007/06/extending_the_n_2.html for an online tutorial.

4.9 Distributed Caching

Ehcache 1.2 introduced a full-featured, fine-grained distributed caching mechanism for clusters, supporting multiple replication mechanisms through plugins.

4.9.1 Support for replication via RMI or JGroups

Ehcache 1.6 supports replication via RMI, JGroups, JMS or Terracotta.

4.9.2 Peer Discovery

Peer discovery may be either manually configured or automatic, using multicast. Multicast is simple, and adds and removes peers automatically. Manual configuration gives fine control and is useful for situations where multicast is blocked.

4.9.3 Reliable Delivery

The built-in delivery mechanism uses RMI with custom sockets over TCP, not UDP.

4.9.4 Synchronous Or Asynchronous Replication

Replication can be set to synchronous Or asynchronous, per cache.

4.9.5 Copy Or Invalidate Replication

Replication can be set to copy or invalidate, per cache, as is appropriate.

4.9.6 Transparent Replication

No programming changes are required to make use of replication. Only configuration in ehcache.xml.

4.9.7 Extensible

Distributed caching, introduced in Ehcache 1.2 involves many choices and tradeoffs. The Ehcache team believe that one size will not fit all. Implementers can use built-in mechanisms or write their own. A plugin development guide is included for this purpose.

4.9.8 Bootstrapping from Peers

Distributed caches enter and leave the cluster at different times. Caches can be configured to bootstrap themselves from the cluster when they are first initialized.

An abstract factory, BootstrapCacheLoaderFactory has been defined along with an interface BootstrapCacheLoader along with an RMI based default implementation.

4.10 Cache Server

Ehcache now comes with a Cache Server, available as a WAR for most web containers, or as a standalone server. The Cache Server has two apis: RESTful resource oriented, and SOAP. Both support clients in any programming language.

4.10.1 RESTful cache server

The Ehcache implementation strictly follows the RESTful resource-oriented architecture style.

Specifically:

- The HTTP methods GET, HEAD, PUT/POST and DELETE are used to specify the method of the operation. The URI does not contain method information.
- The scoping information, used to identify the resource to perform the method on, is contained in the URI path.
- The RESTful Web Service is described by and exposes a WADL (Web Application Description Language) file. It contains the URIs you can call, and what data to pass and get back. Use the OPTIONS method to return the WADL.

  For performance, HTTP/1.1 caching features are fully supported such as Last-Modified, ETag and so on. Ehcache responsds correctly to HEAD and conditional GET requests.

4.10.2 SOAP cache server

The Ehcache RESTFul Web Services API exposes the singleton CacheManager, which typically has been configured in ehcache.xml or an IoC container. Multiple CacheManagers are not supported.

The API definition is as follows:

- WSDL - EhcacheWebServiceEndpointService.wsdl
- Types - EhcacheWebServiceEndpointService_schema1.xsd

4.10.3 comes as a WAR or as a complete server

The standalone server comes with its own embedded Glassfish web container.

It also comes packaged as a WAR for deployment to any Servlet 2.5 web container. Glassfish V2/3, Tomcat 6 and Jetty 6 have been tested.

4.11 Java EE and Applied Caching

High quality implementations for common caching scenarios and patterns.

4.11.1 Blocking Cache to avoid duplicate processing for concurrent operations

A cache which blocks subsequent threads until the first read thread populates a cache entry.

4.11.2 SelfPopulating Cache for pull through caching of expensive operations

SelfPopulatingCache - a read-through cache. A cache that populates elements as they are requested without requiring the caller to know how the entries are populated. It also enables refreshes of cache entries without blocking reads on the same entries.

4.11.3 Java EE Gzipping Servlet Filter

- CachingFilter - an abstract, extensible caching filter.
- SimplePageCachingFilter

  A high performance Java EE servlet filter that caches pages based on the request URI and Query String. It also gzips the pages and delivers them to browsers either gzipped or ungzipped depending on the HTTP request headers. Use to cache entire Servlet pages, whether from JSP, velocity, or any other rendering technology.

  Tested with Orion and Tomcat.
- SimplePageFragmentCachingFilter

  A high performance Java EE filter that caches page fragments based on the request URI and Query String. Use with Servlet request dispatchers to cache parts of pages, whether from JSP, velocity, or any other rendering technology. Can be used from JSPs using jsp:include.

  Tested with Orion and Tomcat.
- Works with Servlet 2.3 and Servlet 2.4 specifications.

4.11.4 Cacheable Commands

This is the trusty old command pattern with a twist: asynchronous behaviour, fault tolerance and caching. Creates a command, caches it and then attempts to execute it.

4.11.5 Works with Hibernate

Tested with Hibernate2.1.8 and Hibernate3.1.3, which can utilise all of the new features except for Object API and multiple session factories each using a different Ehcache CacheManager.

A new `net.sf.ehcache.hibernate.EhCacheProvider` makes those additional features available to Hibernate-3.1.3. A version of the new provider should make it into the Hibernate3.2 release.

4.11.6 Works with Google App Engine

Ehcache-1.6 is compatible with Google App Engine.

See the Google App Engine HowTo.

4.12 High Quality

4.12.1 High Test Coverage

The Ehcache team believe that the first and most important quality measure is a well designed and comprehensive test suite.

Ehcache has a relatively high 86% test coverage of source code. This has edged higher over time. Clover enforces the test coverage. Most of the missing 14% is logging and exception paths.

4.12.2 Automated Load, Limit and Performance System Tests

The Ehcache JUnit test suite contains some long-running system tests which place high load on different Ehcache subsystems to the point of failure and then are back off to just below that point. The same is done with limits such as the amount of Elements that can fit in a given heap size. The same is also done with performance testing of each subsystem and the whole together. The same is also done with network tests for cache replication.

The tests serve a number of purposes:

- establishing well understood metrics and limits
- preventing regressions
- reproducing any reported issues in production
- Allowing the design principle of graceful degradation to be achieved. For example, the asynchronous cache replicator uses SoftReferences for queued messages, so that the messages will be reclaimed before before an OutOfMemoryError occurs, thus favouring stability over replication.

4.12.3 Specific Concurrency Testing

Ehcache also has concurrency testing, which typically uses 50 concurrent threads hammering a piece of code. The test suites are also run on multi-core or multi-cpu machines so that concurrency is real rather than simulated. Additionally, every concurrency related issue that has ever been anticipated or resulted in a bug report has a unit test which prevents the condition from recurring. There are no reported issues that have not been reproduced in a unit test.

Concurrency unit tests are somewhat difficult to write, and are often overlooked. The team considers these tests a major factor in ehcache's quality.

4.12.4 Production tested

Ehcache came about in the first place because of production issues with another open source cache.

Final release versions of Ehcache have been production tested on a very busy e-commerce site, supporting thousands of concurrent users, gigabyte size caches on large multi-cpu machines. It has been the experience of the team that most threading issues do not surface until this type of load has been applied. Once an issue has been identified and investigated a concurrency unit test can then be crafted.

4.12.5 Fully documented

A core belief held by the project team is that a project needs good documentation to be useful.

In ehcache, this is manifested by:

- comprehensive written documentation
- Complete, meaningful JavaDoc for every package, class and public and protected method. Checkstyle rules enforce this level of documentation.
- an up-to-date FAQ

4.12.6 Trusted by Popular Frameworks

Ehcache is used extensively. See the Who is Using? page, or browse Google.

4.12.7 Conservative Commit policy

Projects like Linux maintain their quality through a restricted change process, whereby changes are submitted as patches, then reviewed by the maintainer and included, or modified. Ehcache follows the same process.

4.12.8 Full public information on the history of every bug

Through the SourceForge project bug tracker, the full history of all bugs are shown, including current status. We take this for granted in an open source project, as this is typically a feature that all open source projects have, but this transparency makes it possible to gauge the quality and riskiness of a library, something not usually possible in commercial products.

4.12.9 Responsiveness to serious bugs

The Ehcache team is serious about quality. If one user is having a problem, it probably means others are too, or will have. The Ehcache team use Ehcache themselves in production. Every effort will be made to provide fixes for serious production problems as soon as possible. These will be committed to trunk. From there an affected user can apply the fix to their own branch.

4.13 Open Source Licensing

4.13.1 Apache 2.0 license

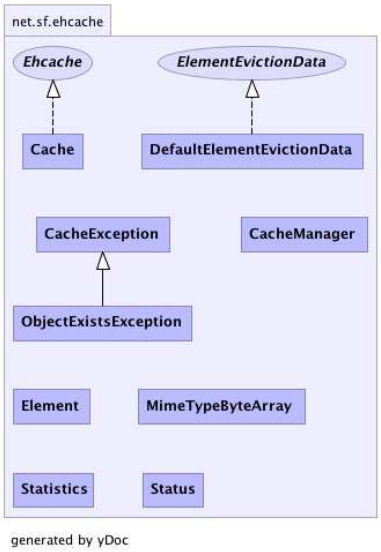
Ehcache's original Apache1.1 copyright and licensing was reviewed and approved by the Apache Software Foundation, making Ehcache suitable for use in Apache projects. ehcache-1.2 is released under the updated Apache 2.0 license.

The Apache license is also friendly one, making it safe and easy to include Ehcache in other open source projects or commercial products.

Chapter 5

Key Ehcache Concepts

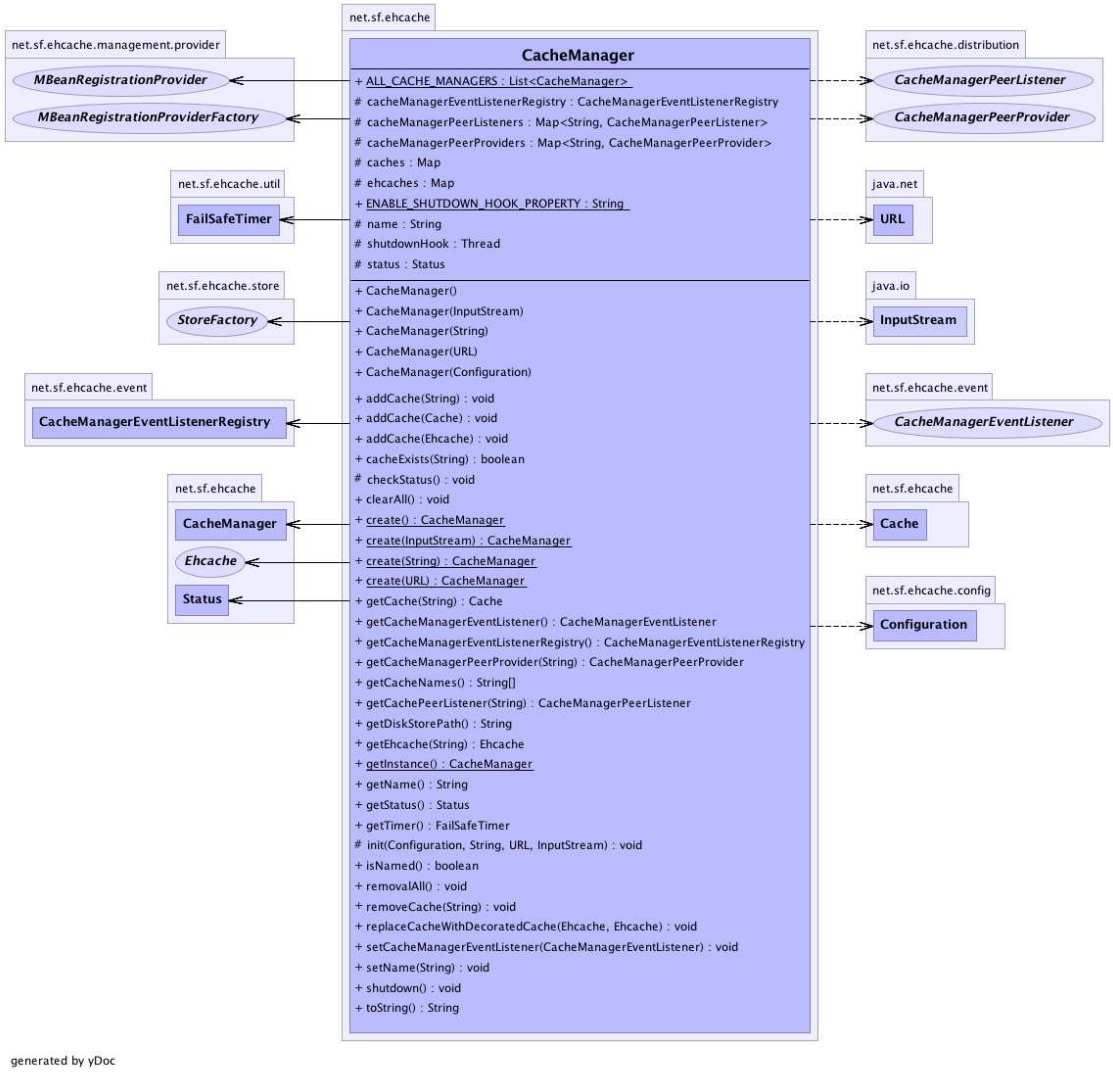
5.1 Key Ehcache Classes



Top Level Package Diagram

Ehcache consists of a `CacheManager`, which manages caches. Caches contain elements, which are essentially name value pairs. Caches are physically implemented either in-memory, or on disk.

5.1.1 CacheManager



CacheManager Class Diagram

The `CacheManager` comprises Caches which in turn comprise Elements.

Creation of, access to and removal of caches is controlled by the `CacheManager`.

CacheManager Creation Modes

`CacheManager` supports two creation modes: singleton and instance.

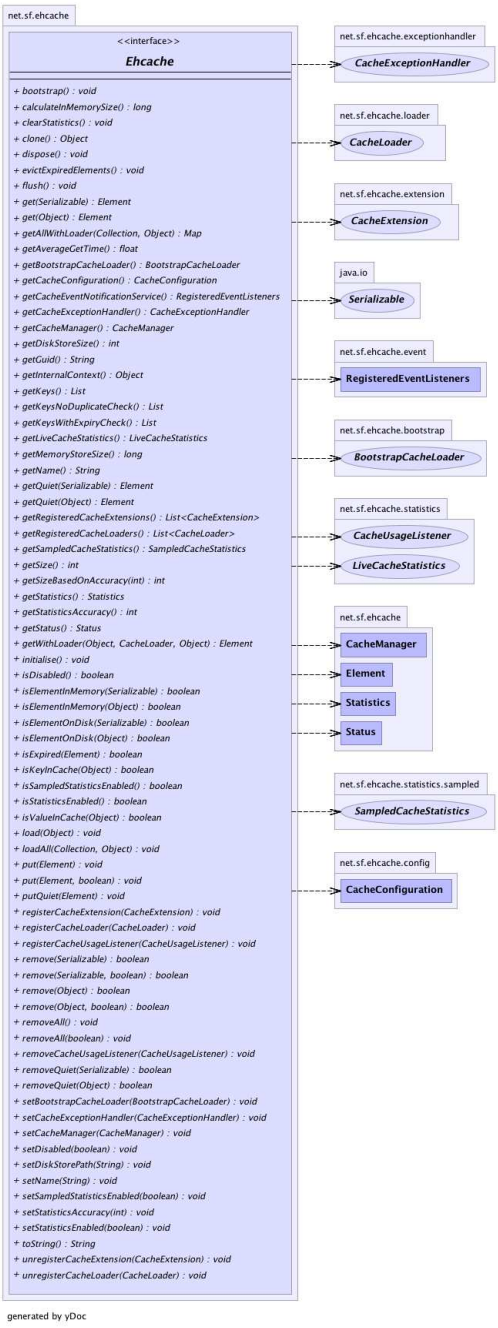
Singleton Mode Ehcache-1.1 supported only one `CacheManager` instance which was a singleton. CacheManager can still be used in this way using the static factory methods.

Instance Mode From ehcache-1.2, CacheManager has constructors which mirror the various static create methods. This enables multiple CacheManagers to be created and used concurrently. Each CacheManager requires its own configuration.

If the Caches under management use only the MemoryStore, there are no special considerations. If Caches use the DiskStore, the diskStore path specified in each CacheManager configuration should be unique. When a new CacheManager is created, a check is made that there are no other CacheManagers using the same diskStore path. If there are, a CacheException is thrown. If a CacheManager is part of a cluster, there will also be listener ports which must be unique.

Mixed Singleton and Instance Mode If an application creates instances of CacheManager using a constructor, and also calls a static create method, there will exist a singleton instance of CacheManager which will be returned each time the create method is called together with any other instances created via constructor. The two types will coexist peacefully.

5.1.2 Ehcache



Ehcache Interface Diagram

All caches implement the `Ehcache` interface. A cache has a name and attributes. Each cache contains Elements.

A Cache in Ehcache is analogous to a cache region in other caching systems.

Cache elements are stored in the `MemoryStore`. Optionally they also overflow to a `DiskStore`.

5.1.3 Element

Element Class Diagram

An element is an atomic entry in a cache. It has a key, a value and a record of accesses. Elements are put into and removed from caches. They can also expire and be removed by the Cache, depending on the Cache settings.

As of ehcache-1.2 there is an API for Objects in addition to the one for Serializable. Non-serializable Objects can use all parts of Ehcache except for DiskStore and replication. If an attempt is made to persist or replicate them they are discarded without error and with a DEBUG level log message.

The APIs are identical except for the return methods from Element. Two new methods on Element: getObjectValue and getKeyValue are the only API differences between the Serializable and Object APIs. This makes it very easy to start with caching Objects and then change your Objects to Seralizable to participate in the extra features when needed. Also a large number of Java classes are simply not Serializable.

5.2 Cache Usage Patterns

Caches can be used in different ways. Each of these ways follows a cache usage pattern. Ehcache supports the following:

- direct manipulation
- pull-through
- self-populating

5.2.1 Direct Manipulation

Here, to put something in the cache you do `cache.put(Element element)` and to get something from the cache you do `cache.get(Object key)`.

You are aware you are using a cache and you are doing so consciously.

5.2.2 Self Populating

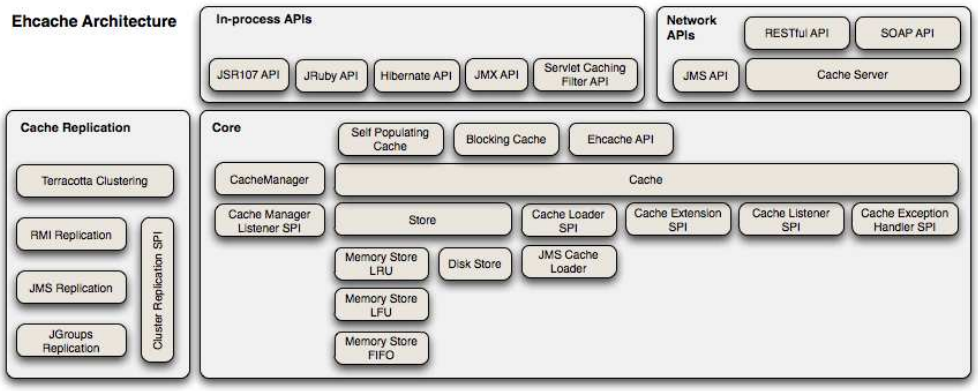
Here, you just do gets to the cache using `cache.get(Object key)`. The cache itself knows how to populate an entry.

See the SelfPopulatingCache for more on this pattern.

Chapter 6

Architecture

This diagram shows the architecture of Ehcache. It is current as of Ehcache-1.6.2.



Chapter 7

Configuration

Caches can be configured in Ehcache either declaratively, in xml, or by creating them programmatically and specifying their parameters in the constructor.

While both approaches are fully supported it is generally a good idea to separate the cache configuration from runtime use. There are also these benefits:

- It is easy if you have all of your configuration in one place. Caches consume memory, and disk space. They need to be carefully tuned. You can see the total effect in a configuration file. You could do this code, but it would not as visible.
- Cache configuration can be changed at deployment time.
- Configuration errors can be checked for at start-up, rather than causing a runtime error.

This chapter covers XML declarative configuration. See the Code samples for programmatic configuration.

Ehcache is redistributed by lots of projects. They may or may not provide a sample Ehcache XML configuration file. If one is not provided, download Ehcache from http://ehcache.org. It, and the ehcache.xsd is provided in the distribution.

7.1 ehcache.xsd

Ehcache configuration files must be comply with the Ehcache XML schema, ehcache.xsd, reproduced below.

It can also be downloaded from http://ehcache.org/ehcache.xsd.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
    version="1.7">

  <xs:element name="ehcache" >
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" maxOccurs="1" ref="diskStore"/>
        <xs:element minOccurs="0" maxOccurs="1"
              ref="cacheManagerEventListenerFactory"/>
        <xs:element minOccurs="0" maxOccurs="unbounded"
              ref="cacheManagerPeerProviderFactory"/>
        <xs:element minOccurs="0" maxOccurs="unbounded"
```

```
            ref="cacheManagerPeerListenerFactory"/>
      <xs:element minOccurs="0" maxOccurs="1"
            ref="terracottaConfig"/>
      <xs:element ref="defaultCache"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="cache"/>
    </xs:sequence>
    <xs:attribute name="name" use="optional"/>
    <xs:attribute name="updateCheck" use="optional" type="xs:boolean" default="true"/>
    <xs:attribute name="monitoring" use="optional" type="monitoringType"
    default="autodetect"/>
  </xs:complexType>
</xs:element>
<xs:element name="diskStore">
  <xs:complexType>
    <xs:attribute name="path" use="optional" />
  </xs:complexType>
</xs:element>
<xs:element name="cacheManagerEventListenerFactory">
  <xs:complexType>
    <xs:attribute name="class" use="required"/>
    <xs:attribute name="properties" use="optional"/>
    <xs:attribute name="propertySeparator" use="optional"/>
  </xs:complexType>
</xs:element>
<xs:element name="cacheManagerPeerProviderFactory">
  <xs:complexType>
    <xs:attribute name="class" use="required"/>
    <xs:attribute name="properties" use="optional"/>
    <xs:attribute name="propertySeparator" use="optional"/>
  </xs:complexType>
</xs:element>
<xs:element name="cacheManagerPeerListenerFactory">
  <xs:complexType>
    <xs:attribute name="class" use="required"/>
    <xs:attribute name="properties" use="optional"/>
    <xs:attribute name="propertySeparator" use="optional"/>
  </xs:complexType>
</xs:element>
<xs:element name="terracottaConfig">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="tc-config" minOccurs="0" maxOccurs="1">
        <xs:complexType>
          <xs:sequence>
            <xs:any minOccurs="0" maxOccurs="unbounded" processContents="skip" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="url" use="optional" default="localhost:9510"/>
    <xs:attribute name="registerStatsMBean" type="xs:boolean" use="optional"/>
  </xs:complexType>
</xs:element>
<!-- add clone support for addition of cacheExceptionHandler. Important! -->
<xs:element name="defaultCache">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="cacheEventListenerFactory"/>
```

```
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="cacheExtensionFactory"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="cacheLoaderFactory"/>
      <xs:element minOccurs="0" maxOccurs="1" ref="bootstrapCacheLoaderFactory"/>
      <xs:element minOccurs="0" maxOccurs="1" ref="cacheExceptionHandlerFactory"/>
      <xs:element minOccurs="0" maxOccurs="1" ref="terracotta"/>
    </xs:sequence>
    <xs:attribute name="diskExpiryThreadIntervalSeconds" use="optional" type="xs:integer"/>
    <xs:attribute name="diskSpoolBufferSizeMB" use="optional" type="xs:integer"/>
    <xs:attribute name="diskPersistent" use="optional" type="xs:boolean"/>
    <xs:attribute name="eternal" use="required" type="xs:boolean"/>
    <xs:attribute name="maxElementsInMemory" use="required" type="xs:integer"/>
    <xs:attribute name="clearOnFlush" use="optional" type="xs:boolean"/>
    <xs:attribute name="memoryStoreEvictionPolicy" use="optional" type="xs:string"/>
    <xs:attribute name="overflowToDisk" use="required" type="xs:boolean"/>
    <xs:attribute name="timeToIdleSeconds" use="optional" type="xs:integer"/>
    <xs:attribute name="timeToLiveSeconds" use="optional" type="xs:integer"/>
    <xs:attribute name="maxElementsOnDisk" use="optional" type="xs:integer"/>
  </xs:complexType>
</xs:element>
<xs:element name="cache">
  <xs:complexType>
    <xs:sequence >
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="cacheEventListenerFactory"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="cacheExtensionFactory"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="cacheLoaderFactory"/>
      <xs:element minOccurs="0" maxOccurs="1" ref="bootstrapCacheLoaderFactory"/>
      <xs:element minOccurs="0" maxOccurs="1" ref="cacheExceptionHandlerFactory"/>
      <xs:element minOccurs="0" maxOccurs="1" ref="terracotta"/>
    </xs:sequence>
    <xs:attribute name="diskExpiryThreadIntervalSeconds" use="optional"
    type="xs:integer"/>
    <xs:attribute name="diskSpoolBufferSizeMB" use="optional" type="xs:integer"/>
    <xs:attribute name="diskPersistent" use="optional" type="xs:boolean"/>
    <xs:attribute name="eternal" use="required" type="xs:boolean"/>
    <xs:attribute name="maxElementsInMemory" use="required" type="xs:integer"/>
    <xs:attribute name="memoryStoreEvictionPolicy" use="optional" type="xs:string"/>
    <xs:attribute name="clearOnFlush" use="optional" type="xs:boolean"/>
    <xs:attribute name="name" use="required" type="xs:string"/>
    <xs:attribute name="overflowToDisk" use="required" type="xs:boolean"/>
    <xs:attribute name="timeToIdleSeconds" use="optional" type="xs:integer"/>
    <xs:attribute name="timeToLiveSeconds" use="optional" type="xs:integer"/>
    <xs:attribute name="maxElementsOnDisk" use="optional" type="xs:integer"/>
  </xs:complexType>
</xs:element>
<xs:element name="cacheEventListenerFactory">
  <xs:complexType>
    <xs:attribute name="class" use="required"/>
    <xs:attribute name="properties" use="optional"/>
    <xs:attribute name="propertySeparator" use="optional"/>
  </xs:complexType>
</xs:element>
<xs:element name="bootstrapCacheLoaderFactory">
  <xs:complexType>
    <xs:attribute name="class" use="required"/>
    <xs:attribute name="properties" use="optional"/>
    <xs:attribute name="propertySeparator" use="optional"/>
  </xs:complexType>
</xs:element>
```

```
  <xs:element name="cacheExtensionFactory">
    <xs:complexType>
      <xs:attribute name="class" use="required"/>
      <xs:attribute name="properties" use="optional"/>
      <xs:attribute name="propertySeparator" use="optional"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="cacheExceptionHandlerFactory">
    <xs:complexType>
      <xs:attribute name="class" use="required"/>
      <xs:attribute name="properties" use="optional"/>
      <xs:attribute name="propertySeparator" use="optional"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="cacheLoaderFactory">
    <xs:complexType>
      <xs:attribute name="class" use="required"/>
      <xs:attribute name="properties" use="optional"/>
      <xs:attribute name="propertySeparator" use="optional"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="terracotta">
    <xs:complexType>
      <xs:attribute name="clustered" use="optional" type="xs:boolean" default="true"/>
      <xs:attribute name="valueMode" use="optional"
        type="terracottaCacheValueType" default="serialization"/>
      <xs:attribute name="coherentReads" use="optional" type="xs:boolean" default="true"/>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="monitoringType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="autodetect" />
      <xs:enumeration value="on" />
      <xs:enumeration value="off" />
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="terracottaCacheValueType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="serialization" />
      <xs:enumeration value="identity" />
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

7.2 ehcache-failsafe.xml

If the CacheManager default constructor or factory method is called, Ehcache looks for a file called ehcache.xml in the top level of the classpath. Failing that it looks for ehcache-failsafe.xml in the classpath. ehcache-failsafe.xml is packaged in the Ehcache jar and should always be found.

ehcache-failsafe.xml provides an extremely simple default configuration to enable users to get started before they create their own ehcache.xml.

If it used Ehcache will emit a warning, reminding the user to set up a proper configuration.

The meaning of the elements and attributes are explained in the section on ehcache.xml.

```
<ehcache>
    <diskStore path="java.io.tmpdir"/>
    <defaultCache
            maxElementsInMemory="10000"
            eternal="false"
            timeToIdleSeconds="120"
            timeToLiveSeconds="120"
            overflowToDisk="true"
            maxElementsOnDisk="10000000"
            diskPersistent="false"
            diskExpiryThreadIntervalSeconds="120"
            memoryStoreEvictionPolicy="LRU"
            />
</ehcache>
```

7.3 ehcache.xml and other configuration files

Prior to ehcache-1.6, Ehcache only supported ASCII ehcache.xml configuration files. Since ehcache-1.6, UTF8 is supported, so that configuration can use Unicode. As UTF8 is backwardly compatible with ASCII, no conversion is necessary.

If the CacheManager default constructor or factory method is called, Ehcache looks for a file called ehcache.xml in the top level of the classpath.

The non-default creation methods allow a configuration file to be specified which can be called anything.

One XML configuration is required for each CacheManager that is created. It is an error to use the same configuration, because things like directory paths and listener ports will conflict. Ehcache will attempt to resolve conflicts and will emit a warning reminding the user to configure a separate configuration for multiple CacheManagers with conflicting settings.

The sample ehcache.xml, which is included in the Ehcache distribution is reproduced below. The sample contains full commentary required to configure each element. Further information can be found in specific chapters in the Guide.

It can also be downloaded from http://ehcache.org/ehcache.xml.

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

<!--
CacheManager Configuration
==========================
An ehcache.xml corresponds to a single CacheManager.

See instructions below or the ehcache schema (ehcache.xsd) on how to configure.

System property tokens can be specified in this file which are replaced when the
configuration is loaded. For example multicastGroupPort=${multicastGroupPort}
can be replaced with the System property either from an environment variable or
a system property specified with a command line switch such as
-DmulticastGroupPort=4446.

The attributes of <ehcache> are:
* name - an optional name for the CacheManager.  The name is optional and primarily used
for documentation or to distinguish Terracotta clustered cache state.  With Terracotta
clustered caches, a combination of CacheManager name and cache name uniquely identify a
particular cache store in the Terracotta clustered memory.
* updateCheck - an optional boolean flag specifying whether this CacheManager should check
```

```
for new versions of Ehcache over the Internet.  If not specified, updateCheck="true".
* monitoring - an optional setting that determines whether the CacheManager should
automatically register the SampledCacheMBean with the system MBean server.

Currently, this monitoring is only useful when using Terracotta clustering and using the
Terracotta Developer Console. With the "autodetect" value, the presence of Terracotta
clustering will be detected and monitoring, via the Developer Console, will be enabled.
Other allowed values are "on" and "off".  The default is "autodetect". This setting does
not perform any function when used with JMX monitors.
-->
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="ehcache.xsd"
             updateCheck="true" monitoring="autodetect">

<!--
DiskStore configuration
=======================

The diskStore element is optional. To turn off disk store path creation, comment out
the diskStore element below.

Configure it if you have overflowToDisk or diskPersistent enabled for any cache.

If it is not configured, and a cache is created which requires a disk store, a warning
will be issued and java.io.tmpdir will automatically be used.

diskStore has only one attribute - "path". It is the path to the directory where
.data and .index files will be created.

If the path is one of the following Java System Property it is replaced by its value
in the running VM. For backward compatibility these are not specified without being
enclosed in the ${token} replacement syntax.

The following properties are translated:
* user.home - User’s home directory
* user.dir - User’s current working directory
* java.io.tmpdir - Default temp file path
* ehcache.disk.store.dir - A system property you would normally specify on the command
 line e.g. java -Dehcache.disk.store.dir=/u01/myapp/diskdir ...

Subdirectories can be specified below the property e.g. java.io.tmpdir/one

-->
<diskStore path="java.io.tmpdir"/>

<!--
CacheManagerEventListener
=========================
Specifies a CacheManagerEventListenerFactory which is notified when Caches are added
or removed from the CacheManager.

The attributes of CacheManagerEventListenerFactory are:
* class - a fully qualified factory class name
* properties - comma separated properties having meaning only to the factory.

Sets the fully qualified class name to be registered as the CacheManager event listener.

The events include:
```

```
* adding a Cache
* removing a Cache

Callbacks to listener methods are synchronous and unsynchronized. It is the
responsibility of the implementer to safely handle the potential performance and
thread safety issues depending on what their listener is doing.

If no class is specified, no listener is created. There is no default.
-->
<cacheManagerEventListenerFactory class="" properties=""/>


<!--
CacheManagerPeerProvider
========================
(For distributed operation)

Specifies a CacheManagerPeerProviderFactory which will be used to create a
CacheManagerPeerProvider, which discovers other CacheManagers in the cluster.

One or more providers can be configured. The first one in the ehcache.xml is the
default, which is used for replication and bootstrapping.

The attributes of cacheManagerPeerProviderFactory are:
* class - a fully qualified factory class name
* properties - comma separated properties having meaning only to the factory.

Providers are available for RMI, JGroups and JMS as shown following.

RMICacheManagerPeerProvider
+++++++++++++++++++++++++++

Ehcache comes with a built-in RMI-based distribution system with two means of discovery
of CacheManager peers participating in the cluster:

* automatic, using a multicast group. This one automatically discovers peers and detects
  changes such as peers entering and leaving the group
* manual, using manual rmiURL configuration. A hardcoded list of peers is provided at
  configuration time.

Configuring Automatic Discovery:
Automatic discovery is configured as per the following example:
<cacheManagerPeerProviderFactory
                    class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
                    properties="hostName=fully_qualified_hostname_or_ip,
                                peerDiscovery=automatic, multicastGroupAddress=230.0.0.1,
                                multicastGroupPort=4446, timeToLive=32"/>

Valid properties are:
* peerDiscovery (mandatory) - specify "automatic"
* multicastGroupAddress (mandatory) - specify a valid multicast group address
* multicastGroupPort (mandatory) - specify a dedicated port for the multicast heartbeat
  traffic
* timeToLive - specify a value between 0 and 255 which determines how far the packets
  will propagate.

  By convention, the restrictions are:
  0   - the same host
```

```
  1   - the same subnet
  32  - the same site
  64  - the same region
  128 - the same continent
  255 - unrestricted

 * hostName - the hostname or IP of the interface to be used for sending and receiving
   multicast packets (relevant to mulithomed hosts only)

Configuring Manual Discovery:
Manual discovery requires a unique configuration per host. It is contains a list of
rmiURLs for the peers, other than itself. So, if we have server1, server2 and server3
the configuration will be:

In server1's configuration:
<cacheManagerPeerProviderFactory class=
                      "net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
                      properties="peerDiscovery=manual,
                      rmiUrls=//server2:40000/sampleCache1|//server3:40000/sampleCache1
                      | //server2:40000/sampleCache2|//server3:40000/sampleCache2"
                      propertySeparator="," />

In server2's configuration:
<cacheManagerPeerProviderFactory class=
                      "net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
                      properties="peerDiscovery=manual,
                      rmiUrls=//server1:40000/sampleCache1|//server3:40000/sampleCache1
                      | //server1:40000/sampleCache2|//server3:40000/sampleCache2"
                      propertySeparator="," />

In server3's configuration:
<cacheManagerPeerProviderFactory class=
                      "net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
                      properties="peerDiscovery=manual,
                      rmiUrls=//server1:40000/sampleCache1|//server2:40000/sampleCache1
                      | //server1:40000/sampleCache2|//server2:40000/sampleCache2"
                      propertySeparator="," />


Valid properties are:
* peerDiscovery (mandatory) - specify "manual"
* rmiUrls (mandatory) - specify a pipe separated list of rmiUrls, in the form
                        //hostname:port
* hostname (optional) - the hostname is the hostname of the remote CacheManager peer.
  The port is the listening port of the RMICacheManagerPeerListener of the remote
  CacheManager peer.

JGroupsCacheManagerPeerProvider
++++++++++++++++++++++++++++++
<cacheManagerPeerProviderFactory
 class="net.sf.ehcache.distribution.jgroups.JGroupsCacheManagerPeerProviderFactory"
 properties="connect=UDP(mcast_addr=231.12.21.132;mcast_port=45566;ip_ttl=32;
 mcast_send_buf_size=150000;mcast_recv_buf_size=80000):
 PING(timeout=2000;num_initial_members=6):
 MERGE2(min_interval=5000;max_interval=10000):
 FD_SOCK:VERIFY_SUSPECT(timeout=1500):
 pbcast.NAKACK(gc_lag=10;retransmit_timeout=3000):
 UNICAST(timeout=5000):
```

```
 pbcast.STABLE(desired_avg_gossip=20000):
 FRAG:
 pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;shun=false;print_local_addr=false)"
 propertySeparator="::"
        />
 The only property necessary is the connect String used by jgroups to configure itself.
 Refer to the JGroups documentation for explanation of all the protocols. The example above
 uses UDP multicast. If the connect property is not specified the default JGroups connection
 will be used.


JMSCacheManagerPeerProviderFactory
++++++++++++++++++++++++++++++++
<cacheManagerPeerProviderFactory
        class="net.sf.ehcache.distribution.jms.JMSCacheManagerPeerProviderFactory"
        properties="..."
        propertySeparator=","
        />

The JMS PeerProviderFactory uses JNDI to maintain message queue independence. Refer to the
    manual for full configuration examples using ActiveMQ and Open Message Queue.

Valid properties are:
* initialContextFactoryName (mandatory) - the name of the factory used to create the
    message queue initial context.
* providerURL (mandatory) - the JNDI configuration information for the service provider
    to use.
* topicConnectionFactoryBindingName (mandatory) - the JNDI binding name for the
    TopicConnectionFactory
* topicBindingName (mandatory) - the JNDI binding name for the topic name
* getQueueBindingName (mandatory only if using jmsCacheLoader) - the JNDI binding name for
    the queue name
* securityPrincipalName - the JNDI java.naming.security.principal
* securityCredentials - the JNDI java.naming.security.credentials
* urlPkgPrefixes - the JNDI java.naming.factory.url.pkgs
* userName - the user name to use when creating the TopicConnection to the Message Queue
* password - the password to use when creating the TopicConnection to the Message Queue
* acknowledgementMode - the JMS Acknowledgement mode for both publisher and subscriber.
    The available choices are AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE and SESSION_TRANSACTED.
      The default is AUTO_ACKNOWLEDGE.
-->
<cacheManagerPeerProviderFactory
        class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
        properties="peerDiscovery=automatic,
                    multicastGroupAddress=230.0.0.1,
                    multicastGroupPort=4446, timeToLive=1"
        propertySeparator=","
        />


<!--
CacheManagerPeerListener
========================
(Enable for distributed operation)

Specifies a CacheManagerPeerListenerFactory which will be used to create a
CacheManagerPeerListener, which listens for messages from cache replicators participating
 in the cluster.
```

```
The attributes of cacheManagerPeerListenerFactory are:
class - a fully qualified factory class name
properties - comma separated properties having meaning only to the factory.

Ehcache comes with a built-in RMI-based distribution system. The listener component is
RMICacheManagerPeerListener which is configured using
RMICacheManagerPeerListenerFactory. It is configured as per the following example:

<cacheManagerPeerListenerFactory
    class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"
    properties="hostName=fully_qualified_hostname_or_ip,
                port=40001,
                remoteObjectPort=40002,
                socketTimeoutMillis=120000"
                propertySeparator="," />

All properties are optional. They are:
* hostName - the hostName of the host the listener is running on. Specify
  where the host is multihomed and you want to control the interface over which cluster
  messages are received. Defaults to the host name of the default interface if not
  specified.
* port - the port the RMI Registry listener listens on. This defaults to a free port
  if not specified.
* remoteObjectPort - the port number on which the remote objects bound in the registry
  receive calls. This defaults to a free port if not specified.
* socketTimeoutMillis - the number of ms client sockets will stay open when sending
  messages to the listener. This should be long enough for the slowest message.
  If not specified it defaults to 120000ms.

-->
<cacheManagerPeerListenerFactory
        class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"/>

<!--
TerracottaConfig
========================
(Enable for Terracotta clustered operation)

Note: You need to install and run one or more Terracotta servers to use Terracotta
clustering. See http://www.terracotta.org/web/display/orgsite/Download.

Specifies a TerracottaConfig which will be used to configure the Terracotta
runtime for this CacheManager.

Configuration can be specified in two main ways: by reference to a source of
configuration or by use of an embedded Terracotta configuration file.

To specify a reference to a source (or sources) of configuration, use the url
attribute.  The url attribute must contain a comma-separated list of:
* path to Terracotta configuration file (usually named tc-config.xml)
* URL to Terracotta configuration file
* <server host>:<port> of running Terracotta Server instance

Simplest example for pointing to a Terracotta server on this machine:
<terracottaConfig url="localhost:9510"/>

Example using a path to Terracotta configuration file:
```

```
<terracottaConfig url="/app/config/tc-config.xml"/>

Example using a URL to a Terracotta configuration file:
<terracottaConfig url="http://internal/ehcache/app/tc-config.xml"/>

Example using multiple Terracotta server instance URLs (for fault tolerance):
<terracottaConfig url="host1:9510,host2:9510,host3:9510"/>

To embed a Terracotta configuration file within the ehcache configuration, simply
place a normal Terracotta XML config within the <terracottaConfig> element.

Example:
<terracottaConfig>
    <tc-config>
        <servers>
            <server host="server1" name="s1"/>
            <server host="server2" name="s2"/>
        </servers>
        <clients>
            <logs>app/logs-%i</logs>
        </clients>
    </tc-config>
</terracottaConfig>

For more information on the Terracotta configuration, see the Terracotta documentation.
-->

<!--
Cache configuration
===================

The following attributes are required.

name:
Sets the name of the cache. This is used to identify the cache. It must be unique.

maxElementsInMemory:
Sets the maximum number of objects that will be created in memory

maxElementsOnDisk:
Sets the maximum number of objects that will be maintained in the DiskStore
The default value is zero, meaning unlimited.

eternal:
Sets whether elements are eternal. If eternal,  timeouts are ignored and the
element is never expired.

overflowToDisk:
Sets whether elements can overflow to disk when the memory store
has reached the maxInMemory limit.

The following attributes and elements are optional.

timeToIdleSeconds:
Sets the time to idle for an element before it expires.
i.e. The maximum amount of time between accesses before an element expires
Is only used if the element is not eternal.
Optional attribute. A value of 0 means that an Element can idle for infinity.
```

The default value is 0.

timeToLiveSeconds:
Sets the time to live for an element before it expires.
i.e. The maximum time between creation time and when an element expires.
Is only used if the element is not eternal.
Optional attribute. A value of 0 means that and Element can live for infinity.
The default value is 0.

diskPersistent:
Whether the disk store persists between restarts of the Virtual Machine.
The default value is false.

diskExpiryThreadIntervalSeconds:
The number of seconds between runs of the disk expiry thread. The default value
is 120 seconds.

diskSpoolBufferSizeMB:
This is the size to allocate the DiskStore for a spool buffer. Writes are made
to this area and then asynchronously written to disk. The default size is 30MB.
Each spool buffer is used only by its cache. If you get OutOfMemory errors consider
lowering this value. To improve DiskStore performance consider increasing it. Trace level
logging in the DiskStore will show if put back ups are occurring.

clearOnFlush:
whether the MemoryStore should be cleared when flush() is called on the cache.
By default, this is true i.e. the MemoryStore is cleared.

memoryStoreEvictionPolicy:
Policy would be enforced upon reaching the maxElementsInMemory limit. Default
policy is Least Recently Used (specified as LRU). Other policies available -
First In First Out (specified as FIFO) and Less Frequently Used
(specified as LFU)

Cache elements can also contain sub elements which take the same format of a factory class
and properties. Defined sub-elements are:

* cacheEventListenerFactory - Enables registration of listeners for cache events, such as
  put, remove, update, and expire.

* bootstrapCacheLoaderFactory - Specifies a BootstrapCacheLoader, which is called by a
  cache on initialisation to prepopulate itself.

* cacheExtensionFactory - Specifies a CacheExtension, a generic mechansim to tie a class
  which holds a reference to a cache to the cache lifecycle.

* cacheExceptionHandlerFactory - Specifies a CacheExceptionHandler, which is called when
  cache exceptions occur.

* cacheLoaderFactory - Specifies a CacheLoader, which can be used both asynchronously and
  synchronously to load objects into a cache. More than one cacheLoaderFactory element
  can be added, in which case the loaders form a chain which are executed in order. If a
  loader returns null, the next in chain is called.

RMI Cache Replication
++++++++++++++++++++

Each cache that will be distributed needs to set a cache event listener which replicates messages to the other CacheManager peers. For the built-in RMI implementation this is done by adding a cacheEventListenerFactory element of type RMICacheReplicatorFactory to each distributed cache's configuration as per the following example:

```
<cacheEventListenerFactory class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"
     properties="replicateAsynchronously=true,
     replicatePuts=true,
     replicatePutsViaCopy=false,
     replicateUpdates=true,
     replicateUpdatesViaCopy=true,
     replicateRemovals=true
     asynchronousReplicationIntervalMillis=<number of milliseconds"
     propertySeparator="," />
```

The RMICacheReplicatorFactory recognises the following properties:

* replicatePuts=true|false - whether new elements placed in a cache are replicated to others. Defaults to true.

* replicatePutsViaCopy=true|false - whether the new elements are copied to other caches (true), or whether a remove message is sent. Defaults to true.

* replicateUpdates=true|false - whether new elements which override an element already existing with the same key are replicated. Defaults to true.

* replicateRemovals=true - whether element removals are replicated. Defaults to true.

* replicateAsynchronously=true | false - whether replications are asynchronous (true) or synchronous (false). Defaults to true.

* replicateUpdatesViaCopy=true | false - whether the new elements are copied to other caches (true), or whether a remove message is sent. Defaults to true.

* asynchronousReplicationIntervalMillis=<number of milliseconds> - The asynchronous replicator runs at a set interval of milliseconds. The default is 1000. The minimum is 10. This property is only applicable if replicateAsynchronously=true

JGroups Replication
++++++++++++++++++++

For the Jgroups replication this is done with:

```
<cacheEventListenerFactory
    class="net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory"
                properties="replicateAsynchronously=true, replicatePuts=true,
                replicateUpdates=true, replicateUpdatesViaCopy=false,
                replicateRemovals=true,asynchronousReplicationIntervalMillis=1000"/>
```

This listener supports the same properties as the RMICacheReplicationFactory.

JMS Replication
+++++++++++++++

For JMS-based replication this is done with:

```
<cacheEventListenerFactory
      class="net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory"
      properties="replicateAsynchronously=true,
```

```
                    replicatePuts=true,
                    replicateUpdates=true,
                    replicateUpdatesViaCopy=true,
                    replicateRemovals=true,
                    asynchronousReplicationIntervalMillis=1000"
        propertySeparator=","/>
```

This listener supports the same properties as the RMICacheReplicationFactory.

Cluster Bootstrapping
++++++++++++++++++++

Bootstrapping a cluster may use a different mechanism to replication. e.g you can mix JMS replication with bootstrap via RMI - just make sure you have the cacheManagerPeerProviderFactory and cacheManagerPeerListenerFactory configured.

There are two bootstrapping mechanisms: RMI and JGroups.

RMI Bootstrap

The RMIBootstrapCacheLoader bootstraps caches in clusters where RMICacheReplicators are used. It is configured as per the following example:

```
<bootstrapCacheLoaderFactory
    class="net.sf.ehcache.distribution.RMIBootstrapCacheLoaderFactory"
    properties="bootstrapAsynchronously=true, maximumChunkSizeBytes=5000000"
    propertySeparator="," />
```

The RMIBootstrapCacheLoaderFactory recognises the following optional properties:

* bootstrapAsynchronously=true|false - whether the bootstrap happens in the background after the cache has started. If false, bootstrapping must complete before the cache is made available. The default value is true.

* maximumChunkSizeBytes=<integer> - Caches can potentially be very large, larger than the memory limits of the VM. This property allows the bootstraper to fetched elements in chunks. The default chunk size is 5000000 (5MB).

JGroups Bootstrap

Here is an example of bootstrap configuration using JGroups boostrap:

```
<bootstrapCacheLoaderFactory
    class="net.sf.ehcache.distribution.jgroups.JGroupsBootstrapCacheLoaderFactory"
                                properties="bootstrapAsynchronously=true"/>
```

The configuration properties are the same as for RMI above. Note that JGroups bootstrap only supports asynchronous bootstrap mode.

Cache Exception Handling

By default, most cache operations will propagate a runtime CacheException on failure. An interceptor, using a dynamic proxy, may be configured so that a CacheExceptionHandler can be configured to intercept Exceptions. Errors are not intercepted.

It is configured as per the following example:

```
  <cacheExceptionHandlerFactory class="com.example.ExampleExceptionHandlerFactory"
                                properties="logLevel=FINE"/>
```

Caches with ExceptionHandling configured are not of type Cache, but are of type Ehcache only, and are not available using CacheManager.getCache(), but using CacheManager.getEhcache().

Cache Loader

A default CacheLoader may be set which loads objects into the cache through asynchronous and synchronous methods on Cache. This is different to the bootstrap cache loader, which is used only in distributed caching.

It is configured as per the following example:

```
    <cacheLoaderFactory class="com.example.ExampleCacheLoaderFactory"
                                properties="type=int,startCounter=10"/>
```

Cache Extension

CacheExtensions are a general purpose mechanism to allow generic extensions to a Cache. CacheExtensions are tied into the Cache lifecycle.

CacheExtensions are created using the CacheExtensionFactory which has a <code>createCacheCacheExtension()</code> method which takes as a parameter a Cache and properties. It can thus call back into any public method on Cache, including, of course, the load methods.

Extensions are added as per the following example:

```
     <cacheExtensionFactory class="com.example.FileWatchingCacheRefresherExtensionFactory"
                        properties="refreshIntervalMillis=18000, loaderTimeout=3000,
                                    flushPeriod=whatever, someOtherProperty=someValue ..."/>
```

Terracotta Clustering

Cache elements can also contain information about whether the cache can be clustered with Terracotta. The <terracotta> sub-element has the following attributes:

* clustered=true|false - indicates whether this cache should be clustered with Terracotta. By default, if the <terracotta> element is included, clustered=true.
* valueMode=serialization|identity - indicates whether this cache should be clustered with serialized copies of the values or using Terracotta identity mode. By default, values will be cached in serialization mode which is similar to other replicated Ehcache modes. The identity mode is only available in certain Terracotta deployment scenarios and will maintain actual object identity of the keys and values across the cluster. In this case, all users of a value retrieved from the cache are using the same clustered value and must provide appropriate locking for any changes made to the value (or objects referred to by the value).
* coherentReads=true|false - indicates whether this cache should have coherent reads with guaranteed consistency across the cluster. By default, this setting is true. If you set this property to false, reads are allowed to check the local value without locking, possibly seeing stale values. This is a performance optimization with weaker concurrency guarantees and should generally be used with caches that contain read-only data or where the application can tolerate reading stale data.

The simplest example to indicate clustering:

```
    <terracotta/>

To indicate the cache should not be clustered (or remove the <terracotta> element
 altogether):

    <terracotta clustered="false"/>

To indicate the cache should be clustered using identity mode:

    <terracotta clustered="true" valueMode="identity"/>
-->

<!--
Mandatory Default Cache configuration. These settings will be applied to caches
created programmtically using CacheManager.add(String cacheName).

The defaultCache has an implicit name "default" which is a reserved cache name.
-->
<defaultCache
        maxElementsInMemory="10000"
        eternal="false"
        timeToIdleSeconds="120"
        timeToLiveSeconds="120"
        overflowToDisk="true"
        diskSpoolBufferSizeMB="30"
        maxElementsOnDisk="10000000"
        diskPersistent="false"
        diskExpiryThreadIntervalSeconds="120"
        memoryStoreEvictionPolicy="LRU"
        />

<!--
Sample caches. Following are some example caches. Remove these before use.
-->

<!--
Sample cache named sampleCache1
This cache contains a maximum in memory of 10000 elements, and will expire
an element if it is idle for more than 5 minutes and lives for more than
10 minutes.

If there are more than 10000 elements it will overflow to the
disk cache, which in this configuration will go to wherever java.io.tmp is
defined on your system. On a standard Linux system this will be /tmp"
-->
<cache name="sampleCache1"
       maxElementsInMemory="10000"
       maxElementsOnDisk="1000"
       eternal="false"
       overflowToDisk="true"
       diskSpoolBufferSizeMB="20"
       timeToIdleSeconds="300"
       timeToLiveSeconds="600"
       memoryStoreEvictionPolicy="LFU"
        />
```

```
<!--
Sample cache named sampleCache2
This cache has a maximum of 1000 elements in memory. There is no overflow to disk, so 1000
is also the maximum cache size. Note that when a cache is eternal, timeToLive and
timeToIdle are not used and do not need to be specified.
-->
<cache name="sampleCache2"
       maxElementsInMemory="1000"
       eternal="true"
       overflowToDisk="false"
       memoryStoreEvictionPolicy="FIFO"
        />


<!--
Sample cache named sampleCache3. This cache overflows to disk. The disk store is
persistent between cache and VM restarts. The disk expiry thread interval is set to 10
minutes, overriding the default of 2 minutes.
-->
<cache name="sampleCache3"
       maxElementsInMemory="500"
       eternal="false"
       overflowToDisk="true"
       timeToIdleSeconds="300"
       timeToLiveSeconds="600"
       diskPersistent="true"
       diskExpiryThreadIntervalSeconds="1"
       memoryStoreEvictionPolicy="LFU"
        />


<!--
Sample distributed cache named sampleDistributedCache1.
This cache replicates using defaults.
It also bootstraps from the cluster, using default properties.
-->
<cache name="sampleDistributedCache1"
       maxElementsInMemory="10"
       eternal="false"
       timeToIdleSeconds="100"
       timeToLiveSeconds="100"
       overflowToDisk="false">

    <cacheEventListenerFactory
            class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>
    <bootstrapCacheLoaderFactory
            class="net.sf.ehcache.distribution.RMIBootstrapCacheLoaderFactory"/>
</cache>


<!--
Sample distributed cache named sampleDistributedCache2.
This cache replicates using specific properties.
It only replicates updates and does so synchronously via copy
-->
<cache name="sampleDistributedCache2"
       maxElementsInMemory="10"
       eternal="false"
```

```
       timeToIdleSeconds="100"
       timeToLiveSeconds="100"
       overflowToDisk="false">
    <cacheEventListenerFactory
            class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"
            properties="replicateAsynchronously=false, replicatePuts=false,
                        replicatePutsViaCopy=false, replicateUpdates=true,
                        replicateUpdatesViaCopy=true, replicateRemovals=false"/>
</cache>

<!--
Sample distributed cache named sampleDistributedCache3.
This cache replicates using defaults except that the asynchronous replication
interval is set to 200ms.
This one includes / and # which were illegal in ehcache 1.5.
-->
<cache name="sample/DistributedCache3"
       maxElementsInMemory="10"
       eternal="false"
       timeToIdleSeconds="100"
       timeToLiveSeconds="100"
       overflowToDisk="true">
    <cacheEventListenerFactory
            class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"
            properties="asynchronousReplicationIntervalMillis=200"/>
</cache>

<!--
Sample Terracotta clustered cache named sampleTerracottaCache.
This cache uses Terracotta to cluster the contents of the cache.
-->
<!--
<cache name="sampleTerracottaCache"
       maxElementsInMemory="1000"
       eternal="false"
       timeToIdleSeconds="3600"
       timeToLiveSeconds="1800"
       overflowToDisk="false">
    <terracotta/>
</cache>
-->



</ehcache>
```

7.4 Special System Properties

7.4.1 net.sf.ehcache.disabled

Setting this System Property to `true` disables caching in ehcache. If disabled no elements will be added to a cache. i.e. puts are silently discarded.

e.g. `java -Dnet.sf.ehcache.disabled=true` in the Java command line.

7.4.2 net.sf.ehcache.use.classic.lru

Set this System property to `true` to use the older LruMemoryStore implementation when LRU is selected as the eviction policy.

This is provided for ease of migration.

e.g. `java -Dnet.sf.ehcache.use.classic.lru=true` in the Java command line. Storage Options

Ehcache has two stores:

- a MemoryStore and
- a DiskStore

7.5 Memory Store

The `MemoryStore` is always enabled. It is not directly manipulated, but is a component of every cache.

- Suitable Element Types

  All Elements are suitable for placement in the MemoryStore.

  It has the following characteristics:

  - Safety

    Thread safe for use by multiple concurrent threads.

    Tested for memory leaks. See MemoryCacheTest#testMemoryLeak. This test passes for Ehcache but exploits a number of memory leaks in JCS. JCS will give an OutOfMemory error with a default 64M in 10 seconds.

  - Backed By JDK

    LinkedHashMap The `MemoryStore` for JDK1.4 and JDK 5 it is backed by an extended LinkedHashMap. This provides a combined linked list and a hash map, and is ideally suited for caching. Using this standard Java class simplifies the implementation of the memory cache. It directly supports obtaining the least recently used element.

  - Fast

    The memory store, being all in memory, is the fastest caching option.

7.5.1 Memory Use, Spooling and Expiry Strategy

All caches specify their maximum in-memory size, in terms of the number of elements, at configuration time.

When an element is added to a cache and it goes beyond its maximum memory size, an existing element is either deleted, if overflowToDisk is false, or evaluated for spooling to disk, if overflowToDisk is true. In the latter case, a check for expiry is carried out. If it is expired it is deleted; if not it is spooled. The eviction of an item from the memory store is based on the MemoryStoreEvictionPolicy setting specified in the configuration file.

memoryStoreEvictionPolicy is an optional attribute in ehcache.xml introduced since 1.2. Legal values are LRU (default), LFU and FIFO.

LRU, LFU and FIFO eviction policies are supported. LRU is the default, consistent with all earlier releases of ehcache.

- Least Recently Used (LRU) - Default

  The eldest element, is the Least Recently Used (LRU). The last used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.

- Less Frequently Used (LFU)

  For each get call on the element the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element with least number of hits, the Less Frequently Used element, is evicted.

- First In First Out (FIFO)

  Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

  For all the eviction policies there are also `putQuiet` and `getQuiet` methods which do not update the last used timestamp.

  When there is a `get` or a `getQuiet` on an element, it is checked for expiry. If expired, it is removed and null is returned.

  Note that at any point in time there will usually be some expired elements in the cache. Memory sizing of an application must always take into account the maximum size of each cache. There is a convenience method which can provide an estimate of the size in bytes of the `MemoryStore`. See calculateInMemorySize(). It returns the serialized size of the cache. Do not use this method in production. It is very slow. It is only meant to provide a rough estimate.

  The alternative would have been to have an expiry thread. This is a trade-off between lower memory use and short locking periods and cpu utilisation. The design is in favour of the latter. For those concerned with memory use, simply reduce the `maxElementsInMemory`.

7.6 DiskStore

The `DiskStore` provides a disk spooling facility.

7.6.1 `DiskStores` are Optional

The diskStore element in ehcache.xml is now optional (as of 1.5). If all caches use only `MemoryStores`, then there is no need to configure a diskStore. This simplifies configuration, and uses less threads. It is also good where where multiple CacheManagers are being used, and multiple disk store paths would need to be configured.

If one or more caches requires a DiskStore, and none is configured, java.io.tmpdir will be used and a warning message will be logged to encourage explicity configuration of the diskStore path.

Turning off disk stores

To turn off disk store path creation, comment out the diskStore element in ehcache.xml.

The `ehcache-failsafe.xml` configuration uses a disk store. This will remain the case so as to not affect existing Ehcache deployments. So, if you do not wish to use a disk store make sure you specify your own ehcache.xml and comment out the diskStore element.

7.6.2 Suitable Element Types

Only `Elements` which are `Serializable` can be placed in the DiskStore. Any non serializable `Elements` which attempt to overflow to the `DiskStore` will be removed instead, and a WARNING level log message emitted.

7.6.3 Storage

Files

The disk store creates a data file for each cache on startup called "*cache_name*.data", and, if the `DiskStore` is configured to be persistent, an index file called "**cache name**.index" on flushing of the `DiskStore` either explicitly using `Cache.flush` or on `CacheManager` shutdown.

Storage Location

Files are created in the directory specified by the diskStore configuration element. The diskStore configuration for the ehcache-failsafe.xml and bundled sample configuration file ehcache.xml is "java.io.tmpdir", which causes files to be created in the system's temporary directory.

diskStore Element

The `diskStore` element is has one attribute called `path`. --- *diskStore path="java.io.tmpdir"/* --- Legal values for the path attibute are legal file system paths. e.g.for Unix

```
/home/application/cache
```

The following system properties are also legal, in which case they are translated:

- user.home - User's home directory
- user.dir - User's current working directory
- java.io.tmpdir - Default temp file path
- ehcache.disk.store.di?r - A system property you would normally specify on the command line e.g. java -Dehcache.disk.store.dir=/u01/myapp/diskdir ...

  Subdirectories can be specified below the system property e.g.

  ```
  java.io.tmpdir/one

  becomes, on a Unix system,

  /tmp/one
  ```

7.6.4 Expiry

One thread per cache is used to remove expired elements. The optional attribute `diskExpiryThreadIntervalSeconds` sets the interval between runs of the expiry thread. Warning: setting this to a low value is not recommended.
It can cause excessive `DiskStore` locking and high cpu utilisation. The default value is 120 seconds.

7.6.5 Eviction

If the `maxElementsOnDisk` attribute is set, elements will be evicted from the `DiskStore` when it exceeds that amount. The LFU algorithm is used for these evictions. It is not configurable to use another algorithm.

7.6.6 Serializable Objects

Only Serializable objects can be stored in a `DiskStore`. A NotSerializableException will be thrown if the object is not serializable.

7.6.7 Safety

`DiskStore`s are thread safe.

7.6.8 Persistence

`DiskStore` persistence is controlled by the diskPersistent configuration element. If false or omitted, `DiskStore`s will not persist between `CacheManager` restarts. The data file for each cache will be deleted, if it exists, both on shutdown and startup. No data from a previous instance `CacheManager` is available.

If diskPersistent is true, the data file, and an index file, are saved. Cache Elements are available to a new `CacheManager`. This `CacheManager` may be in the same VM instance, or a new one.

The data file is updated continuously during operation of the Disk Store if `overflowToDisk` is true. Otherwise it is not updated until either `cache.flush()` is called or the cache is disposed.

In all cases the index file is only written when dispose is called on the `DiskStore`. This happens when the CacheManager is shut down, a Cache is disposed, or the VM is being shut down. It is recommended that the CacheManager shutdown() method be used. See Virtual Machine Shutdown Considerations for guidance on how to safely shut the Virtual Machine down.

When a `DiskStore` is persisted, the following steps take place:

- Any non-expired Elements of the `MemoryStore` are flushed to the DiskStore
- Elements awaiting spooling are spooled to the data file
- The free list and element list are serialized to the index file

On startup the following steps take place:

- An attempt is made to read the index file. If it does not exist or cannot be read successfully, due to disk corruption, upgrade of ehcache, change in JDK version etc, then the data file is deleted and the `DiskStore` starts with no Elements in it.
- If the index file is read successfully, the free list and element list are loaded into memory. Once this is done, the index file contents are removed. This way, if there is a dirty shutdown, when restarted, Ehcache will delete the dirt index and data files.
- The `DiskStore` starts. All data is available.
- The expiry thread starts. It will delete Elements which have expired.

These actions favour safety over persistence. Ehcache is a cache, not a database. If a file gets dirty, all data is deleted. Once started there is further checking for corruption. When a get is done, if the Element cannot be successfully derserialized, it is deleted, and null is returned. These measures prevent corrupt and inconsistent data being returned.

- Fragmentation

  Expiring an element frees its space on the file. This space is available for reuse by new elements. The element is also removed from the in-memory index of elements.

- Speed

  Spool requests are placed in-memory and then asynchronously written to disk. There is one thread per cache. An in-memory index of elements on disk is maintained to quickly resolve whether a key exists on disk, and if so to seek it and read it.

- Serialization

  Writes to and from the disk use ObjectInputStream and the Java serialization mechanism. This is not required for the MemoryStore. As a result the DiskStore can never be as fast as the MemoryStore.

  Serialization speed is affected by the size of the objects being serialized and their type. It has been found in the ElementTest test that:

  – The serialization time for a Java object being a large Map of String arrays was 126ms, where the a serialized size was 349,225 bytes.
  – The serialization time for a byte[] was 7ms, where the serialized size was 310,232 bytes

  Byte arrays are 20 times faster to serialize. Make use of byte arrays to increase DiskStore performance.

- RAMFS

  One option to speed up disk stores is to use a RAM file system. On some operating systems there are a plethora of file systems to choose from. For example, the Disk Cache has been successfully used with Linux' RAMFS file system. This file system simply consists of memory. Linux presents it as a file system. The Disk Cache treats it like a normal disk - it is just way faster. With this type of file system, object serialization becomes the limiting factor to performance.

  – Operation of a Cache where overflowToDisk is false and diskPersistent is true

    In this configuration case, the disk will be written on `flush` or `shutdown`.

    The next time the cache is started, the disk store will initialise but will not permit overflow from the MemoryStore. In all other respects it acts like a normal disk store.

    In practice this means that persistent in-memory cache will start up with all of its elements on disk. As gets cause cache hits, they will be loaded up into the `MemoryStore`. The oher thing that may happen is that the elements will expire, in which case the `DiskStore` expiry thread will reap them, (or they will get removed on a get if they are expired).

    So, the Ehcache design does not load them all into memory on start up, but lazily loads them as required.

Chapter 8

Cache Eviction Algorithms

8.1 Eviction

A cache eviction algorithm is a way of deciding which `Element` to evict when the cache is full.

In Ehcache the `MemoryStore` has a fixed limited size set by `maxElementsInMemory`. When the store gets full, `elements` are evicted. The eviction algorithms in Ehcache determines which elements is evicted. The default is LRU.

What happens on eviction depends on the cache configuration. If a `DiskStore` is configured, the evicted element will overflow to disk, otherwise it will be removed.

The `DiskStore` size by default is unbounded. But a maximum size can be set using the `maxElementsOnDisk` cache attribute. If the `DiskStore` is full, then adding an element will cause one to be evicted. The `DiskStore` eviction algorithm is not configurable. It uses LFU.

8.1.1 Supported `MemoryStore` Eviction Algorithms

The idea here is, given a limit on the number of items to cache, how to choose the thing to evict that gives the *best* result.

In 1966 Laszlo Belady showed that the most efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This it a theoretical result that is unimplementable without domain knowledge. The Least Recently Used ("LRU") algorithm is often used as a proxy. It works pretty well because of the locality of reference phenonemon. As a result, LRU is the default eviction algorithm in Ehcache, as it is in most caches.

Ehcache users may sometimes have a good domain knowledge. Accordingly, Ehcache provides three eviction algorithms to choose from for the `MemoryStore`.

8.1.2 `MemoryStore` Eviction Algorithms

The `MemoryStore` supports three eviction algorithms: LRU, LFU and FIFO.

The default is LRU.

Least Recently Used (LRU)

The eldest element, is the Least Recently Used (LRU). The last used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.

Less Frequently Used (LFU)

For each get call on the element the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached) the element with least number of hits, the Less Frequently Used element, is evicted.

If cache element use follows a pareto distribution, this algorithm may give better results than LRU.

LFU is an algorithm unique to Ehcache. It takes a random sample of the Elements and evicts the smallest. Using the sample size of 30 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 99.99% of the time.

First In First Out (FIFO)

Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

This algorithm is used if the use of an element makes it less likely to be used in the future. An example here would be an authentication cache.

8.1.3 `DiskStore` Eviction Algorithms

The `DiskStore` uses the Less Frequently Used algorithm to evict an element when it is full.

Chapter 9

Code Samples

This page shows some of the more common code samples to get you started. Code samples for each feature are in the relevant chapters.

- Using the CacheManager
  - Singleton versus Instance
  - Ways of loading Cache Configuration
  - Adding and Removing Caches Programmatically
  - Shutdown the CacheManager
- Using Caches
  - Obtaining a reference to a Cache
  - Performing CRUD operations
  - Disk Persistence on demand
  - Obtaining Cache Sizes
  - Obtaining Statistics of Cache Hits and Misses
- Programmatically Creating Caches
  - Creating a new cache from defaults
  - Creating a new cache with custom parameters
- Registering CacheStatistics in an MBeanServer
- JCache Examples
- Terracotta Clustering Examples
- Cache Server Examples
- Browse the JUnit Tests

9.1 Using the CacheManager

All usages of Ehcache start with the creation of a CacheManager.

9.1.1 Singleton versus Instance

As of ehcache-1.2, Ehcache CacheManagers can be created as either singletons (use the create factory method) or instances (use new).

Create a singleton CacheManager using defaults, then list caches.

```
CacheManager.create();
String[] cacheNames = CacheManager.getInstance().getCacheNames();
```

Create a CacheManager instance using defaults, then list caches.

```
 CacheManager manager = new CacheManager();
 String[] cacheNames = manager.getCacheNames();
```

Create two CacheManagers, each with a different configuration, and list the caches in each.

```
 CacheManager manager1 = new CacheManager("src/config/ehcache1.xml");
 CacheManager manager2 = new CacheManager("src/config/ehcache2.xml");
 String[] cacheNamesForManager1 = manager1.getCacheNames();
 String[] cacheNamesForManager2 = manager2.getCacheNames();
```

9.1.2 Ways of loading Cache Configuration

When the CacheManager is created it creates caches found in the configuration.

Create a CacheManager using defaults. Ehcache will look for ehcache.xml in the classpath.

```
 CacheManager manager = new CacheManager();
```

Create a CacheManager specifying the path of a configuration file.

```
CacheManager manager = new CacheManager("src/config/ehcache.xml");
```

Create a CacheManager from a configuration resource in the classpath.

```
URL url = getClass().getResource("/anotherconfigurationname.xml");
CacheManager manager = new CacheManager(url);
```

Create a CacheManager from a configuration in an InputStream.

```
InputStream fis = new FileInputStream(new File("src/config/ehcache.xml").getAbsolutePath());
try {
    CacheManager manager = new CacheManager(fis);
} finally {
    fis.close();
}
```

9.1.3 Adding and Removing Caches Programmatically

You are not just stuck with the caches that were placed in the configuration. You can create and remove them programmatically.

Add a cache using defaults, then use it. The following example creates a cache called *testCache*, which will be configured using defaultCache from the configuration.

```
CacheManager singletonManager = CacheManager.create();
singletonManager.addCache("testCache");
Cache test = singletonManager.getCache("testCache");
```

Create a Cache and add it to the CacheManager, then use it. Note that Caches are not usable until they have been added to a CacheManager.

```
CacheManager singletonManager = CacheManager.create();
Cache memoryOnlyCache = new Cache("testCache", 5000, false, false, 5, 2);
manager.addCache(memoryOnlyCache);
Cache test = singletonManager.getCache("testCache");
```

See Cache#Cache(...) for the full parameters for a new Cache:

Remove cache called sampleCache1

```
CacheManager singletonManager = CacheManager.create();
singletonManager.removeCache("sampleCache1");
```

9.1.4 Shutdown the CacheManager

Ehcache should be shutdown after use. It does have a shutdown hook, but it is best practice to shut it down in your code.

Shutdown the singleton CacheManager

```
CacheManager.getInstance().shutdown();
```

Shutdown a CacheManager instance, assuming you have a reference to the CacheManager called *manager*

```
manager.shutdown();
```

See the CacheManagerTest for more examples.

9.2 Using Caches

All of these examples refer to *manager*, which is a reference to a CacheManager, which has a cache in it called *sampleCache1*.

9.2.1 Obtaining a reference to a Cache

Obtain a Cache called "sampleCache1", which has been preconfigured in the configuration file

```
Cache cache = manager.getCache("sampleCache1");
```

9.2.2 Performing CRUD operations

Put an element into a cache

```
Cache cache = manager.getCache("sampleCache1");
Element element = new Element("key1", "value1");
cache.put(element);
```

Update an element in a cache. Even though `cache.put()` is used, Ehcache knows there is an existing element, and considers the put an update for the purpose of notifying cache listeners.

```
Cache cache = manager.getCache("sampleCache1");
cache.put(new Element("key1", "value1"));
//This updates the entry for "key1"
cache.put(new Element("key1", "value2"));
```

Get a Serializable value from an element in a cache with a key of "key1".

```
Cache cache = manager.getCache("sampleCache1");
Element element = cache.get("key1");
Serializable value = element.getValue();
```

Get a NonSerializable value from an element in a cache with a key of "key1".

```
Cache cache = manager.getCache("sampleCache1");
Element element = cache.get("key1");
Object value = element.getObjectValue();
```

Remove an element from a cache with a key of "key1".

```
Cache cache = manager.getCache("sampleCache1");
cache.remove("key1");
```

9.2.3 Disk Persistence on demand

sampleCache1 has a persistent diskStore. We wish to ensure that the data and index are written immediately.

```
Cache cache = manager.getCache("sampleCache1");
cache.flush();
```

9.2.4 Obtaining Cache Sizes

Get the number of elements currently in the `Cache`.

```
Cache cache = manager.getCache("sampleCache1");
int elementsInMemory = cache.getSize();
```

Get the number of elements currently in the `MemoryStore`.

```
Cache cache = manager.getCache("sampleCache1");
long elementsInMemory = cache.getMemoryStoreSize();
```

Get the number of elements currently in the `DiskStore`.

```
Cache cache = manager.getCache("sampleCache1");
long elementsInMemory = cache.getDiskStoreSize();
```

9.2.5 Obtaining Statistics of Cache Hits and Misses

These methods are useful for tuning cache configurations.

Get the number of times requested items were found in the cache. i.e. cache hits

```
Cache cache = manager.getCache("sampleCache1");
int hits = cache.getHitCount();
```

Get the number of times requested items were found in the `MemoryStore` of the cache.

```
Cache cache = manager.getCache("sampleCache1");
int hits = cache.getMemoryStoreHitCount();
```

Get the number of times requested items were found in the `DiskStore` of the cache.

```
Cache cache = manager.getCache("sampleCache1");
int hits = cache.getDiskStoreCount();
```

Get the number of times requested items were not found in the cache. i.e. cache misses.

```
Cache cache = manager.getCache("sampleCache1");
int hits = cache.getMissCountNotFound();
```

Get the number of times requested items were not found in the cache due to expiry of the elements.

```
Cache cache = manager.getCache("sampleCache1");
int hits = cache.getMissCountExpired();
```

These are just the most commonly used methods. See CacheTest for more examples. See Cache for the full API.

9.3 Creating a new cache from defaults

A new cache with a given name can be created from defaults very simply:

```
manager.addCache("cache name");
```

9.4 Creating a new cache with custom parameters

The configuration for a Cache can be specified programmatically in the Cache constructor:

```
public Cache(
    String name,
    int maxElementsInMemory,
    MemoryStoreEvictionPolicy memoryStoreEvictionPolicy,
    boolean overflowToDisk,
    boolean eternal,
    long timeToLiveSeconds,
    long timeToIdleSeconds,
    boolean diskPersistent,
    long diskExpiryThreadIntervalSeconds) {
    ...
   }
```

Here is an example which creates a cache called test.

```
//Create a CacheManager using defaults
CacheManager manager = CacheManager.create();

//Create a Cache specifying its configuration.

Cache testCache = new Cache("test", maxElements,
MemoryStoreEvictionPolicy.LFU, true, false, 60, 30, false, 0);
manager.addCache(cache);
```

Once the cache is created, add it to the list of caches managed by the CacheManager:

```
manager.addCache(testCache);
```

The cache is not usable until it has been added.

9.5 Registering CacheStatistics in an MBeanServer

This example shows how to register CacheStatistics in the JDK1.5 platform MBeanServer, which works with the JConsole management agent.

```
CacheManager manager = new CacheManager();
MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
ManagementService.registerMBeans(manager, mBeanServer, false, false, false, true);
```

9.6 Browse the JUnit Tests

Ehcache comes with a comprehensive JUnit test suite, which not only tests the code, but shows you how to use ehcache.

A link to browsable unit test source code for the major Ehcache classes is given per section. The unit tests are also in the src.zip in the Ehcache tarball.

9.7 JCache Examples

See the examples in the JCache Chapter.

9.8 Terracotta Example

See the fully worked examples in the Terracotta Clustering Chapter.

9.9 Cache Server Examples

See the examples in the Cache Server Chapter.

Chapter 10

Java Requirements and Dependencies

10.1 Java Requirements

Current Ehcache releases require Java 1.5 and 1.6 at runtime.

Ehcache 1.5 requires Java 1.4.

The Ehcache DX product which provides management and monitoring will work with Ehcache 1.2.3 but only for Java 1.5 or higher.

10.2 Mandatory Dependencies

Ehcache core 1.6 through to 1.7.0 has no dependencies.

Ehcache core 1.7.1 depends on SLF4J (www.slf4j.org), an increasingly commonly used logging framework which provides a choice of concrete logging implementation.

Other modules have dependencies as specified in their maven poms.

Chapter 11

Logging

11.1 Java Util Logging

As of 1.7.1, Ehcache uses the the slf4j logging facade. Plug in your own logging framework.

11.2 Recommended Logging Levels

Ehcache seeks to trade off informing production support developers or important messages and cluttering the log.

ERROR ERROR messages should not occur in normal production and indicate that action should be taken.

WARN WARN messages generally indicate a configuration change should be made or an unusual event has occurred.

DEBUG DEBUG and TRACE messages are for development use. All DEBUG level statements are surrounded with a guard so that no performance cost is incurred unless the logging level is set.

Setting the logging level to DEBUG should provide more information on the source of any problems. Many logging systems enable a logging level change to be made without restarting the application.

Chapter 12

Remote Network debugging and monitoring for Distributed Caches

12.1 Introduction

The ehcache-1.x-remote-debugger.jar} can be used to debug replicated cache operations. When started with the same configuration as the cluster, it will join the cluster and then report cluster events for the cache of interest. By providing a window into the cluster it can help to identify the cause of cluster problems.

12.2 Packaging

From version 1.5 it is packaged in its own distribution tarball along with a maven module.

It is provided as an executable jar.

12.3 Limitations

This version of the debugger has been tested only with the default RMI based replication.

12.4 Usage

It is invoked as follows:

```
java -classpath [add your application jars here]
 -jar ehcache-debugger-1.5.0.jar ehcache.xml sampleCache1
    path_to_ehcache.xml [cacheToMonitor]
```

Note: Add to the classpath any libraries your project uses in addition to these above, otherwise RMI will attempt to load them remotely which requires specific security policy settings that surprise most people.

It takes one or two arguments:

- the first argument, which is mandatory, is the Ehcache configuration file e.g. app/config/ehcache.xml. This file should be configured to allow Ehcache to joing the cluster. Using one of the existing ehcache.xml files from the other nodes normally is sufficient.
- the second argument, which is optional, is the name of the cache e.g. distributedCache1

  If only the first argument is passed, it will print our a list of caches with replication configured from the configuration file, which are then available for monitoring.

  If the second argument is also provided, the debugger will monitor cache operations received for the given cache.

  This is done by registering a CacheEventListener which prints out each operation.

12.4.1 Output

When monitoring a cache it prints a list of caches with replication configured, prints notifications as they happen, and periodically prints the cache name, size and total events received. See sample output below which is produced when the RemoteDebuggerTest is run.

```
Caches with replication configured which are available for monitoring are:
sampleCache19 sampleCache20 sampleCache26 sampleCache42 sampleCache33
sampleCache51 sampleCache40 sampleCache32 sampleCache18 sampleCache25
sampleCache9 sampleCache15 sampleCache56 sampleCache31 sampleCache7
sampleCache12 sampleCache17 sampleCache45 sampleCache41 sampleCache30
sampleCache13 sampleCache46 sampleCache4 sampleCache36 sampleCache29
sampleCache50 sampleCache37 sampleCache49 sampleCache48 sampleCache38
sampleCache6 sampleCache2 sampleCache55 sampleCache16 sampleCache27
sampleCache11 sampleCache3 sampleCache54 sampleCache28 sampleCache10
sampleCache8 sampleCache47 sampleCache5 sampleCache53 sampleCache39
sampleCache23 sampleCache34 sampleCache22 sampleCache44 sampleCache52
sampleCache24 sampleCache35 sampleCache21 sampleCache43 sampleCache1
Monitoring cache: sampleCache1
Cache: sampleCache1 Notifications received: 0 Elements in cache: 0
Received put notification for element [ key = this is an id, value=this is
a value, version=1, hitCount=0, CreationTime = 1210656023456,
LastAccessTime = 0 ]
Received update notification for element [ key = this is an id, value=this
is a value, version=1210656025351, hitCount=0, CreationTime =
1210656024458, LastAccessTime = 0 ]
Cache: sampleCache1 Notifications received: 2 Elements in cache: 1
Received remove notification for element this is an id
Received removeAll notification.
```

12.4.2 Providing more Detailed Logging

If you see nothing happening, but cache operations should be going through, enable trace (LOG4J) or finest (JDK) level logging on *code*net.sf.ehcache.distribution*/code* in the logging configuration being used by the debugger. A large volume of log messages will appear. The normal problem is that the CacheManager has not joined the cluster. Look for the list of cache peers.

12.4.3 Yes, but I still have a cluster problem

Check the FAQ where a lot of commonly reported errors and their solutions are provided. Beyond that, post to the forums or mailing list or contact Ehcache for support.

Chapter 13

Garbage Collection

Applications which use Ehcache can be expected to have larger heaps. Some Ehcache applications have heap sizes greater than 6GB.

Ehcache works well at this scale. However large heaps or long held object, which is what a cache is, can place strain on the default Garbage Collector.

Note. The following documentation relates to Sun JDK 1.5.

13.1 Detecting Garbage Collection Problems

A full garbage collection event pauses all threads in the JVM. Nothing happens during the pause. If this pause takes more than a few seconds it will become noticeable.

The clearest way to see if this is happening is to run `jstat`. The following command will produce a log of garbage collection statistics, updated each ten seconds.

```
    jstat -gcutil <pid> 10 1000000
```

The thing to watch for is the Full Garbage Collection Time. The difference between the total time for each reading is the time the system spends time paused. If there is a jump more than a few seconds this will not be acceptable in most application contexts.

13.2 Garbage Collection Tuning

The Sun core garbage collection team has offered the following tuning suggestion for virtual machiens with large heaps using caching:

```
java ... -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC
    -XX:NewSize=<1/4 of total heap size> -XX:SurvivorRatio=16
```

The reasoning for each setting is as follows:

- -XX:+DisableExplicitGC - some libs call System.gc(). This is usually a bad idea and could explain some of what we saw.
- -XX:+UseConcMarkSweepGC - use the low pause collector
- -XX:NewSize=*1/4 of total heap size* -XX:SurvivorRatio=16

13.3 Distributed Caching Garbage Collection Tuning

Some users have reported that enabling distributed caching causes a full GC each minute. This is an issue with RMI generally, which can be worked around by increasing the interval for garbage collection. The effect that RMI is having is similar to a user application calling `System.gc()` each minute. In the settings above this is disabled, but it does not disable the full GC initiated by RMI.

The default in JDK6 was increased to 1 hour. The following system properties control the interval.

```
-Dsun.rmi.dgc.client.gcInterval=60000
-Dsun.rmi.dgc.server.gcInterval=60000
```

See http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4403367 for the bug report and detailed instructions on workarounds.

Increase the interval as required in your application.

Chapter 14

JMX Management and Monitoring

14.1 Terracotta Monitoring Products

An extensive new monitoring product, available in Ehcache DX, provides a monitoring server with probes supporting Ehcache-1.2.3 and higher for standalone and clustered Ehcache. It comes with a web console and a RESTful API for operations integration.

See the Ehcache DX documentation for more information.

When using Ehcache 1.7 with Terracotta clustering, the Terracotta Developer Console shows statistics for ehcache.

14.2 JMX Overview

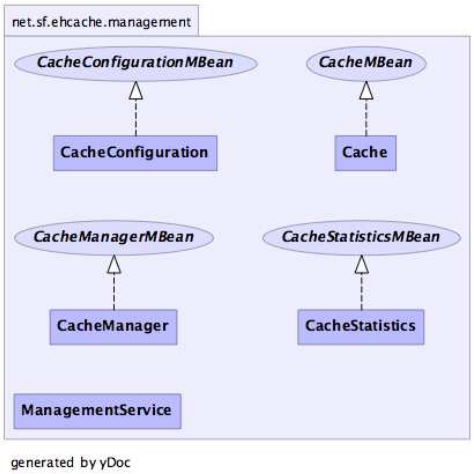
JMX, part of JDK1.5, and available as a download for 1.4, creates a standard way of instrumenting classes and making them available to a management and monitoring infrastructure.

The `net.sf.ehcache.management` package contains MBeans and a `ManagementService` for JMX management of ehcache. It is in a separate package so that JMX libraries are only required if you wish to use it - there is no leakage of JMX dependencies into the core Ehcache package.

This implementation attempts to follow Sun's JMX best practices. See http://java.sun.com/javase/technologies/core/mntr-mgmt/ javamanagement/best-practices.jsp.

Use `net.sf.ehcache.management.ManagementService.registerMBeans(...)` static method to register a selection of MBeans to the MBeanServer provided to the method.

If you wish to monitor Ehcache but not use JMX, just use the existing public methods on `Cache` and `CacheStatistics`.



The Management Package

14.3 MBeans

Ehcache uses Standard MBeans. MBeans are available for the following:

- CacheManager
- Cache
- CacheConfiguration
- CacheStatistics

  All MBean attributes are available to a local MBeanServer. The CacheManager MBean allows traversal to its collection of Cache MBeans. Each Cache MBean likewise allows traversal to its CacheConfiguration MBean and its CacheStatistics MBean.

14.4 JMX Remoting

The JMX Remote API allows connection from a remote JMX Agent to an MBeanServer via an `MBeanServerConnection`.

Only `Serializable` attributes are available remotely. The following Ehcache MBean attributes are available remotely:

- limited CacheManager attributes
- limited Cache attributes
- all CacheConfiguration attributes
- all CacheStatistics attributes

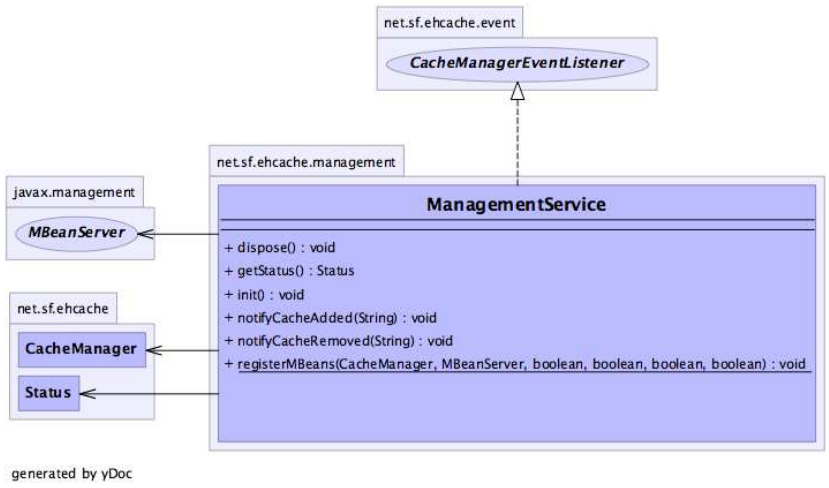
  Most attributes use built-in types. To access all attributes, you need to add ehcache.jar to the remote JMX client's classpath e.g. `jconsole -J-Djava.class.path=ehcache.jar`.

14.5 `ObjectName` naming scheme

- CacheManager - "net.sf.ehcache:type=CacheManager,name=*CacheManager*"
- Cache - "net.sf.ehcache:type=Cache,CacheManager=*cacheManagerName*,name=*cacheName*"
- CacheConfiguration - "net.sf.ehcache:type=CacheConfiguration,CacheManager=*cacheManagerName*,name=*cacheName*
- CacheStatistics - "net.sf.ehcache:type=CacheStatistics,CacheManager=*cacheManagerName*,name=*cacheName*"

14.6 The Management Service

The `ManagementService` class is the API entry point.



ManagementService

There is only one method, `ManagementService.registerMBeans` which is used to initiate JMX registration of an Ehcache CacheManager's instrumented MBeans.

The `ManagementService` is a `CacheManagerEventListener` and is therefore notified of any new Caches added or disposed and updates the MBeanServer appropriately.

Once initiated the MBeans remain registered in the MBeanServer until the CacheManager shuts down, at which time the MBeans are deregistered. This behaviour ensures correct behaviour in application servers where applications are deployed and undeployed.

```
/**
 * This method causes the selected monitoring options to be be registered
 * with the provided MBeanServer for caches in the given CacheManager.
 * <p/>
 * While registering the CacheManager enables traversal to all of the other
 *  items,
 * this requires programmatic traversal. The other options allow entry points closer
 * to an item of interest and are more accessible from JMX management tools like JConsole.
 * Moreover CacheManager and Cache are not serializable, so remote monitoring is not
 * possible * for CacheManager or Cache, while CacheStatistics and CacheConfiguration are.
 * Finally * CacheManager and Cache enable management operations to be performed.
 * <p/>
```

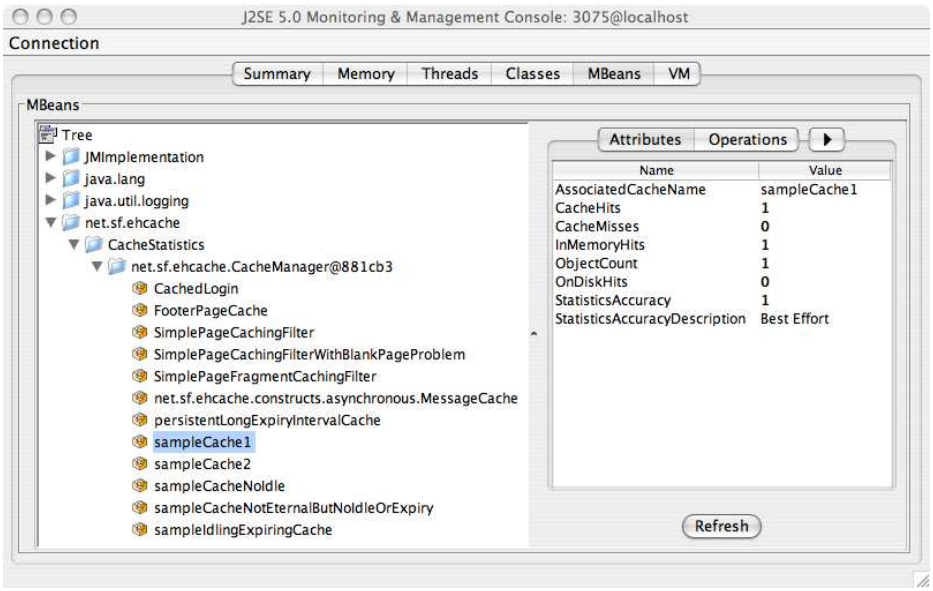
```
 * Once monitoring is enabled caches will automatically added and removed from the
 * MBeanServer * as they are added and disposed of from the CacheManager. When the
 * CacheManager itself * shutsdown all registered MBeans will be unregistered.
 *
 * @param cacheManager the CacheManager to listen to
 * @param mBeanServer the MBeanServer to register MBeans to
 * @param registerCacheManager Whether to register the CacheManager MBean
 * @param registerCaches Whether to register the Cache MBeans
 * @param registerCacheConfigurations Whether to register the CacheConfiguration MBeans
 * @param registerCacheStatistics Whether to register the CacheStatistics MBeans
 */
public static void registerMBeans(
        net.sf.ehcache.CacheManager cacheManager,
        MBeanServer mBeanServer,
        boolean registerCacheManager,
        boolean registerCaches,
        boolean registerCacheConfigurations,
        boolean registerCacheStatistics) throws CacheException {
```

14.7 JConsole Example

This example shows how to register CacheStatistics in the JDK1.5 platform MBeanServer, which works with the JConsole management agent.

```
    CacheManager manager = new CacheManager();
    MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
    ManagementService.registerMBeans(manager, mBeanServer, false, false, false, true);
```

CacheStatistics MBeans are then registered.



CacheStatistics MBeans in JConsole

14.8 JMX Tutorial

See http://weblogs.java.net/blog/maxpoon/archive/2007/06/extending_the_n_2.html for an online tutorial.

Chapter 15

Class loading and Class Loaders

Class loading within the plethora of environments Ehcache can be running is a somewhat vexed issue.

Since ehcache-1.2 all classloading is done in a standard way in one utility class: `ClassLoaderUtil`.

15.1 Plugin class loading

Ehcache allows plugins for events and distribution. These are loaded and created as follows:

```
/**
 * Creates a new class instance. Logs errors along the way. Classes are loaded using the
 * Ehcache standard classloader.
 *
 * @param className a fully qualified class name
 * @return null if the instance cannot be loaded
 */
public static Object createNewInstance(String className) throws CacheException {
    Class clazz;
    Object newInstance;
    try {
        clazz = Class.forName(className, true, getStandardClassLoader());
    } catch (ClassNotFoundException e) {
        //try fallback
        try {
            clazz = Class.forName(className, true, getFallbackClassLoader());
        } catch (ClassNotFoundException ex) {
            throw new CacheException("Unable to load class " + className +
                    ". Initial cause was " + e.getMessage(), e);
        }
    }

    try {
        newInstance = clazz.newInstance();
    } catch (IllegalAccessException e) {
        throw new CacheException("Unable to load class " + className +
                ". Initial cause was " + e.getMessage(), e);
    } catch (InstantiationException e) {
        throw new CacheException("Unable to load class " + className +
                ". Initial cause was " + e.getMessage(), e);
    }
    return newInstance;
```

```
}

/**
 * Gets the <code>ClassLoader</code> that all classes in ehcache, and extensions, should
 * use for classloading. All ClassLoading in Ehcache should use this one. This is the only
 * thing that seems to work for all of the class loading situations found in the wild.
 * @return the thread context class loader.
 */
public static ClassLoader getStandardClassLoader() {
    return Thread.currentThread().getContextClassLoader();
}

/**
 * Gets a fallback <code>ClassLoader</code> that all classes in ehcache, and extensions,
 * should use for classloading. This is used if the context class loader does not work.
 * @return the <code>ClassLoaderUtil.class.getClassLoader();</code>
 */
public static ClassLoader getFallbackClassLoader() {
    return ClassLoaderUtil.class.getClassLoader();
}
```

If this does not work for some reason a CacheException is thrown with a detailed error message.

15.2 Loading of ehcache.xml resources

If the configuration is otherwise unspecified, Ehcache looks for a configuration in the following order:

- Thread.currentThread().getContextClassLoader().getResource("/ehcache.xml")
- ConfigurationFactory.class.getResource("/ehcache.xml")
- ConfigurationFactory.class.getResource("/ehcache-failsafe.xml")

Ehcache uses the first configuration found.

Note the use of "/ehcache.xml" which requires that ehcache.xml be placed at the root of the classpath, i.e. not in any package.

Chapter 16

Performance Considerations

16.1 DiskStore

Ehcache comes with a `MemoryStore` and a `DiskStore`. The `MemoryStore` is approximately an order of magnitude faster than the `DiskStore`. The reason is that the `DiskStore` incurs the following extra overhead:

- Serialization of the key and value
- Eviction from the `MemoryStore` using an eviction algorithm
- Reading from disk

Note that writing to disk is not a synchronous performance overhead because it is handled by a separate thread.

A Cache should alway have its `maximumSize` attribute set to 1 or higher. A Cache with a maximum size of 1 has twice the performance of a disk only cache, i.e. one where the `maximumSize` is set to 0. For this reason a warning will be issued if a Cache is created with a 0 `maximumSize`.

16.2 Replication

The asynchronous replicator is the highest performance. There are two different effects:

- Because it is asynchronous the caller returns immediately
- The messages are placed in a queue. As the queue is processed, multiple messages are sent in one RMI call, dramatically accelerating replication performance.

Chapter 17

Cache Decorators

Ehcache 1.2 introduced the Ehcache interface, of which Cache is an implementation. It is possible and encouraged to create Ehcache decorators that are backed by a Cache instance, implement Ehcache and provide extra functionality.

The Decorator pattern is one of the the well known Gang of Four patterns.

17.1 Creating a Decorator

Cache decorators are created as follows:

```
BlockingCache newBlockingCache = new BlockingCache(cache);
```

The class must implement Ehcache.

17.2 Accessing the decorated cache

Having created a decorator it is generally useful to put it in a place where multiple threads may access it. This can be achieved in multiple ways.

17.2.1 Using CacheManager to access decorated caches

A built-in way is to replace the Cache in CacheManager with the decorated one. This is achieved as in the following example:

```
cacheManager.replaceCacheWithDecoratedCache(cache, newBlockingCache);
```

The `CacheManager` `replaceCacheWithDecoratedCache` method requires that the decorated cache be built from the underlying cache from the same name.

Note that any overwritten Ehcache methods will take on new behaviours without casting, as per the normal rules of Java. Casting is only required for new methods that the decorator introduces.

Any calls to get the cache out of the CacheManager now return the decorated one.

A word of caution. This method should be called in an appropriately synchronized init style method before multiple threads attempt to use it. All threads must be referencing the same decorated cache. An example of a suitable init method is found in `CachingFilter`:

```
/**
 * The cache holding the web pages. Ensure that all threads for a given cache name
 * are using the same instance of this.
 */
private BlockingCache blockingCache;

/**
 * Initialises blockingCache to use
 *
 * @throws CacheException The most likely cause is that a cache has not been
 *                        configured in Ehcache's configuration file ehcache.xml for the
 *                        filter name
 */
public void doInit() throws CacheException {
    synchronized (this.getClass()) {
        if (blockingCache == null) {
            final String cacheName = getCacheName();
            Ehcache cache = getCacheManager().getEhcache(cacheName);
            if (!(cache instanceof BlockingCache)) {
                //decorate and substitute
                BlockingCache newBlockingCache = new BlockingCache(cache);
                getCacheManager().replaceCacheWithDecoratedCache(cache, newBlockingCache);
            }
            blockingCache = (BlockingCache) getCacheManager().getEhcache(getCacheName());
        }
    }
}
```

```
Ehcache blockingCache = singletonManager.getEhcache("sampleCache1");
```

The returned cache will exhibit the decorations.

17.3 Built-in Decorators

17.3.1 BlockingCache

A blocking decorator for an Ehcache, backed by a @link Ehcache.

It allows concurrent read access to elements already in the cache. If the element is null, other reads will block until an element with the same key is put into the cache.

This is useful for constructing read-through or self-populating caches.

BlockingCache is used by `CachingFilter`.

BlockingCache

17.3.2 SelfPopulatingCache

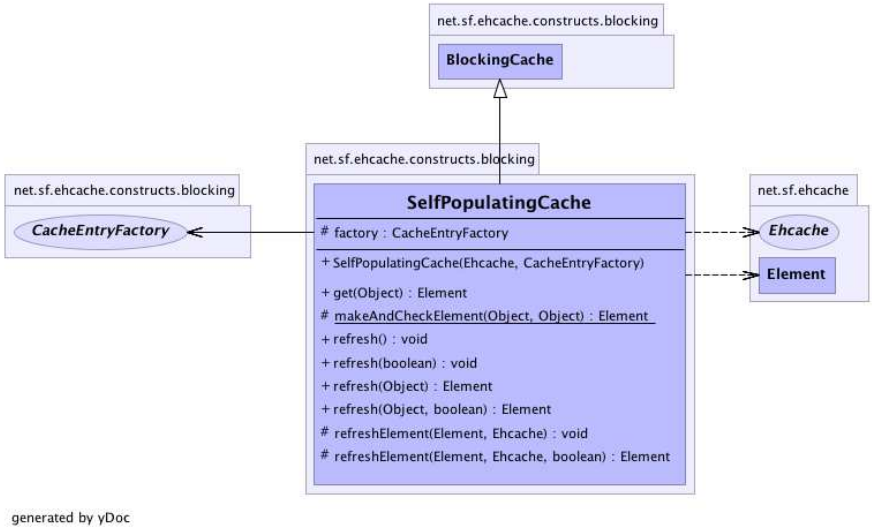
A selfpopulating decorator for @link Ehcache that creates entries on demand.

Clients of the cache simply call it without needing knowledge of whether the entry exists in the cache. If null the entry is created.

The cache is designed to be refreshed. Refreshes operate on the backing cache, and do not degrade performance of get calls.

SelfPopulatingCache extends BlockingCache. Multiple threads attempting to access a null element will block until the first thread completes. If refresh is being called the threads do not block - they return the stale data.

This is very useful for engineering highly scalable systems.



SelfPopulatingCache

17.3.3 Caches with Exception Handling

These are decorated. See Cache Exception Handlers for full details.

Chapter 18

Shutting Down Ehcache

If you are using persistent disk stores, or distributed caching, care should be taken to shutdown ehcache.

Note that Hibernate automatically shuts down its Ehcache `CacheManager`.

The recommended way to shutdown the Ehcache is:

- to call `CacheManager.shutdown()`
- in a web app, register the Ehcache `ShutdownListener`

  Though not recommended, Ehcache also lets you register a JVM shutdown hook.

18.1 ServletContextListener

Ehcache proivdes a ServletContextListener that shutsdown CacheManager. Use this when you want to shutdown Ehcache automatically when the web application is shutdown.

To receive notification events, this class must be configured in the deployment descriptor for the web application.

To do so, add the following to web.xml in your web application:

```
<listener>
     <listener-class>net.sf.ehcache.constructs.web.ShutdownListener</listener-class>
</listener>
```

18.2 The Shutdown Hook

Ehcache CacheManager can optionally register a shutdown hook.

To do so, set the system property `net.sf.ehcache.enableShutdownHook=true`.

This will shutdown the CacheManager when it detects the Virtual Machine shutting down and it is not already shut down.

18.2.1 When to use the shutdown hook

Use the shutdown hook where:

- you need guaranteed orderly shutdown, when for example using persistent disk stores, or distributed caching.
- CacheManager is not already being shutdown by a framework you are using or by your application.

  Having said that, shutdown hooks are inherently dangerous. The JVM is shutting down, so sometimes things that can never be null are. Ehcache guards against as many of these as it can, but the shutdown hook should be the last option to use.

18.2.2 What the shutdown hook does

The shutdown hook is on CacheManager. It simply calls the shutdown method.

The sequence of events is:

- call dispose for each registered CacheManager event listener
- call dispose for each Cache.

  Each Cache will:

  - shutdown the MemoryStore. The MemoryStore will flush to the DiskStore
  - shutdown the DiskStore. If the DiskStore is persistent, it will write the entries and index to disk.
  - shutdown each registered CacheEventListener
  - set the Cache status to shutdown, preventing any further operations on it.
- set the CacheManager status to shutdown, preventing any further operations on it

18.2.3 When a shutdown hook will run, and when it will not

The shutdown hook runs when:

- a program exists normally. e.g. System.exit() is called, or the last non-daemon thread exits
- the Virtual Machine is terminated. e.g. CTRL-C. This corresponds to `kill -SIGTERM pid` or `kill -15 pid` on Unix systems.

The shutdown hook will not run when:

- the Virtual Machine aborts
- A SIGKILL signal is sent to the Virtual Machine process on Unix systems. e.g. `kill -SIGKILL pid` or `kill -9 pid`
- A `TerminateProcess` call is sent to the process on Windows systems.

18.3 Dirty Shutdown

If Ehcache is shutdown dirty then any persistent disk stores will be corrupted. They will be deleted, with a log message, on the next startup.

Replications waiting to happen to other nodes in a distributed cache will also not get written.

Chapter 19

Web Caching

Ehcache provides a set of general purpose web caching filters in the `ehcache-web` module.

Using these can make an amazing difference to web application performance. A typical server can deliver 5000+ pages per second from the page cache. With built-in gzipping, storage and network transmission is highly efficient. Cache pages and fragments make excellent candidates for `DiskStore` storage, because the object graphs are simple and the largest part is already a `byte[]`.

19.1 SimplePageCachingFilter

This is a simple caching filter suitable for caching compressable HTTP responses such as HTML, XML or JSON.

It uses a Singleton CacheManager created with the default factory method. Override to use a different CacheManager

It is suitable for:

- complete responses i.e. not fragments.
- A content type suitable for gzipping. e.g. text or text/html

  For fragments see the SimplePageFragmentCachingFilter.

19.2 Keys

Pages are cached based on their key. The key for this cache is the URI followed by the query string. An example is `/admin/SomePage.jsp?id=1234&name=Beagle`.

This key technique is suitable for a wide range of uses. It is independent of hostname and port number, so will work well in situations where there are multiple domains which get the same content, or where users access based on different port numbers.

A problem can occur with tracking software, where unique ids are inserted into request query strings. Because each request generates a unique key, there will never be a cache hit. For these situations it is better to parse the request parameters and override `calculateKey(javax.servlet.http.HttpServletRequest)` with an implementation that takes account of only the significant ones.

19.3 Configuring the cacheName

A cache entry in ehcache.xml should be configured with the name of the filter.

Names can be set using the init-param *code*cacheName*/code*, or by sub-classing this class and overriding the name.

19.4 Concurent Cache Misses

A cache miss will cause the filter chain, upstream of the caching filter to be processed. To avoid threads requesting the same key to do useless duplicate work, these threads block behind the first thread.

The thead timeout can be set to fail after a certain wait by setting the init-param *code*blockingTimeoutMillis*/code*. By default threads wait indefinitely. In the event upstream processing never returns, eventually the web server may get overwhelmed with connections it has not responded to. By setting a timeout, the waiting threads will only block for the set time, and then throw a @link net.sf.ehcache.constructs.blocking.LockTimeoutException. Under either scenario an upstream failure will still cause a failure.

19.5 Gzipping

Significant network efficiencies, and page loading speedups, can be gained by gzipping responses.

Whether a response can be gzipped depends on:

- Whether the user agent can accept GZIP encoding. This feature is part of HTTP1.1. If a browser accepts GZIP encoding it will advertise this by including in its HTTP header: All common browsers except IE 5.2 on Macintosh are capable of accepting gzip encoding. Most search engine robots do not accept gzip encoding.
- Whether the user agent has advertised its acceptance of gzip encoding. This is on a per request basis. If they will accept a gzip response to their request they must include the following in the HTTP request header:

  ```
  Accept-Encoding: gzip
  ```

  Responses are automatically gzipped and stored that way in the cache. For requests which do not accept gzip encoding the page is retrieved from the cache, ungzipped and returned to the user agent. The ungzipping is high performance.

19.6 Caching Headers

The `SimpleCachingHeadersPageCachingFilter` extends `SimplePageCachingFilter` to provide the HTTP cache headers: ETag, Last-Modified and Expires. It supports conditional GET.

Because browsers and other HTTP clients have the expiry information returned in the response headers, they do not even need to request the page again. Even once the local browser copy has expired, the browser will do a conditional GET.

So why would you ever want to use SimplePageCachingFilter, which does not set these headers? The answer is that in some caching scenarios you may wish to remove a page before its natural expiry. Consider a scenario where a web page shows dynamic data. Under Ehcache the Element can be removed at any time. However if a browser is holding expiry information, those browsers will have to wait until the expiry time before getting updated. The caching in this scenario is more about defraying server load rather than minimising browser calls.

19.7 Init-Params

The following init-params are supported:

- `cacheName` - the name in ehcache.xml used by the filter.
- `blockingTimeoutMillis` - the time, in milliseconds, to wait for the filter chain to return with a response on a cache miss. This is useful to fail fast in the event of an infrastructure failure.

19.8 Reentrance

Care should be taken not to define a filter chain such that the same `CachingFilter` class is reentered. The `CachingFilter` uses the `BlockingCache`. It blocks until the thread which did a get which results in a null does a put. If reentry happens a second get happens before the first put. The second get could wait indefinitely. This situation is monitored and if it happens, an IllegalStateException will be thrown.

19.9 SimplePageFragmentCachingFilter

The SimplePageFragmentCachingFilter does everyting that SimplePageCachingFilter does, except it never gzips, so the fragments can be combined. There is variant of this filter which sets browser caching headers, because that is only applicable to the entire page.

19.10 Example web.xml configuration

```
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
     http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd "
     version="2.5">

  <filter>
    <filter-name>CachePage1CachingFilter</filter-name>
    <filter-class>net.sf.ehcache.constructs.web.filter.SimplePageCachingFilter
    </filter-class>
    <init-param>
      <param-name>suppressStackTraces</param-name>
      <param-value>false</param-value>
    </init-param>
    <init-param>
      <param-name>cacheName</param-name>
      <param-value>CachePage1CachingFilter</param-value>
    </init-param>
  </filter>

  <filter>
    <filter-name>SimplePageFragmentCachingFilter</filter-name>
    <filter-class>net.sf.ehcache.constructs.web.filter.SimplePageFragmentCachingFilter
    </filter-class>
    <init-param>
      <param-name>suppressStackTraces</param-name>
      <param-value>false</param-value>
    </init-param>
```

```
    <init-param>
      <param-name>cacheName</param-name>
      <param-value>SimplePageFragmentCachingFilter</param-value>
    </init-param>
  </filter>

  <filter>
    <filter-name>SimpleCachingHeadersPageCachingFilter</filter-name>
    <filter-class>net.sf.ehcache.constructs.web.filter.SimpleCachingHeadersPageCachingFilter
    </filter-class>
    <init-param>
      <param-name>suppressStackTraces</param-name>
      <param-value>false</param-value>
    </init-param>
    <init-param>
      <param-name>cacheName</param-name>
      <param-value>CachedPage2Cache</param-value>
    </init-param>

  </filter>

  <!-- This is a filter chain. They are executed in the order below.
  Do not change the order. -->
  <filter-mapping>
    <filter-name>CachePage1CachingFilter</filter-name>
    <url-pattern>/CachedPage.jsp</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>FORWARD</dispatcher>
  </filter-mapping>

  <filter-mapping>
    <filter-name>SimplePageFragmentCachingFilter</filter-name>
    <url-pattern>/include/Footer.jsp</url-pattern>
  </filter-mapping>

  <filter-mapping>
    <filter-name>SimplePageFragmentCachingFilter</filter-name>
    <url-pattern>/fragment/CachedFragment.jsp</url-pattern>
  </filter-mapping>

  <filter-mapping>
    <filter-name>SimpleCachingHeadersPageCachingFilter</filter-name>
    <url-pattern>/CachedPage2.jsp</url-pattern>
  </filter-mapping>
```

An ehcache.xml configuration file, matching the above would then be:

```
<Ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="../../main/config/ehcache.xsd">

    <diskStore path="java.io.tmpdir"/>


    <defaultCache
            maxElementsInMemory="10"
            eternal="false"
```

```
            timeToIdleSeconds="5"
            timeToLiveSeconds="10"
            overflowToDisk="true"
            />


    <!-- Page and Page Fragment Caches -->

    <cache name="CachePage1CachingFilter"
           maxElementsInMemory="10"
           eternal="false"
           timeToIdleSeconds="10000"
           timeToLiveSeconds="10000"
           overflowToDisk="true">
    </cache>

    <cache name="CachedPage2Cache"
           maxElementsInMemory="10"
           eternal="false"
           timeToLiveSeconds="3600"
           overflowToDisk="true">
    </cache>

    <cache name="SimplePageFragmentCachingFilter"
           maxElementsInMemory="10"
           eternal="false"
           timeToIdleSeconds="10000"
           timeToLiveSeconds="10000"
           overflowToDisk="true">
    </cache>

    <cache name="SimpleCachingHeadersTimeoutPageCachingFilter"
           maxElementsInMemory="10"
           eternal="false"
           timeToIdleSeconds="10000"
           timeToLiveSeconds="10000"
           overflowToDisk="true">
    </cache>

</ehcache>
```

19.11 CachingFilter Exceptions

Additional exception types have been added to the Caching Filter.

19.11.1 FilterNonReentrantException

Thrown when it is detected that a caching filter's doFilter method is reentered by the same thread. Reentrant calls will block indefinitely because the first request has not yet unblocked the cache. Nasty.

19.11.2 AlreadyGzippedException

The web package performs gzipping operations. One cause of problems on web browsers is getting content that is double or triple gzipped. They will either get gobblydeegook or a blank page. This exception is thrown when a gzip is attempted on already gzipped content.

19.11.3 ResponseHeadersNotModifiableException

A gzip encoding header needs to be added for gzipped content. The HttpServletResponse#setHeader() method is used for that purpose. If the header had already been set, the new value normally overwrites the previous one. In some cases according to the servlet specification, setHeader silently fails. Two scenarios where this happens are:

- The response is committed.
- RequestDispatcher#include method caused the request. Distributed Caching with ehcache

  Ehcache provides a pluggable distributed caching mechanism. This enables for multiple CacheManagers and their caches in multiple JVMs to share data with each other.

19.12 Pluggable Mechanisms

Ehcache has a pluggable cache replication scheme which enables the addition of cache replication mechanisms.

The following distribution mechanisms are supported in Ehcache 1.7:

- Terracotta
- RMI
- JGroups
- JMS
- Cache Server

Each of the is covered in its own chapter.

19.13 The need for shared cache data

Many production applications are deployed in clusters. If each application maintains its own cache, then updates made to one cache will not appear in the others. A workaround for web based applications is to use sticky sessions, so that a user, having established a session on one server, stays on that server for the rest of the session. A workaround for transaction processing systems using Hibernate is to do a session.refresh on each persistent object as part of the save. session.refresh explicitly reloads the object from the database, ignoring any cache values.

19.14 Replicated Caches

One solution is to replicate data between the caches to keep them consistent, or coherent. Typical operations which Applicable operations include:

- put
- update (put which overwrites an existing entry)
- remove

Update supports updateViaCopy or updateViaInvalidate. The latter sends the a remove message out to the cache cluster, so that other caches remove the Element, thus preserving coherency. It is typically a lower cost option than a copy.

19.15 Using a Cache Server

Ehcache 1.5 supports the Ehcache Cache Server.

To achieve shared data, all JVMs read to and write from a Cache Server, which runs it in its own JVM.

To achieve redundancy, the Ehcache inside the Cache Server can be set up in its own cluster.

This technique will be expanded upon in Ehcache 1.6.

19.16 Notification Strategies

The best way of notifying of put and update depends on the nature of the cache.

If the Element is not available anywhere else then the Element itself should form the payload of the notification. An example is a cached web page. This notification strategy is called copy.

Where the cached data is available in a database, there are two choices. Copy as before, or invalidate the data. By invalidating the data, the application tied to the other cache instance will be forced to refresh its cache from the database, preserving cache coherency. Only the Element key needs to be passed over the network.

Ehcache supports notification through copy and invalidate, selectable per cache.

19.17 Potential Issues with Distributed Caching

19.17.1 Potential for Inconsistent Data

Timing scenarios, race conditions, delivery, reliability constraints and concurrent updates to the same cached data can cause inconsistency (and thus a lack of coherency) across the cache instances.

This potential exists within the Ehcache implementation. These issues are the same as what is seen when two completely separate systems are sharing a database; a common scenario.

Whether data inconsistency is a problem depends on the data and how it is used. For those times when it is important, Ehcache provides for synchronous delivery of puts and updates via invalidation. These are discussed below:

Synchronous Delivery

Delivery can be specified to be synchronous or asynchronous. Asynchronous delivery gives faster returns to operations on the local cache and is usually preferred. Synchronous delivery adds time to the local operation, however delivery of an update to all peers in the cluster happens before the cache operation returns.

Put and Update via Invalidation

The default is to update other caches by copying the new value to them. If the replicatePutsViaCopy property is set to false in the replication configuration, puts are made by removing the element in any other cache peers. If the replicateUpdatesViaCopy property is set to false in the replication configuration, updates are made by removing the element in any other cache peers.

This forces the applications using the cache peers to return to a canonical source for the data.

A similar effect can be obtained by setting the element TTL to a low value such as a second.

Note that these features impact cache performance and should not be used where the main purpose of a cache is performance boosting over coherency.

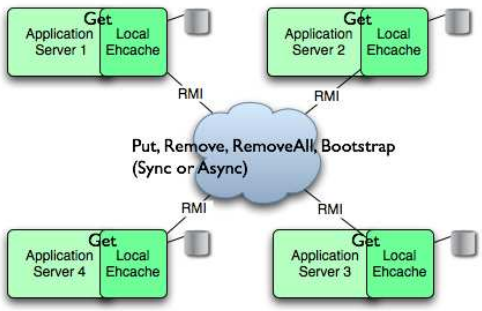
19.17.2 Use of Time To Idle

Time To Idle is inconsistent with distributed caching. Time-to-idle makes some entries live longer on some nodes than in others because of cache usage patterns. However, the cache entry "last touched" timestamp is not replicated across the distributed cache.

Do not use Time To Idle with distributed caching, unless you do not care about inconsistent data across nodes.

Chapter 20

RMI Distributed Caching



Since version 1.2, Ehcache has provided distributed caching using RMI.

An RMI implementation is desirable because:

- it itself is the default remoting mechanism in Java
- it is mature
- it allows tuning of TCP socket options
- Element keys and values for disk storage must already be Serializable, therefore directly transmittable over RMI without the need for conversion to a third format such as XML.
- it can be configured to pass through firewalls
- RMI had improvements added to it with each release of Java, which can then be taken advantage of.

While RMI is a point-to-point protocol, which can generate a lot of network traffic, Ehcache manages this through batching of communications for the asynchronous replicator.

To set up RMI distributed caching you need to configure the CacheManager with:

- a PeerProvider
- a CacheManagerPeerListener

  The for each cache that will operate distributed, you then need to add one of the RMI cacheEventListener types to propagate messages.

  You can also optionally configure a cache to bootstrap from other caches in the cluster.

20.1 Suitable Element Types

Only Serializable Elements are suitable for replication.

Some operations, such as remove, work off Element keys rather than the full Element itself. In this case the operation will be replicated provided the key is Serializable, even if the Element is not.

20.2 Configuring the Peer Provider

20.2.1 Peer Discovery

Ehcache has the notion of a group of caches acting as a distributed cache. Each of the caches is a peer to the others. There is no master cache. How do you know about the other caches that are in your cluster? This problem can be given the name Peer Discovery.

Ehcache provides two mechanisms for peer discovery, just like a car: manual and automatic.

To use one of the built-in peer discovery mechanisms specify the class attribute of `cacheManagerPeerProviderFactory` as `net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory` in the ehcache.xml configuration file.

20.2.2 Automatic Peer Discovery

Automatic discovery uses TCP multicast to establish and maintain a multicast group. It features minimal configuration and automatic addition to and deletion of members from the group. No a priori knowledge of the servers in the cluster is required. This is recommended as the default option.

Peers send heartbeats to the group once per second. If a peer has not been heard of for 5 seconds it is dropped from the group. If a new peer starts sending heartbeats it is admitted to the group.

Any cache within the configuration set up as replicated will be made available for discovery by other peers.

To set automatic peer discovery, specify the properties attribute of `cacheManagerPeerProviderFactory` as follows:

peerDiscovery=automatic multicastGroupAddress=multicast address |multicast host name multicastGroupPort=port timeToLive=0-255 (See below in common problems before setting this) hostName= the hostname or IP of the interface to be used for sending and receiving multicast packets (relevant to mulithomed hosts only)

Example

Suppose you have two servers in a cluster. You wish to distribute sampleCache11 and sampleCache12. The configuration required for each server is identical:

Configuration for server1 and server2

```
<cacheManagerPeerProviderFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=automatic, multicastGroupAddress=230.0.0.1,
multicastGroupPort=4446, timeToLive=32"/>
```

20.2.3 Manual Peer Discovery

Manual peer configuration requires the IP address and port of each listener to be known. Peers cannot be added or removed at runtime. Manual peer discovery is recommended where there are technical difficulties using multicast, such as a router between servers in a cluster that does not propagate multicast datagrams. You can also use it to set up one way replications of data, by having server2 know about server1 but not vice versa.

To set manual peer discovery, specify the properties attribute of `cacheManagerPeerProviderFactory` as follows: peerDiscovery=manual rmiUrls=//server:port/cacheName, ...

The rmiUrls is a list of the cache peers of the server being configured. Do not include the server being configured in the list.

Example

Suppose you have two servers in a cluster. You wish to distribute sampleCache11 and sampleCache12. Following is the configuration required for each server:

Configuration for server1

```
<cacheManagerPeerProviderFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=manual,
rmiUrls=//server2:40001/sampleCache11|//server2:40001/sampleCache12"/>
```

Configuration for server2

```
<cacheManagerPeerProviderFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"

properties="peerDiscovery=manual,
rmiUrls=//server1:40001/sampleCache11|//server1:40001/sampleCache12"/>
```

20.3 Configuring the CacheManagerPeerListener

A CacheManagerPeerListener listens for messages from peers to the current CacheManager.

You configure the CacheManagerPeerListener by specifiying a CacheManagerPeerListenerFactory which is used to create the CacheManagerPeerListener using the plugin mechanism.

The attributes of cacheManagerPeerListenerFactory are:

- class - a fully qualified factory class name * properties - comma separated properties having meaning only to the factory.

  Ehcache comes with a built-in RMI-based distribution system. The listener component is RMI-CacheManagerPeerListener which is configured using RMICacheManagerPeerListenerFactory. It is configured as per the following example:

  ```
  <cacheManagerPeerListenerFactory
  class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"

  properties="hostName=localhost, port=40001,
  socketTimeoutMillis=2000"/>
  ```

Valid properties are:

- hostName (optional) - the hostName of the host the listener is running on. Specify where the host is multihomed and you want to control the interface over which cluster messages are received.

  The hostname is checked for reachability during CacheManager initialisation.

  If the hostName is unreachable, the CacheManager will refuse to start and an CacheException will be thrown indicating connection was refused.

  If unspecified, the hostname will use `InetAddress.getLocalHost().getHostAddress()`, which corresponds to the default host network interface.

  Warning: Explicitly setting this to localhost refers to the local loopback of 127.0.0.1, which is not network visible and will cause no replications to be received from remote hosts. You should only use this setting when multiple CacheManagers are on the same machine.

- port (mandatory) - the port the listener listens on.

- socketTimeoutMillis (optional) - the number of seconds client sockets will wait when sending messages to this listener until they give up. By default this is 2000ms.

20.4 Configuring Cache Replicators

Each cache that will be distributed needs to set a cache event listener which then replicates messages to the other CacheManager peers. This is done by adding a cacheEventListenerFactory element to each cache's configuration.

```
<!-- Sample cache named sampleCache2. -->
<cache name="sampleCache2"
   maxElementsInMemory="10"
   eternal="false"
   timeToIdleSeconds="100"
   timeToLiveSeconds="100"
   overflowToDisk="false">
 <cacheEventListenerFactory
    class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"
    properties="replicateAsynchronously=true, replicatePuts=true, replicateUpdates=true,
    replicateUpdatesViaCopy=false, replicateRemovals=true "/>
</cache>
```

class - use net.sf.ehcache.distribution.RMICacheReplicatorFactory

The factory recognises the following properties:

- replicatePuts=true |false - whether new elements placed in a cache are replicated to others. Defaults to true.

- replicateUpdates=true |false - whether new elements which override an element already existing with the same key are replicated. Defaults to true.

- replicateRemovals=true - whether element removals are replicated. Defaults to true.

- replicateAsynchronously=true |false - whether replications are asyncrhonous (true) or synchronous (false). Defaults to true.

- replicateUpdatesViaCopy=true |false - whether the new elements are copied to other caches (true), or whether a remove message is sent. Defaults to true.

To reduce typing if you want default behaviour, which is replicate everything in asynchronous mode, you can leave off the `RMICacheReplicatorFactory` properties as per the following example:

```
<!-- Sample cache named sampleCache4. All missing RMICacheReplicatorFactory properties
     default to true -->
<cache name="sampleCache4"
       maxElementsInMemory="10"
       eternal="true"
       overflowToDisk="false"
       memoryStoreEvictionPolicy="LFU">
    <cacheEventListenerFactory
        class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>
</cache>
```

20.5 Configuring Bootstrap from a Cache Peer

When a peer comes up, it will be incoherent with other caches. When the bootstrap completes it will be partially coherent. Bootstrap gets the list of keys from a random peer, and then loads those in batches from random peers. If bootstrap fails then the Cache will not start (not like this right now). However if a distributed cache operation occurs which is then overwritten by bootstrap there is a chance that the cache could be inconsistent.

Here are some scenarios:

Delete overwritten by boostrap put --- Cache A keys with values: 1, 2, 3, 4, 5

Cache B starts bootstrap

Cache A removes key 2

Cache B removes key 2 and then bootstrap puts it back

Put overwritten by boostrap put --- Cache A keys with values: 1, 2, 3, 4, 5

Cache B starts bootstrap

Cache A updates the value of key 2

Cache B updates the value of key 2 and then bootstrap overwrites it with the old value

---

The solution is for bootstrap to get a list of keys and write them all before committing transactions.

This could cause synchronous transaction replicates to back up. To solve this problem, commits will be accepted, but not written to the cache until after bootstrap. Coherency is maintained because the cache is not available until bootstrap has completed and the transactions have been completed.

20.6 Full Example

Ehcache's own integration tests provide complete examples of RMI-based replication. The best example is the integration test for cache replication. You can see it online here: http://ehcache.org/xref-test/net/sf/ehcache/distribution/RMI

The test uses 5 ehcache.xml's representing 5 CacheManagers set up to distribute using RMI.

20.7 Common Problems

20.7.1 Tomcat on Windows

There is a bug in Tomcat and/or the JDK where any RMI listener will fail to start on Tomcat if the installation path has spaces in it. See http://archives.java.sun.com/cgi-bin/wa?A2=ind0205&L=rmi-users&P=797 and http://www.ontotext.com/kim/doc/sys-doc/faq-howto-bugs/known-bugs.html.

As the default on Windows is to install Tomcat in "Program Files", this issue will occur by default.

20.7.2 Multicast Blocking

The automatic peer discovery process relies on multicast. Multicast can be blocked by routers. Virtualisation technologies like Xen and VMWare may be blocking multicast. If so enable it. You may also need to turn it on in the configuration for your network interface card.

An easy way to tell if your multicast is getting through is to use the Ehcache remote debugger and watch for the heartbeat packets to arrive.

20.7.3 Multicast Not Propagating Far Enough or Propagating Too Far

You can control how far the multicast packets propagate by setting the badly misnamed time to live. Using the multicast IP protocol, the timeToLive value indicates the scope or range in which a packet may be forwarded. By convention:

```
0 is restricted to the same host
1 is restricted to the same subnet
32 is restricted to the same site
64 is restricted to the same region
128 is restricted to the same continent
255 is unrestricted
```

The default value in Java is 1, which propagates to the same subnet. Change the timeToLive property to restrict or expand propagation.

Chapter 21

Distributed Caching using JGroups

As of version 1.5, JGroups can be used as the underlying mechanism for the distributed operations in ehcache. JGroups offers a very flexible protocol stack, reliable unicast and multicast message transmission. On the down side JGroups can be complex to configure and some protocol stacks have dependencies on others.

To set up distributed caching using JGroups you need to configure a PeerProviderFactory of type JGroupsCacheManagerPeerProviderFactory which is done globally for a CacheManager For each cache that will operate distributed, you then need to add a cacheEventListenerFactory of type JGroupsCacheReplicatorFactory to propagate messages.

21.1 Suitable Element Types

Only Serializable Elements are suitable for replication.

Some operations, such as remove, work off Element keys rather than the full Element itself. In this case the operation will be replicated provided the key is Serializable, even if the Element is not.

21.2 Peer Discovery

If you use the UDP multicast stack there is no additional configuration. If you use a TCP stack you will need to specify the initial hosts in the cluster.

21.3 Configuration

There are two things to configure:

- The JGroupsCacheManagerPeerProviderFactory which is done once per CacheManager and therefore once per ehcache.xml file.
- The JGroupsCacheReplicatorFactory which is added to each cache's configuration.

  The main configuration happens in the JGroupsCacheManagerPeerProviderFactory connect subproperty. A connect property is passed directly to the JGroups channel and therefore all the protocol stacks and options available in JGroups can be set.

21.4 Example configuration using UDP Multicast

Suppose you have two servers in a cluster. You wish to distribute sampleCache11 and sampleCache12 and you wish to use UDP multicast as the underlying mechanism.

The configuration for server1 and server2 are identical and will look like this:

```
<cacheManagerPeerProviderFactory
     class="net.sf.ehcache.distribution.jgroups.JGroupsCacheManagerPeerProviderFactory"
     properties="connect=UDP(mcast_addr=231.12.21.132;mcast_port=45566;):PING:
     MERGE2:FD_SOCK:VERIFY_SUSPECT:pbcast.NAKACK:UNICAST:pbcast.STABLE:FRAG:pbcast.GMS"
     propertySeparator="::"
     />
```

21.5 Example configuration using TCP Unicast

The TCP protocol requires the IP address of all servers to be known. They are configured through the `TCPPING` protocol of Jgroups.

Suppose you have 2 servers host1 and host2, then the configuration is:

```
 <cacheManagerPeerProviderFactory
    class="net.sf.ehcache.distribution.jgroups.JGroupsCacheManagerPeerProviderFactory"
    properties="connect=TCP(start_port=7800):
        TCPPING(initial_hosts=host1[7800],host2[7800];port_range=10;timeout=3000;
        num_initial_members=3;up_thread=true;down_thread=true):
        VERIFY_SUSPECT(timeout=1500;down_thread=false;up_thread=false):
        pbcast.NAKACK(down_thread=true;up_thread=true;gc_lag=100;retransmit_timeout=3000):
        pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;shun=false;
        print_local_addr=false;down_thread=true;up_thread=true)"
    propertySeparator="::" />
```

21.6 Protocol considerations.

You should read the JGroups documentation to configure the protocols correctly.

See http://www.jgroups.org/javagroupsnew/docs/manual/html_single/index.html.

If using UDP you should at least configure PING, FD_SOCK (Failure detection), VERIFY_SUSPECT, pbcast.NAKACK (Message reliability), pbcast.STABLE (message garbage collection).

21.7 Configuring CacheReplicators

Each cache that will be distributed needs to set a cache event listener which then replicates messages to the other CacheManager peers. This is done by adding a cacheEventListenerFactory element to each cache's configuration. The properties are identical to the one used for RMI replication.

The listener factory *MUST* be of type `JGroupsCacheReplicatorFactory`.

```
<!-- Sample cache named sampleCache2. -->
<cache name="sampleCache2"
   maxElementsInMemory="10"
   eternal="false"
   timeToIdleSeconds="100"
```

```
   timeToLiveSeconds="100"
   overflowToDisk="false">
   <cacheEventListenerFactory
   class="net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory"
   properties="replicateAsynchronously=true, replicatePuts=true,
   replicateUpdates=true, replicateUpdatesViaCopy=false, replicateRemovals=true" />
</cache>
```

The configuration options are explained below:

class - use net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory

The factory recognises the following properties:

- replicatePuts=true |false - whether new elements placed in a cache are replicated to others. Defaults to true.
- replicateUpdates=true |false - whether new elements which override an element already existing with the same key are replicated. Defaults to true.
- replicateRemovals=true - whether element removals are replicated. Defaults to true.
- replicateAsynchronously=true |false - whether replications are asyncrhonous (true) or synchronous (false). Defaults to true.
- replicateUpdatesViaCopy=true |false - whether the new elements are copied to other caches (true), or whether a remove message is sent. Defaults to true.
- asynchronousReplicationIntervalMillis default 1000ms Time between updates when replication is asynchroneous

21.8 Complete Sample configuration

A typical complete configuration for one replicated cache configured for UDP will look like:

```
<Ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../../../main/config/ehcache.xsd">

<diskStore path="java.io.tmpdir/one"/>

<cacheManagerPeerProviderFactory class="net.sf.ehcache.distribution.jgroups
    .JGroupsCacheManagerPeerProviderFactory"
     properties="connect=UDP(mcast_addr=231.12.21.132;mcast_port=45566;ip_ttl=32;
     mcast_send_buf_size=150000;mcast_recv_buf_size=80000):
     PING(timeout=2000;num_initial_members=6):
     MERGE2(min_interval=5000;max_interval=10000):
     FD_SOCK:VERIFY_SUSPECT(timeout=1500):
     pbcast.NAKACK(gc_lag=10;retransmit_timeout=3000):
     UNICAST(timeout=5000):
     pbcast.STABLE(desired_avg_gossip=20000):
     FRAG:
     pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;
     shun=false;print_local_addr=true)"
 propertySeparator="::"
     />

<cache name="sampleCacheAsync"
```

```
   maxElementsInMemory="1000"
   eternal="false"
   timeToIdleSeconds="1000"
   timeToLiveSeconds="1000"
   overflowToDisk="false">
    <cacheEventListenerFactory
    class="net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory"
    properties="replicateAsynchronously=true, replicatePuts=true,
replicateUpdates=true, replicateUpdatesViaCopy=false,
replicateRemovals=true" />
</cache>

</ehcache>
```

21.9 Common Problems

If replication using JGroups doesnt't work the way you have it configured try this configuration which has been extensively tested:

```
<cacheManagerPeerProviderFactory
    class="net.sf.ehcache.distribution.jgroups.JGroupsCacheManagerPeerProviderFactory"/>

<cache name="sampleCacheAsync"
   maxElementsInMemory="1000"
   eternal="false"
   timeToIdleSeconds="1000"
   timeToLiveSeconds="1000"
   overflowToDisk="false">
   <cacheEventListenerFactory
    class="net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory"
    properties="replicateAsynchronously=true, replicatePuts=true,
    replicateUpdates=true, replicateUpdatesViaCopy=false,
    replicateRemovals=true" />
</cache>
```

If this fails to replicate, try to get the example programs from JGroups to run:

http://www.jgroups.org/javagroupsnew/docs/manual/html/ch02.html#d0e451

and

http://www.jgroups.org/javagroupsnew/docs/manual/html/ch02.html#ItDoesntWork

Once you have figured out the connection string that works in your network for JGroups, you can directly paste it in the connect property of `JGroupsCacheManagerPeerProviderFactory`.

Chapter 22

Distributed Caching using JMS

As of version 1.6, JMS can be used as the underlying mechanism for the distributed operations in Ehcache with the jmsreplication module.

JMS, ("Java Message Service") is an industry standard mechanism for interacting with message queues. Message queues themselves are a very mature piece of infrastructure used in many enterprise software contexts. Because they are a required part of the Java EE specification, the large enterprise vendors all provide their own implementations. There are also several open source choices including Open MQ and Active MQ. Ehcache is integration tested against both of these.

The Ehcache jmsreplication module lets organisations with a message queue investment leverage it for caching.

It provides:

- replication between cache nodes using a replication topic, in accordance with ehcache's standard replication mechanism
- pushing of data directly to cache nodes from external topic publishers, in any language. This is done by sending the data to the replication topic, where it automatically picked up by the cache subscribers.
- a JMSCacheLoader, which sends cache load requests to a queue. Either an Ehcache cluster node, or an external queue receiver can respond.

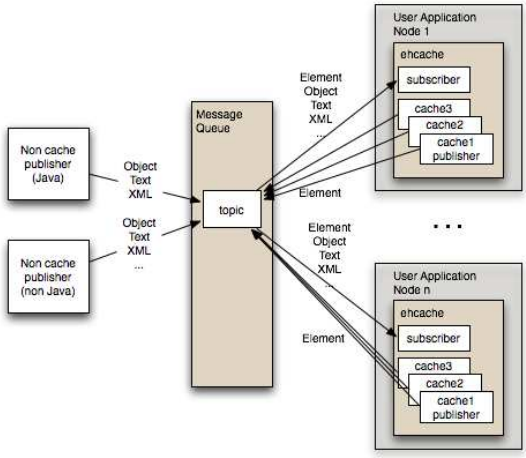
22.1 Ehcache Replication and External Publishers

Ehcache replicates using JMS as follows:

- Each cache node subscribes to a predefined topic, configured as the *topicBindingName* in ehcache.xml.
- Each replicated cache publishes cache `Elements` to that topic. Replication is configured per cache.

  To set up distributed caching using JMS you need to configure a JMSCacheManagerPeerProvider-Factory which is done globally for a CacheManager.

  For each cache that wishing to replicate, you add a JGroupsCacheReplicatorFactory element to the cache element.



22.1.1 Configuration

Message Queue Configuration

Each cluster needs to use a fixed topic name for replication. Set up a topic using the tools in your message queue. Out of the box, both ActiveMQ and Open MQ support auto creation of destinations, so this step may be optional.

Ehcache Configuration

Configuration is done in the ehcache.xml.

There are two things to configure:

- The JMSCacheManagerPeerProviderFactory which is done once per CacheManager and therefore once per ehcache.xml file.
- The JMSCacheReplicatorFactory which is added to each cache's configuration if you want that cache replicated.

  The main configuration happens in the JGroupsCacheManagerPeerProviderFactory connect sub-property. A connect property is passed directly to the JGroups channel and therefore all the protocol stacks and options available in JGroups can be set.

Configuring the JMSCacheManagerPeerProviderFactory Following is the configuration instructions as it appears in the sample ehcache.xml shipped with ehcache:

```
{Configuring JMS replication}.
==========================

<cacheManagerPeerProviderFactory
        class="net.sf.ehcache.distribution.jms.JMSCacheManagerPeerProviderFactory"
        properties="..."
        propertySeparator=","
        />

The JMS PeerProviderFactory uses JNDI to maintain message queue independence.
Refer to the manual for full configuration examples using ActiveMQ and Open Message Queue.
```

```
Valid properties are:
* initialContextFactoryName (mandatory) - the name of the factory used to create the
    message queue initial context.
* providerURL (mandatory) - the JNDI configuration information for the service provider to
use.
* topicConnectionFactoryBindingName (mandatory) - the JNDI binding name for the
    TopicConnectionFactory
* topicBindingName (mandatory) - the JNDI binding name for the topic name
* securityPrincipalName - the JNDI java.naming.security.principal
* securityCredentials - the JNDI java.naming.security.credentials
* urlPkgPrefixes - the JNDI java.naming.factory.url.pkgs
* userName - the user name to use when creating the TopicConnection to the Message Queue
* password - the password to use when creating the TopicConnection to the Message Queue
* acknowledgementMode - the JMS Acknowledgement mode for both publisher and subscriber.
    The available choices are
                        AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE and SESSION_TRANSACTED.
                        The default is AUTO_ACKNOWLEDGE.
* listenToTopic - true or false. If false, this cache will send to the JMS topic but will
    not listen for updates.
* Default is true.
```

Example Configurations Usage is best illustrated with concrete examples for Active MQ and Open MQ.

Configuring the JMSCacheManagerPeerProviderFactory for Active MQ This configuration works with Active MQ out of the box.

```
<cacheManagerPeerProviderFactory
            class="net.sf.ehcache.distribution.jms.JMSCacheManagerPeerProviderFactory"
            properties="initialContextFactoryName=ExampleActiveMQInitialContextFactory,
                providerURL=tcp://localhost:61616,
                topicConnectionFactoryBindingName=topicConnectionFactory,
                topicBindingName=ehcache"
            propertySeparator=","
            />
```

You need to provide your own ActiveMQInitialContextFactory for the initialContextFactoryName.

An example which should work for most purposes is:

```
public class ExampleActiveMQInitialContextFactory extends ActiveMQInitialContextFactory {

/**
 * {@inheritDoc}
 */
@Override
@SuppressWarnings("unchecked")
public Context getInitialContext(Hashtable environment) throws NamingException {

    Map<String, Object> data = new ConcurrentHashMap<String, Object>();

    String factoryBindingName = (String)environment.get(JMSCacheManagerPeerProviderFactory
    .TOPIC_CONNECTION_FACTORY_BINDING_NAME);

    try {
        data.put(factoryBindingName, createConnectionFactory(environment));
```

```
    } catch (URISyntaxException e) {
        throw new NamingException("Error initialisating ConnectionFactory with message "
        + e.getMessage());
    }

    String topicBindingName = (String)environment.get(JMSCacheManagerPeerProviderFactory
    .TOPIC_BINDING_NAME);

    data.put(topicBindingName, createTopic(topicBindingName));

    return createContext(environment, data);
}
}
```

Configuring the JMSCacheManagerPeerProviderFactory for Open MQ This configuration works with an out of the box Open MQ.

```
<cacheManagerPeerProviderFactory
            class="net.sf.ehcache.distribution.jms.JMSCacheManagerPeerProviderFactory"
            properties="initialContextFactoryName=com.sun.jndi.fscontext.RefFSContextFactory,
                providerURL=file:///tmp,
                topicConnectionFactoryBindingName=MyConnectionFactory,
                topicBindingName=ehcache"
            propertySeparator=","
            />
```

To set up the Open MQ file system initial context to work with this example use the following `imqobjmgr` commands to create the requires objects in the context.

```
imqobjmgr add -t tf -l 'MyConnectionFactory' -j java.naming.provider.url \
=file:///tmp -j java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory -f
imqobjmgr add -t t -l 'ehcache' -o 'imqDestinationName=EhcacheTopicDest'
-j java.naming.provider.url\
=file:///tmp -j java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory -f
```

Configuring the JMSCacheReplicatorFactory This is the same as configuring any of the cache replicators. The class should be `net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory`.

See the following example:

```
<cache name="sampleCacheAsync"
      maxElementsInMemory="1000"
      eternal="false"
      timeToIdleSeconds="1000"
      timeToLiveSeconds="1000"
      overflowToDisk="false">
   <cacheEventListenerFactory
          class="net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory"
          properties="replicateAsynchronously=true,
                      replicatePuts=true,
                      replicateUpdates=true,
                      replicateUpdatesViaCopy=true,
                      replicateRemovals=true,
                      asynchronousReplicationIntervalMillis=1000"
```

```
            propertySeparator=","/>
</cache>
```

22.1.2 External JMS Publishers

Anything that can publish to a message queue can also add cache entries to ehcache. These are called non-cache publishers.

Required Message Properties

Publishers need to set up to four String properties on each message: cacheName, action, mimeType and key.

`cacheName` Property A JMS message property which contains the name of the cache to operate on.

If no cacheName is set the message will be *ignored*. A warning log message will indicate that the message has been ignored.

`action` Property A JMS message property which contains the action to perform on the cache.

Available actions are strings labeled `PUT`, `REMOVE` and `REMOVE_ALL`.

If not set no action is performed. A warning log message will indicate that the message has been ignored.

`mimeType` Property A JMS message property which contains the mimeType of the message. Applies to the `PUT` action. If not set the message is interpreted as follows:

ObjectMessage - if it is an net.sf.ehcache.Element, then it is treated as such and stored in the cache.

For other objects, a new Element is created using the object in the ObjectMessage as the value and the key property as a key. Because objects are already typed, the mimeType is ignored.

TextMessage - Stored in the cache as value of MimeTypeByteArray. The mimeType should be specified. If not specified it is stored as type `text/plain`.

BytesMessage - Stored in the cache as value of MimeTypeByteArray. The mimeType should be specified. If not specified it is stored as type `application/octet-stream`.

Other message types are not supported.

To send XML use a TextMessage or BytesMessage and set the mimeType to `application/xml`.It will be stored in the cache as a value of MimeTypeByteArray.

The `REMOVE` and `REMOVE_ALL` actions do not require a `mimeType` property.

`key` Property The key in the cache on which to operate on. The key is of type String.

The `REMOVE_ALL` action does not require a key property.

If an ObjectMessage of type net.sf.ehcache.Element is sent, the key is contained in the element. Any key set as a property is ignored.

If the key is required but not provided, a warning log message will indicate that the message has been ignored.

Code Samples

These samples use Open MQ as the message queue and use it with out of the box defaults. They are heavily based on Ehcache's own JMS integration tests. See the test source for more details.

Messages should be sent to the topic that Ehcache is listening on. In these samples it is `EhcacheTopicDest`.

All samples get a Topic Connection using the following method:

```
private TopicConnection getMQConnection() throws JMSException {
    com.sun.messaging.ConnectionFactory factory = new com.sun.messaging.ConnectionFactory();
    factory.setProperty(ConnectionConfiguration.imqAddressList, "localhost:7676");
    factory.setProperty(ConnectionConfiguration.imqReconnectEnabled, "true");
    TopicConnection myConnection = factory.createTopicConnection();
    return myConnection;
}
```

PUT a Java Object into an Ehcache JMS Cluster

```
String payload = "this is an object";
TopicConnection connection = getMQConnection();
connection.start();

TopicSession publisherSession =
    connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

ObjectMessage message = publisherSession.createObjectMessage(payload);
message.setStringProperty(ACTION_PROPERTY, "PUT");
message.setStringProperty(CACHE_NAME_PROPERTY, "sampleCacheAsync");
//don't set. Should work.
//message.setStringProperty(MIME_TYPE_PROPERTY, null);
//should work. Key should be ignored when sending an element.
message.setStringProperty(KEY_PROPERTY, "1234");


Topic topic = publisherSession.createTopic("EhcacheTopicDest");
TopicPublisher publisher = publisherSession.createPublisher(topic);
publisher.send(message);

connection.stop();
```

Ehcache will create an Element in cache "sampleCacheAsync" with key "1234" and a Java class String value of "this is an object".

PUT XML into an Ehcache JMS Cluster

```
TopicConnection connection = getMQConnection();
connection.start();

TopicSession publisherSession = connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);

String value = "<?xml version=\"1.0\"?>\n" +
        "<oldjoke>\n" +
        "<burns>Say <quote>goodnight</quote>,\n" +
        "Gracie.</burns>\n" +
        "<allen><quote>Goodnight, \n" +
```

```
        "Gracie.</quote></allen>\n" +
        "<applause/>\n" +
        "</oldjoke>";

TextMessage message = publisherSession.createTextMessage(value);
message.setStringProperty(ACTION_PROPERTY, "PUT");
message.setStringProperty(CACHE_NAME_PROPERTY, "sampleCacheAsync");
message.setStringProperty(MIME_TYPE_PROPERTY, "application/xml");
message.setStringProperty(KEY_PROPERTY, "1234");


Topic topic = publisherSession.createTopic("EhcacheTopicDest");
TopicPublisher publisher = publisherSession.createPublisher(topic);
publisher.send(message);

connection.stop();
```

Ehcache will create an Element in cache "sampleCacheAsync" with key "1234" and a value of type MimeTypeByteArray.

On a get from the cache the MimeTypeByteArray will be returned. It is an Ehcache value object from which a mimeType and byte[] can be retrieved. The mimeType will be "application/xml". The byte[] will contain the XML String encoded in bytes, using the platform's default charset.

PUT arbitrary bytes into an Ehcache JMS Cluster

```
byte[] bytes = new byte[]{0x34, (byte) 0xe3, (byte) 0x88};
TopicConnection connection = getMQConnection();
connection.start();

TopicSession publisherSession = connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);

BytesMessage message = publisherSession.createBytesMessage();
message.writeBytes(bytes);
message.setStringProperty(ACTION_PROPERTY, "PUT");
message.setStringProperty(CACHE_NAME_PROPERTY, "sampleCacheAsync");
message.setStringProperty(MIME_TYPE_PROPERTY, "application/octet-stream");
message.setStringProperty(KEY_PROPERTY, "1234");


Topic topic = publisherSession.createTopic("EhcacheTopicDest");
TopicPublisher publisher = publisherSession.createPublisher(topic);
publisher.send(message);
```

Ehcache will create an Element in cache "sampleCacheAsync" with key "1234" in and a value of type MimeTypeByteArray.

On a get from the cache the MimeTypeByteArray will be returned. It is an Ehcache value object from which a mimeType and byte[] can be retrieved. The mimeType will be "application/octet-stream". The byte[] will contain the original bytes.

REMOVE

```
TopicConnection connection = getMQConnection();
connection.start();
```

```
TopicSession publisherSession = connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE)

ObjectMessage message = publisherSession.createObjectMessage();
message.setStringProperty(ACTION_PROPERTY, "REMOVE");
message.setStringProperty(CACHE_NAME_PROPERTY, "sampleCacheAsync");
message.setStringProperty(KEY_PROPERTY, "1234");


Topic topic = publisherSession.createTopic("EhcacheTopicDest");
TopicPublisher publisher = publisherSession.createPublisher(topic);
publisher.send(message);
```

Ehcache will remove the Element with key "1234" from cache "sampleCacheAsync" from the cluster.

REMOVE_ALL

```
TopicConnection connection = getMQConnection();
connection.start();

TopicSession publisherSession = connection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);

ObjectMessage message = publisherSession.createObjectMessage();
message.setStringProperty(ACTION_PROPERTY, "REMOVE_ALL");
message.setStringProperty(CACHE_NAME_PROPERTY, "sampleCacheAsync");

Topic topic = publisherSession.createTopic("EhcacheTopicDest");
TopicPublisher publisher = publisherSession.createPublisher(topic);
publisher.send(message);

connection.stop();
```

Ehcache will remove all Elements from cache "sampleCacheAsync" in the cluster.

22.2 Using the JMSCacheLoader

The JMSCacheLoader is a CacheLoader which loads objects into the cache by sending requests to a JMS Queue.

The loader places an ObjectMessage of type JMSEventMessage on the getQueue with an Action of type GET.

It is configured with the following String properties, `loaderArgument`.

The defaultLoaderArgument, or the loaderArgument if specified on the load request. To work with the JMSCacheManagerPeerProvider this should be the name of the cache to load from. For custom responders, it can be anything which has meaning to the responder.

A queue responder will respond to the request. You can either create your own or use the one built-into the JMSCacheManagerPeerProviderFactory, which attempts to load the queue from its cache.

The JMSCacheLoader uses JNDI to maintain message queue independence. Refer to the manual for full configuration examples using ActiveMQ and Open Message Queue.

It is configured as per the following example:

```
<cacheLoaderFactory class="net.sf.ehcache.distribution.jms.JMSCacheLoaderFactory"
   properties="initialContextFactoryName=com.sun.jndi.fscontext.RefFSContextFactory,
```

```
providerURL=file:///tmp,
replicationTopicConnectionFactoryBindingName=MyConnectionFactory,
replicationTopicBindingName=ehcache,
getQueueConnectionFactoryBindingName=queueConnectionFactory,
getQueueBindingName=ehcacheGetQueue,
timeoutMillis=20000
defaultLoaderArgument=/>
```

Valid properties are:

- initialContextFactoryName (mandatory) - the name of the factory used to create the message queue initial context.
- providerURL (mandatory) - the JNDI configuration information for the service provider to use.
- getQueueConnectionFactoryBindingName (mandatory) - the JNDI binding name for the QueueConnectionFactory
- getQueueBindingName (mandatory) - the JNDI binding name for the queue name used to do make requests.
- defaultLoaderArgument - (optional) - an application specific argument. If not supplied as a cache.load() parameter this default value will be used. The argument is passed in the JMS request as a StringProperty called loaderArgument.
- timeoutMillis - time in milliseconds to wait for a reply.
- securityPrincipalName - the JNDI java.naming.security.principal
- securityCredentials - the JNDI java.naming.security.credentials
- urlPkgPrefixes - the JNDI java.naming.factory.url.pkgs
- userName - the user name to use when creating the TopicConnection to the Message Queue
- password - the password to use when creating the TopicConnection to the Message Queue
- acknowledgementMode - the JMS Acknowledgement mode for both publisher and subscriber. The available choices are AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE and SESSION_TRANSACTED. The default is AUTO_ACKNOWLEDGE.

22.2.1 Example Configuration Using Active MQ

```
<cache name="sampleCacheNorep"
   maxElementsInMemory="1000"
   eternal="false"
   timeToIdleSeconds="1000"
   timeToLiveSeconds="1000"
   overflowToDisk="false">
   <cacheEventListenerFactory
    class="net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory"
        properties="replicateAsynchronously=false, replicatePuts=false,
        replicateUpdates=false, replicateUpdatesViaCopy=false,
        replicateRemovals=false, loaderArgument=sampleCacheNorep"
        propertySeparator="," />
   <cacheLoaderFactory class="net.sf.ehcache.distribution.jms.JMSCacheLoaderFactory"
    properties="initialContextFactoryName=net.sf.ehcache.distribution.jms.
        TestActiveMQInitialContextFactory,
```

```
            providerURL=tcp://localhost:61616,
            replicationTopicConnectionFactoryBindingName=topicConnectionFactory,
            getQueueConnectionFactoryBindingName=queueConnectionFactory,
            replicationTopicBindingName=ehcache,
            getQueueBindingName=ehcacheGetQueue,
            timeoutMillis=10000"/>
</cache>
```

22.2.2 Example Configuration Using Open MQ

```
<cache name="sampleCacheNorep"
   maxElementsInMemory="1000"
   eternal="false"
   timeToIdleSeconds="100000"
   timeToLiveSeconds="100000"
   overflowToDisk="false">
   <cacheEventListenerFactory
   class="net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory"
                           properties="replicateAsynchronously=false, replicatePuts=false,
                             replicateUpdates=false, replicateUpdatesViaCopy=false,
                             replicateRemovals=false"
                           propertySeparator=","/>
   <cacheLoaderFactory class="net.sf.ehcache.distribution.jms.JMSCacheLoaderFactory"
        properties="initialContextFactoryName=com.sun.jndi.fscontext.RefFSContextFactory,
        providerURL=file:///tmp,
        replicationTopicConnectionFactoryBindingName=MyConnectionFactory,
        replicationTopicBindingName=ehcache,
        getQueueConnectionFactoryBindingName=queueConnectionFactory,
        getQueueBindingName=ehcacheGetQueue,
        timeoutMillis=10000,
        userName=test,
        password=test"/>
</cache>
```

22.3 Configuring Clients for Message Queue Reliability

Ehcache replication and cache loading is designed to gracefully degrade if the message queue infrastructure stops. Replicates and loads will fail. But when the message queue comes back, these operations will start up again.

For this to work, the ConnectionFactory used with the specific message queue needs to be configured correctly.

For example, with Open MQ, reconnection is configured as follows:

- imqReconnect='true' - without this reconnect will not happen
- imqPingInterval='5' - Consumers will not reconnect until they notice the connection is down. The ping interval
- does this. The default is 30. Set it lower if you want the Ehcache cluster to reform more quickly.
- Finally, unlimited retry attempts are recommended. This is also the default.

  For greater reliability consider using a message queue cluster. Most message queues support clustering. The cluster configuration is once again placed in the ConnectionFactory configuration.

22.4 Tested Message Queues

22.4.1 Sun Open MQ

This open source message queue is tested in integration tests. It works perfectly.

22.4.2 Active MQ

This open source message queue is tested in integration tests. It works perfectly other than having a problem with temporary reply queues which prevents the use of JMSCacheLoader. JMSCacheLoader is not used during replication.

22.4.3 Oracle AQ

Versions up to an including 0.4 do not work, due to Oracle not supporting the unified API (send) for topics.

22.4.4 JBoss Queue

Works as reported by a user.

22.5 Known JMS Issues

22.5.1 Active MQ Temporary Destinatons

ActiveMQ seems to have a bug in at least ActiveMQ 5.1 where it does not cleanup temporary queues, even though they have been deleted. That bug appears to be long standing but was though to have been fixed.

See:

- http://www.nabble.com/Memory-Leak-Using-Temporary-Queues-td11218217.html#a11218217
- http://issues.apache.org/activemq/browse/AMQ-1255

  The JMSCacheLoader uses temporary reply queues when loading. The Active MQ issue is readily reproduced in Ehcache integration testing. Accordingly, use of the JMSCacheLoader with ActiveMQ is not recommended. Open MQ tests fine.

  Active MQ works fine for replication.

22.5.2 WebSphere 5 and 6

Websphere Application Server prevents MessageListeners, which are not MDBs, from being created in the container. While this is a general Java EE limitation, most other app servers either are permissive or can be configured to be permissive. WebSphere 4 worked, but 5 and 6 enforce the restriction.

Accordingly the JMS replicator cannot be used with WebSphere 5 and 6.

Chapter 23

Distributed Caching Using Terracotta

Terracotta has been integrated with Ehcache since Ehcache 1.4.

From version 1.7 Ehcache has been seamlessly integrated with Terracotta 3.1.1 and takes just a few lines of config in ehcache.xml to get up and running.

23.1 Worked Example

As this example shows, running Ehcache with Terracotta clustering is no different from normal programmatic use.

```
import net.sf.ehcache.Cache;
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Element;

public class TerracottaExample {
    CacheManager cacheManager = new CacheManager();

    public TerracottaExample() {
        Cache cache = cacheManager.getCache("sampleTerracottaCache");
        int cacheSize = cache.getKeys().size();
        cache.put(new Element("" + cacheSize, cacheSize));
        for (Object key : cache.getKeys()) {
            System.out.println("Key:" + key);
        }
    }

    public static void main(String[] args) throws Exception {
        new TerracottaExample();
    }
}
```

The above example looks for sampleTerracottaCache.

In ehcache.xml, we need to uncomment or add the following line:

```
    <terracottaConfig url="localhost:9510"/>
```

which tells Ehcache to load the Terracotta server config from localhost port 9510. Note: You must have a Terracotta 3.1.1 or higher server running locally for this example.

Next we want to enable Terracotta clustering for the cache named `sampleTerracottaCache`. Uncomment or add the following in ehcache.xml.

```
<cache name="sampleTerracottaCache"
       maxElementsInMemory="1000"
       eternal="false"
       timeToIdleSeconds="3600"
       timeToLiveSeconds="1800"
       overflowToDisk="false">
    <terracotta/>
</cache>
```

That's it!

23.2 Terracotta Configuration

Terracotta configuration in ehcache.xml is in three parts:

- CacheManager Configuration
- Terracotta Server Configuration
- Enabling Terracotta clustering per cache

23.2.1 CacheManager Configuration

The attributes of *ehcache* are:

- name

  an optional name for the CacheManager. The name is optional and primarily used for documentation or to distinguish Terracotta clustered cache state. With Terracotta clustered caches, a combination of CacheManager name and cache name uniquely identify a particular cache store in the Terracotta clustered memory.

  The name will show up in the Developer Console.s

- updateCheck

  an optional boolean flag specifying whether this CacheManager should check for new versions of Ehcache over the Internet. If not specified, updateCheck="true".

- monitoring

  an optional setting that determines whether the CacheManager should automatically register the SampledCacheMBean with the system MBean server. Currently, this monitoring is only useful when using Terracotta and thus the "autodetect" value will detect the presence of Terracotta and register the MBean. Other allowed values are "on" and "off". The default is "autodetect".

```
<Ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="ehcache.xsd"
             updateCheck="true" monitoring="autodetect">
```

23.2.2 Terracotta Server Configuration

Note: You need to install and run one or more Terracotta servers to use Terracotta clustering.

See http://www.terracotta.org/web/display/orgsite/Download.

With a server/servers up and running you need to specify the location of the servers.

Configuration can be specified in two main ways: by reference to a source of configuration or by use of an embedded Terracotta configuration file.

Specification of a source of configuration

To specify a reference to a source (or sources) of configuration, use the url attribute. The url attribute must contain a comma-separated list of:

- path to the Terracotta configuration file (usually named tc-config.xml)

  Example using a path to Terracotta configuration file:

  ```
  <terracottaConfig url="/app/config/tc-config.xml"/>
  ```

- URL to the Terracotta configuration file

  Example using a URL to a Terracotta configuration file:

  ```
  <terracottaConfig url="http://internal/ehcache/app/tc-config.xml"/>
  ```

- *server host*:*port* of a running Terracotta Server instance

  Example pointing to a Terracotta server installed on localhost:

  ```
  <terracottaConfig url="localhost:9510"/>
  ```

  Example using multiple Terracotta server instance URLs (for fault tolerance):

  ```
  <terracottaConfig url="host1:9510,host2:9510,host3:9510"/>
  ```

Specification using embedded tc-config

To embed a Terracotta configuration file within the Ehcache configuration, simply place the usual Terracotta XML config within the *terracottaConfig* element.

In this example we have two Terracotta servers running on `server1` and `server2`.

```
<terracottaConfig>
    <tc-config>
        <servers>
            <server host="server1" name="s1"/>
            <server host="server2" name="s2"/>
        </servers>
        <clients>
            <logs>app/logs-%i</logs>
        </clients>
    </tc-config>
</terracottaConfig>
```

23.2.3 Enabling Terracotta clustering per cache

Cache elements can also contain information about whether the cache can be clustered with Terracotta.

The *terracotta* sub-element has the following attributes:

- `clustered=true|false`

  Indicates whether this cache should be clustered with Terracotta. By default, if the *terracotta* element is included, clustered=true.

- `valueMode=serialization|identity`

  Indicates whether this cache should be clustered with serialized copies of the values or using Terracotta identity mode. By default, values will be cached in serialization mode which is similar to other replicated Ehcache modes. The identity mode is only available in certain Terracotta deployment scenarios and will maintain actual object identity of the keys and values across the cluster. In this case, all users of a value retrieved from the cache are using the same clustered value and must provide appropriate locking for any changes made to the value (or objects referred to by the value).

- `coherentReads=true|false`

  Indicates whether this cache should have coherent reads with guaranteed consistency across the cluster. By default, this setting is true. If you set this property to false, reads are allowed to check the local value without locking, possibly seeing stale values.

  This is a performance optimization with weaker concurrency guarantees and should generally be used with caches that contain read-only data or where the application can tolerate reading stale data.

  The simplest way to enable clustering is to add:

  ```
  <terracotta/>
  ```

  To indicate the cache should not be clustered (or remove the *terracotta* element altogether):

  ```
  <terracotta clustered="false"/>
  ```

  To indicate the cache should be clustered using identity mode:

  ```
  <terracotta clustered="true" valueMode="identity"/>
  ```

  Following is an example Terracotta clustered cache named sampleTerracottaCache.

```
<cache name="sampleTerracottaCache"
       maxElementsInMemory="1000"
       eternal="false"
       timeToIdleSeconds="3600"
       timeToLiveSeconds="1800"
       overflowToDisk="false">
    <terracotta/>
</cache>
```

23.3 Behaviour differences with `CacheEventListener`s when using Terracotta Clustering

Terracotta clustering works by clustering the `MemoryStore`, unlike the replication mechanisms which use the `CacheEventListener` infrastructure.

This results in a simpler programming contract than with the replication mechanisms.

Things to note:

23.3.1 Cache listeners

Cache listeners listen for changes, including replicated cluster changes, made through the Cache API. Because Terracotta cluster changes happen transparently directly to the `MemoryStore` a listener will not be invoked when an event occurs out on the cluster. If it occurs locally, then it must have occurred through the Cache API, so a local event will be detected by a local listener.

A common use of listeners is to trigger a reload of a just invalidated `Element`. In Terracotta clustering this is avoided as a change in one node is always coherent to the other nodes.

23.3.2 Overflow to Disk

Overflow to Disk is not supported when using Terracotta Clustering. However it is also not needed, because the Terracotta server has its own overflow to disk. Once the `maxElementsInMemory` limit is reached `Elements` will be evicted to the cluster.

23.4 More Information

Please see Terracotta Documentation for much more information.

23.5 FAQ

23.5.1 Is Expiry the same in Terracotta?

timeToIdle and timeToLive work as usual. Ehcache 1.7 introduced a less fine grained age recording in Element which rounds up to the nearest second. Some APIs may be sensitive to this change.

In Ehcache Elements can have overridden TTI and TTLs. Terracotta supports this functionality.

23.5.2 What Eviction strategies are supported?

Ehcache supports LRU, LFU and FIFO eviction strategies.

Terracotta supports LRU and LFU eviction from the local node. Not FIFO and not custom evictors.

23.5.3 What Stores are available and how are they configured?

The Terracotta server provides an additional store, generally referred to as the Level 2 or L2 store.

The MemoryStore in JVM in the local node is referred to as the L1 Store.

maxElementsInMemory - the maximum number of elements in the local L1 store.

maxElementsOnDisk - is overridden when using Terracotta to provide the L2 size.

overflowToDisk normally controls whether to overflow to the DiskStore. This is ignored when using Terracotta - the DiskStore is never used. When the store gets full, elements will always overflow to the Terracotta L2 Store running on the server. The L2 can be further configured with the tcconfig.

23.5.4 When do Elements overflow?

Two things to cause elements to be flushed from L1 to L2.

- the L1 store exceeding maxElementsInMemory
- When the local JMV exceeds 70% of Old Generation. This can be turned off in the TC Config. By default it is on (in 1.7).

23.5.5 How does Element equality work in Serialization mode?

An Element in Ehcache is guaranteed to `.equals()` another as it moves between stores.

In the Express install or Serialization mode of Terracotta, which is the default, Terracotta is the same. Elements will not == each other as they move between stores.

23.5.6 How does Element equality work in Identity mode?

An Element in Ehcache is guaranteed to `.equals()` another as it moves between stores.

However in Identity mode, Terracotta makes a further guarantee - that they will == each other. This is achieved using extensions to the Java Memory Model.

23.5.7 What is the recommended way to write to a database?

Terracotta' non Ehcache API offers an async writethrough to the database which is guaranteed. It uses the TIM Async module and works by putting the database update in a clustered queue. It guarantees that a node, even if the local node fails, will take it out and process it.

That option is not available with Ehcache although it may get added.

23.5.8 If updates to a database bypass the Terracotta clustered application, then how is that coherent?

It isn't. This is a problem with using a database as an integration point. Integration via a message queue, with a Terracotta clustered application acting as a message queue listener and updating the database avoids this. As would The application receiving a REST or SOAP call and writing to the database.

AQ can have DB trigger put in a poll. Or AQ can push it up.

23.5.9 Do CacheEventListeners work?

A local CacheEventListener will work locally, but other nodes in a Terracotta cluster are not notified. With the Ehcache replication non Terracotta mechanisms the remote nodes are updated.

Chapter 24

BlockingCache and SelfPopulatingCache

The `net.sf.ehcache.constructs` package contains some applied caching classes which use the core classes to solve everyday caching problems.

24.1 Blocking Cache

Imagine you have a very busy web site with thousands of concurrent users. Rather than being evenly distributed in what they do, they tend to gravitate to popular pages. These pages are not static, they have dynamic data which goes stale in a few minutes. Or imagine you have collections of data which go stale in a few minutes. In each case the data is extremely expensive to calculate.

Let's say each request thread asks for the same thing. That is a lot of work. Now, add a cache. Get each thread to check the cache; if the data is not there, go and get it and put it in the cache. Now, imagine that there are so many users contending for the same data that in the time it takes the first user to request the data and put it in the cache, 10 other users have done the same thing. The upstream system, whether a JSP or velocity page, or interactions with a service layer or database are doing 10 times more work than they need to.

Enter the BlockingCache.

Blocking Cache

It is blocking because all threads requesting the same key wait for the first thread to complete. Once the first thread has completed the other threads simply obtain the cache entry and return.

The BlockingCache can scale up to very busy systems. Each thread can either wait indefinitely, or you can specify a timeout using the `timeoutMillis` constructor argument.

24.2 SelfPopulatingCache

You want to use the BlockingCache, but the requirement to always release the lock creates gnarly code. You also want to think about what you are doing without thinking about the caching.

Enter the SelfPopulatingCache. The name SelfPopulatingCache is synonymous with Pull-through cache, which is a common caching term. SelfPopulatingCache though always is in addition to a BlockingCache.

SelfPopulatingCache uses a `CacheEntryFactory`, that given a key, knows how to populate the entry.

Note: JCache inspired getWithLoader and getAllWithLoader directly in `Ehcache` which work with a `CacheLoader` may be used as an alternative to SelfPopulatingCache. CacheManager Event Listeners

CacheManager event listeners allow implementers to register callback methods that will be executed when a `CacheManager` event occurs. Cache listeners implement the CacheManagerEventListener interface.

The events include:

- adding a `Cache`
- removing a `Cache`

Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

24.3 Configuration

One CacheManagerEventListenerFactory and hence one CacheManagerEventListener can be specified per CacheManager instance.

The factory is configured as below:

```
<cacheManagerEventListenerFactory class="" properties=""/>
```

The entry specifies a CacheManagerEventListenerFactory which will be used to create a CacheManager-PeerProvider, which is notified when Caches are added or removed from the CacheManager.

The attributes of CacheManagerEventListenerFactory are:

- `class` - a fully qualified factory class name
- `properties` - comma separated properties having meaning only to the factory.

  Callbacks to listener methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

  If no class is specified, or there is no cacheManagerEventListenerFactory element, no listener is created. There is no default.

24.4 Implementing a CacheManagerEventListenerFactory and CacheManagerEventListener

CacheManagerEventListenerFactory is an abstract factory for creating cache manager listeners. Implementers should provide their own concrete factory extending this abstract factory. It can then be configured in ehcache.xml.

The factory class needs to be a concrete subclass of the abstract factory CacheManagerEventListenerFactory, which is reproduced below:

```
/**
 * An abstract factory for creating {@link CacheManagerEventListener}s. Implementers should
 * provide their own concrete factory extending this factory. It can then be configured in
 * ehcache.xml
 *
 * @author Greg Luck
 * @version $Id: cachemanager_event_listeners.apt 735 2008-08-10 23:51:48Z gregluck $
 * @see "http://ehcache.org/documentation/cachemanager_event_listeners.html"
 */
public abstract class CacheManagerEventListenerFactory {

    /**
     * Create a <code>CacheEventListener</code>
     *
     * @param properties implementation specific properties. These are configured as comma
     *                   separated name value pairs in ehcache.xml. Properties may be null
     * @return a constructed CacheManagerEventListener
     */
    public abstract CacheManagerEventListener
            createCacheManagerEventListener(Properties properties);
}
```

The factory creates a concrete implementation of CacheManagerEventListener, which is reproduced below:

```
/**
 * Allows implementers to register callback methods that will be executed when a
 * <code>CacheManager</code> event occurs.
 * The events include:
 * <ol>
 * <li>adding a <code>Cache</code>
 * <li>removing a <code>Cache</code>
 * </ol>
 * <p/>
 * Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of
 * the implementer to safely handle the potential performance and thread safety issues
 * depending on what their listener is doing.
 * @author Greg Luck
 * @version $Id: cachemanager_event_listeners.apt 735 2008-08-10 23:51:48Z gregluck $
 * @since 1.2
 * @see CacheEventListener
 */
public interface CacheManagerEventListener {


/**
 * Called immediately after a cache has been added and activated.
 * <p/>
 * Note that the CacheManager calls this method from a synchronized method. Any attempt to
 * call a synchronized method on CacheManager from this method will cause a deadlock.
 * <p/>
 * Note that activation will also cause a CacheEventListener status change notification
 * from {@link net.sf.ehcache.Status#STATUS_UNINITIALISED} to
 * {@link net.sf.ehcache.Status#STATUS_ALIVE}. Care should be taken on processing that
 * notification because:
 * <ul>
```

```
     * <li>the cache will not yet be accessible from the CacheManager.
     * <li>the addCaches methods whih cause this notification are synchronized on the
     * CacheManager. An attempt to call {@link net.sf.ehcache.CacheManager#getCache(String)}
     * will cause a deadlock.
     * </ul>
     * The calling method will block until this method returns.
     * <p/>
     * @param cacheName the name of the <code>Cache</code> the operation relates to
     * @see CacheEventListener
     */
    void notifyCacheAdded(String cacheName);

    /**
     * Called immediately after a cache has been disposed and removed. The calling method will
     * block until this method returns.
     * <p/>
     * Note that the CacheManager calls this method from a synchronized method. Any attempt to
     * call a synchronized method on CacheManager from this method will cause a deadlock.
     * <p/>
     * Note that a {@link CacheEventListener} status changed will also be triggered. Any
     * attempt from that notification to access CacheManager will also result in a deadlock.
     * @param cacheName the name of the <code>Cache</code> the operation relates to
     */
    void notifyCacheRemoved(String cacheName);

    }
```

The implementations need to be placed in the classpath accessible to ehcache. Ehcache uses the Class-Loader returned by `Thread.currentThread().getContextClassLoader()` to load classes.

Chapter 25

Cache Loaders

A `CacheLoader` is an interface which specifies `load` and `loadAll` methods with a variety of parameters.

CacheLoaders come from JCache, but are a frequently requested feature, so they have been incorporated into the core Ehcache classes and can be configured in ehcache.xml.

CacheLoaders are invoked in the following Cache methods:

- getWithLoader (synchronous)
- getAllWithLoader (synchronous)
- load (asynchronous)
- loadAll (asynchronous)

They are also invoked in similar (though slightly differently named) JCache methods.

The methods will invoke a CacheLoader if there is no entry for the key or keys requested. By implementing CacheLoader, an application form of loading can take place. The get... methods follow the pull-through cache pattern. The load... methods are useful as cache warmers.

CacheLoaders are similar to the CacheEntryFactory used in SelfPopulatingCache. However SelfPopulatingCache is a decorator to ehcache. The CacheLoader functionality is available right in a Cache, Ehcache or JCache and follows a more industry standard convention.

CacheLoaders may be set either declaratively in the ehcache.xml configuration file or programmatically. This becomes the default CacheLoader. Some of the methods invoking loaders enable an override CacheLoader to be passed in as a parameter.

More than one cacheLoader can be registered, in which case the loaders form a chain which are executed in order. If a loader returns null, the next in chain is called.

25.1 Declarative Configuration

cacheLoaderFactory - Specifies a CacheLoader, which can be used both asynchronously and synchronously to load objects into a cache. More than one cacheLoaderFactory element can be added, in which case the loaders form a chain which are executed in order. If a loader returns null, the next in chain is called.

```
<cache ...>
      <cacheLoaderFactory class="com.example.ExampleCacheLoaderFactory"
```

```
                                        properties="type=int,startCounter=10"/>
</cache>
```

25.2 Implementing a CacheLoaderFactory and CacheLoader

CacheLoaderFactory is an abstract factory for creating CacheLoaders. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml

The factory class needs to be a concrete subclass of the abstract factory class CacheLoaderFactory, which is reproduced below:

```
/**
 * An abstract factory for creating cache loaders. Implementers should provide their own
 * concrete factory extending this factory.
 * <p/>
 * There is one factory method for JSR107 Cache Loaders and one for Ehcache ones. The Ehcache
 * loader is a sub interface of the JSR107 Cache Loader.
 * <p/>
 * Note that both the JCache and Ehcache APIs also allow the CacheLoader to be set
 * programmatically.
 * @author Greg Luck
 * @version $Id: cache_loaders.apt 860 2008-12-08 07:58:27Z gregluck $
 */
public abstract class CacheLoaderFactory {

/**
 * Creates a CacheLoader using the JSR107 creational mechanism.
 * This method is called from {@link net.sf.ehcache.jcache.JCacheFactory}
 *
 * @param environment the same environment passed into
 * {@link net.sf.ehcache.jcache.JCacheFactory}.
 * This factory can extract any properties it needs from the environment.
 * @return a constructed CacheLoader
 */
public abstract net.sf.jsr107cache.CacheLoader createCacheLoader(Map environment);


/**
 * Creates a CacheLoader using the Ehcache configuration mechanism at the time
 * the associated cache is created.
 *
 * @param properties implementation specific properties. These are configured as comma
 *                   separated name value pairs in ehcache.xml
 * @return a constructed CacheLoader
 */
public abstract net.sf.ehcache.loader.CacheLoader createCacheLoader(Properties properties);

/**
 * @param cache the cache this extension should hold a reference to,
 * and to whose lifecycle it should be bound.
 * @param properties implementation specific properties configured as delimiter
 *  separated name value pairs in ehcache.xml
 * @return a constructed CacheLoader
 */
public abstract CacheLoader createCacheLoader(Ehcache cache, Properties properties);
```

```
}
```

The factory creates a concrete implementation of the CacheLoader interface, which is reproduced below.

A CacheLoader is bound to the lifecycle of a cache, so that `init()` is called during cache initialization, and `dispose()` is called on disposal of a cache.

```
/**
 * Extends JCache CacheLoader with load methods that take an argument in addition to a key
 * @author Greg Luck
 * @version $Id: cache_loaders.apt 860 2008-12-08 07:58:27Z gregluck $
 */
public interface CacheLoader extends net.sf.jsr107cache.CacheLoader {

/**
 * Load using both a key and an argument.
 * <p/>
 * JCache will call through to the load(key) method, rather than this method,
 * where the argument is null.
 *
 * @param key      the key to load the object for
 * @param argument can be anything that makes sense to the loader
 * @return the Object loaded
 * @throws CacheException
 */
Object load(Object key, Object argument) throws CacheException;

/**
 * Load using both a key and an argument.
 * <p/>
 * JCache will use the loadAll(key) method where the argument is null.
 *
 * @param keys     the keys to load objects for
 * @param argument can be anything that makes sense to the loader
 * @return a map of Objects keyed by the collection of keys passed in.
 * @throws CacheException
 */
Map loadAll(Collection keys, Object argument) throws CacheException;

/**
 * Gets the name of a CacheLoader
 *
 * @return the name of this CacheLoader
 */
String getName();

/**
 * Creates a clone of this extension. This method will only be called by Ehcache before a
 * cache is initialized.
 * <p/>
 * Implementations should throw CloneNotSupportedException if they do not support clone
 * but that will stop them from being used with defaultCache.
 *
 * @return a clone
 * @throws CloneNotSupportedException if the extension could not be cloned.
 */
public CacheLoader clone(Ehcache cache) throws CloneNotSupportedException;
```

```
/**
 * Notifies providers to initialise themselves.
 * <p/>
 * This method is called during the Cache's initialise method after it has changed it's
 * status to alive. Cache operations are legal in this method.
 *
 * @throws net.sf.ehcache.CacheException
 */
void init();

/**
 * Providers may be doing all sorts of exotic things and need to be able to clean up on
 * dispose.
 * <p/>
 * Cache operations are illegal when this method is called. The cache itself is partly
 * disposed when this method is called.
 *
 * @throws net.sf.ehcache.CacheException
 */
void dispose() throws net.sf.ehcache.CacheException;


/**
 * @return the status of the extension
 */
public Status getStatus();
}
```

The implementations need to be placed in the classpath accessible to ehcache.

See the chapter on Classloading for details on how classloading of these classes will be done.

25.3 Programmatic Configuration

The following methods on `Cache` allow runtime interrogation, registration and unregistration of loaders:

```
/**
 * Register a {@link CacheLoader} with the cache. It will then be tied into the cache
 *   lifecycle.
 * <p/>
 * If the CacheLoader is not initialised, initialise it.
 *
 * @param cacheLoader A Cache Loader to register
 */
public void registerCacheLoader(CacheLoader cacheLoader) {
    registeredCacheLoaders.add(cacheLoader);
}

/**
 * Unregister a {@link CacheLoader} with the cache. It will then be detached from the cache
 * lifecycle.
 *
 * @param cacheLoader A Cache Loader to unregister
 */
public void unregisterCacheLoader(CacheLoader cacheLoader) {
    registeredCacheLoaders.remove(cacheLoader);
```

```
}


/**
 * @return the cache loaders as a live list
 */
public List<CacheLoader> getRegisteredCacheLoaders() {
    return registeredCacheLoaders;
}
```

Chapter 26

Cache Event Listeners

Cache listeners allow implementers to register callback methods that will be executed when a cache event occurs. Cache listeners implement the CacheEventListener interface.

The events include:

- an Element has been put
- an Element has been updated. Updated means that an Element exists in the Cache with the same key as the Element being put.
- an Element has been removed
- an Element expires, either because timeToLive or timeToIdle have been reached.

Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

Listeners are guaranteed to be notified of events in the order in which they occurred.

Elements can be put or removed from a Cache without notifying listeners by using the putQuiet and removeQuiet methods.

26.1 Configuration

Cache event listeners are configured per cache. Each cache can have multiple listeners.

Each listener is configured by adding a cacheManagerEventListenerFactory element as follows:

```
<cache ...>
<cacheEventListenerFactory class="" properties=""/>
...
</cache>
```

The entry specifies a CacheManagerEventListenerFactory which is used to create a CachePeerProvider, which then receives notifications.

The attributes of CacheManagerEventListenerFactory are:

- class - a fully qualified factory class name * properties - an optional comma separated properties having meaning only to the factory.

  Callbacks to listener methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

26.2 Implementing a CacheEventListenerFactory and CacheEventListener

CacheEventListenerFactory is an abstract factory for creating cache event listeners. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml

The factory class needs to be a concrete subclass of the abstract factory class CacheEventListenerFactory, which is reproduced below:

```
/**
 * An abstract factory for creating listeners. Implementers should provide their own
 * concrete factory extending this factory. It can then be configured in ehcache.xml
 *
 * @author Greg Luck
 * @version $Id: cache_event_listeners.apt 735 2008-08-10 23:51:48Z gregluck $
 */
public abstract class CacheEventListenerFactory {

/**
 * Create a <code>CacheEventListener</code>
 *
 * @param properties implementation specific properties. These are configured as comma
 *                   separated name value pairs in ehcache.xml
 * @return a constructed CacheEventListener
 */
public abstract CacheEventListener createCacheEventListener(Properties properties);

}
```

The factory creates a concrete implementation of the CacheEventListener interface, which is reproduced below:

```
/**
 * Allows implementers to register callback methods that will be executed when a cache event
 *  occurs.
 * The events include:
 * <ol>
 * <li>put Element
 * <li>update Element
 * <li>remove Element
 * <li>an Element expires, either because timeToLive or timeToIdle has been reached.
 * </ol>
 * <p/>
 * Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of
 * the implementer to safely handle the potential performance and thread safety issues
 * depending on what their listener is doing.
 * <p/>
 * Events are guaranteed to be notified in the order in which they occurred.
 * <p/>
```

```
 * Cache also has putQuiet and removeQuiet methods which do not notify listeners.
 *
 * @author Greg Luck
 * @version $Id: cache_event_listeners.apt 735 2008-08-10 23:51:48Z gregluck $
 * @see CacheManagerEventListener
 * @since 1.2
 */
public interface CacheEventListener extends Cloneable {

/**
 * Called immediately after an element has been removed. The remove method will block until
 * this method returns.
 * <p/>
 * Ehcache does not chech for
 * <p/>
 * As the {@link net.sf.ehcache.Element} has been removed, only what was the key of the
 * element is known.
 * <p/>
 *
 * @param cache   the cache emitting the notification
 * @param element just deleted
 */
void notifyElementRemoved(final Ehcache cache, final Element element) throws CacheException;

/**
 * Called immediately after an element has been put into the cache. The
 * {@link net.sf.ehcache.Cache#put(net.sf.ehcache.Element)} method
 * will block until this method returns.
 * <p/>
 * Implementers may wish to have access to the Element's fields, including value, so the
 * element is provided. Implementers should be careful not to modify the element. The
 * effect of any modifications is undefined.
 *
 * @param cache   the cache emitting the notification
 * @param element the element which was just put into the cache.
 */
void notifyElementPut(final Ehcache cache, final Element element) throws CacheException;

/**
 * Called immediately after an element has been put into the cache and the element already
 * existed in the cache. This is thus an update.
 * <p/>
 * The {@link net.sf.ehcache.Cache#put(net.sf.ehcache.Element)} method
 * will block until this method returns.
 * <p/>
 * Implementers may wish to have access to the Element's fields, including value, so the
 * element is provided. Implementers should be careful not to modify the element. The
 * effect of any modifications is undefined.
 *
 * @param cache   the cache emitting the notification
 * @param element the element which was just put into the cache.
 */
void notifyElementUpdated(final Ehcache cache, final Element element) throws CacheException;


/**
 * Called immediately after an element is <i>found</i> to be expired. The
 * {@link net.sf.ehcache.Cache#remove(Object)} method will block until this method returns.
```

```
     * <p/>
     * As the {@link Element} has been expired, only what was the key of the element is known.
     * <p/>
     * Elements are checked for expiry in Ehcache at the following times:
     * <ul>
     * <li>When a get request is made
     * <li>When an element is spooled to the diskStore in accordance with a MemoryStore
     * eviction policy
     * <li>In the DiskStore when the expiry thread runs, which by default is
     * {@link net.sf.ehcache.Cache#DEFAULT_EXPIRY_THREAD_INTERVAL_SECONDS}
     * </ul>
     * If an element is found to be expired, it is deleted and this method is notified.
     *
     * @param cache   the cache emitting the notification
     * @param element the element that has just expired
     *    <p/>
     *    Deadlock Warning: expiry will often come from the <code>DiskStore</code>
     *    expiry thread. It holds a lock to the DiskStorea the time the
     *    notification is sent. If the implementation of this method calls into a
     *    synchronized <code>Cache</code> method and that subsequently calls into
     *    DiskStore a deadlock will result. Accordingly implementers of this method
     *    should not call back into Cache.
     */
    void notifyElementExpired(final Ehcache cache, final Element element);


    /**
     * Give the replicator a chance to cleanup and free resources when no longer needed
     */
    void dispose();


    /**
     * Creates a clone of this listener. This method will only be called by Ehcache before a
     * cache is initialized.
     * <p/>
     * This may not be possible for listeners after they have been initialized. Implementations
     * should throw CloneNotSupportedException if they do not support clone.
     * @return a clone
     * @throws CloneNotSupportedException if the listener could not be cloned.
     */
    public Object clone() throws CloneNotSupportedException;

    }
```

The implementations need to be placed in the classpath accessible to Ehcache.

See the chapter on Classloading for details on how classloading of these classes will be done.

Chapter 27

Cache Exception Handlers

By default, most cache operations will propagate a runtime CacheException on failure. An interceptor, using a dynamic proxy, may be configured so that a CacheExceptionHandler can be configured to intercept Exceptions. Errors are not intercepted.

Caches with ExceptionHandling configured are of type Ehcache. To get the exception handling behaviour they must be referenced using `CacheManager.getEhcache()`, not `CacheManager.getCache()`, which returns the underlying undecorated cache.

CacheExceptionHandlers may be set either declaratively in the ehcache.xml configuration file or programmatically.

27.1 Declarative Configuration

Cache event listeners are configured per cache. Each cache can have at most one exception handler.

An exception handler is configured by adding a cacheExceptionHandlerFactory element as shown in the following example:

```
<cache ...>
      <cacheExceptionHandlerFactory
        class="net.sf.ehcache.exceptionhandler.CountingExceptionHandlerFactory"
        properties="logLevel=FINE"/>
</cache>
```

27.2 Implementing a CacheExceptionHandlerFactory and CacheExceptionHandler

CacheExceptionHandlerFactory is an abstract factory for creating cache exception handlers. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml

The factory class needs to be a concrete subclass of the abstract factory class CacheExceptionHandlerFactory, which is reproduced below:

```
/**
 * An abstract factory for creating <code>CacheExceptionHandler</code>s at configuration
```

```
 * time, in ehcache.xml.
 * <p/>
 * Extend to create a concrete factory
 *
 * @author <a href="mailto:gluck@gregluck.com">Greg Luck</a>
 * @version $Id: cache_exception_handlers.apt 735 2008-08-10 23:51:48Z gregluck $
 */
public abstract class CacheExceptionHandlerFactory {

/**
 * Create an <code>CacheExceptionHandler</code>
 *
 * @param properties implementation specific properties. These are configured as comma
 *                   separated name value pairs in ehcache.xml
 * @return a constructed CacheExceptionHandler
 */
public abstract CacheExceptionHandler createExceptionHandler(Properties properties);


}
```

The factory creates a concrete implementation of the CacheExceptionHandler interface, which is reproduced below:

```
/**
 * A handler which may be registered with an Ehcache, to handle exception on Cache operations.
 * <p/>
 * Handlers may be registered at configuration time in ehcache.xml, using a
 * CacheExceptionHandlerFactory, or *  set at runtime (a strategy).
 * <p/>
 * If an exception handler is registered, the default behaviour of throwing the exception
 * will not occur. The handler * method <code>onException</code> will be called. Of course, if
 * the handler decides to throw the exception, it will * propagate up through the call stack.
 * If the handler does not, it won't.
 * <p/>
 * Some common Exceptions thrown, and which therefore should be considered when implementing
 * this class are listed below:
 * <ul>
 * <li>{@link IllegalStateException} if the cache is not
 * {@link net.sf.ehcache.Status#STATUS_ALIVE}
 * <li>{@link IllegalArgumentException} if an attempt is made to put a null
 * element into a cache
 * <li>{@link net.sf.ehcache.distribution.RemoteCacheException} if an issue occurs
 *  in remote synchronous replication
 * <li>
 * <li>
 * </ul>
 *
 * @author <a href="mailto:gluck@gregluck.com">Greg Luck</a>
 * @version $Id: cache_exception_handlers.apt 735 2008-08-10 23:51:48Z gregluck $
 */
public interface CacheExceptionHandler {

/**
 * Called if an Exception occurs in a Cache method. This method is not called
 * if an <code>Error</code> occurs.
 *
```

```
 * @param Ehcache   the cache in which the Exception occurred
 * @param key       the key used in the operation, or null if the operation does not use a
 * key or the key was null
 * @param exception the exception caught
 */
void onException(Ehcache ehcache, Object key, Exception exception);
}
```

The implementations need to be placed in the classpath accessible to Ehcache.

See the chapter on Classloading for details on how classloading of these classes will be done.

27.3 Programmatic Configuration

The following example shows how to add exception handling to a cache then adding the cache back into cache manager so that all clients obtain the cache handling decoration.

```
CacheManager cacheManager = ...
Ehcache cache = cacheManger.getCache("exampleCache");
ExceptionHandler handler = new ExampleExceptionHandler(...);
cache.setCacheLoader(handler);
Ehcache proxiedCache = ExceptionHandlingDynamicCacheProxy.createProxy(cache);
cacheManager.replaceCacheWithDecoratedCache(cache, proxiedCache);
```

Chapter 28

Cache Extensions

CacheExtensions are a general purpose mechanism to allow generic extensions to a Cache.

CacheExtensions are tied into the Cache lifecycle. For that reason this interface has the lifecycle methods.

CacheExtensions are created using the CacheExtensionFactory which has a *code*createCacheCacheExtension()*/code* method which takes as a parameter a Cache and properties. It can thus call back into any public method on Cache, including, of course, the load methods.

CacheExtensions are suitable for timing services, where you want to create a timer to perform cache operations. The other way of adding Cache behaviour is to decorate a cache.

See @link net.sf.ehcache.constructs.blocking.BlockingCache for an example of how to do this.

Because a CacheExtension holds a reference to a Cache, the CacheExtension can do things such as registering a CacheEventListener or even a CacheManagerEventListener, all from within a CacheExtension, creating more opportunities for customisation.

28.1 Declarative Configuration

Cache extension are configured per cache. Each cache can have zero or more.

A CacheExtension is configured by adding a cacheExceptionHandlerFactory element as shown in the following example:

```
<cache ...>
 <cacheExtensionFactory class="com.example.FileWatchingCacheRefresherExtensionFactory"
     properties="refreshIntervalMillis=18000, loaderTimeout=3000,
                 flushPeriod=whatever, someOtherProperty=someValue ..."/>

</cache>
```

28.2 Implementing a CacheExtensionFactory and CacheExtension

CacheExtensionFactory is an abstract factory for creating cache extension. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml

The factory class needs to be a concrete subclass of the abstract factory class CacheExtensionFactory, which is reproduced below:

```
/**
 * An abstract factory for creating <code>CacheExtension</code>s. Implementers should
 * provide their own * concrete factory extending this factory. It can then be configured
 * in ehcache.xml.
 *
 * @author <a href="mailto:gluck@gregluck.com">Greg Luck</a>
 * @version $Id: cache_extensions.apt 735 2008-08-10 23:51:48Z gregluck $
 */
public abstract class CacheExtensionFactory {

/**
 * @param cache the cache this extension should hold a reference to, and to whose
 * lifecycle it should be bound.
 * @param properties implementation specific properties configured as delimiter separated
 * name value pairs in ehcache.xml
 */
public abstract CacheExtension createCacheExtension(Ehcache cache, Properties properties);

}
```

The factory creates a concrete implementation of the CacheExtension interface, which is reproduced below:

```
/**
 * This is a general purpose mechanism to allow generic extensions to a Cache.
 * <p/>
 * CacheExtensions are tied into the Cache lifecycle. For that reason this interface has the
 *  lifecycle methods.
 * <p/>
 * CacheExtensions are created using the CacheExtensionFactory which has a
 * <code>createCacheCacheExtension()</code> method which takes as a parameter a Cache and
 * properties. It can thus call back into any public method on Cache, including, of course,
 *  the load methods.
 * <p/>
 * CacheExtensions are suitable for timing services, where you want to create a timer to
 * perform cache operations. The other way of adding Cache behaviour is to decorate a cache.
 * See {@link net.sf.ehcache.constructs.blocking.BlockingCache} for an example of how to do
 * this.
 * <p/>
 * Because a CacheExtension holds a reference to a Cache, the CacheExtension can do things
 * such as registering a CacheEventListener or even a CacheManagerEventListener, all from
 * within a CacheExtension, creating more opportunities for customisation.
 *
 * @author <a href="mailto:gluck@gregluck.com">Greg Luck</a>
 * @version $Id: cache_extensions.apt 735 2008-08-10 23:51:48Z gregluck $
 */
public interface CacheExtension {

/**
 * Notifies providers to initialise themselves.
 * <p/>
 * This method is called during the Cache's initialise method after it has changed it's
 * status to alive. Cache operations are legal in this method.
 *
 * @throws CacheException
 */
void init();
```

```
/**
 * Providers may be doing all sorts of exotic things and need to be able to clean up on
 * dispose.
 * <p/>
 * Cache operations are illegal when this method is called. The cache itself is partly
 * disposed when this method is called.
 *
 * @throws CacheException
 */
void dispose() throws CacheException;

/**
 * Creates a clone of this extension. This method will only be called by Ehcache before a
 * cache is initialized.
 * <p/>
 * Implementations should throw CloneNotSupportedException if they do not support clone
 * but that will stop them from being used with defaultCache.
 *
 * @return a clone
 * @throws CloneNotSupportedException if the extension could not be cloned.
 */
public CacheExtension clone(Ehcache cache) throws CloneNotSupportedException;


/**
 * @return the status of the extension
 */
public Status getStatus();
}
```

The implementations need to be placed in the classpath accessible to ehcache.

See the chapter on Classloading for details on how class loading of these classes will be done.

28.3 Programmatic Configuration

Cache Extensions may also be programmatically added to a Cache as shown.

```
    TestCacheExtension testCacheExtension = new TestCacheExtension(cache, ...);
    testCacheExtension.init();
    cache.registerCacheExtension(testCacheExtension);
```

Chapter 29

Cache Server

29.1 Introduction

Ehcache now comes with a Cache Server, available as a WAR for most web containers, or as a standalone server. The Cache Server has two APIs: RESTful resource oriented, and SOAP. Both support clients in any programming language.

(A Note on terminology: Leonard Richardson and Sam Ruby have done a great job of clarifying the different Web Services architectures and distinguishing them from each other. We use their taxonomy in describing web services. See http://www.oreilly.com/catalog/9780596529260/.)

29.2 RESTful Web Services

Roy Fielding coined the acronym REST, denoting Representational State Transfer, in his PhD thesis.

The Ehcache implementation strictly follows the RESTful resource-oriented architecture style.

Specifically:

- The HTTP methods GET, HEAD, PUT/POST and DELETE are used to specify the method of the operation. The URI does not contain method information.
- The scoping information, used to identify the resource to perform the method on, is contained in the URI path.
- The RESTful Web Service is described by and exposes a WADL (Web Application Description Language) file. It contains the URIs you can call, and what data to pass and get back. Use the OPTIONS method to return the WADL.

  Roy is on the JSR311 expert group. JSR311 and Jersey, the reference implementation, are used to deliver RESTful web services in Ehcache server.

29.2.1 RESTFul Web Services API

The Ehcache RESTFul Web Services API exposes the singleton CacheManager, which typically has been configured in ehcache.xml or an IoC container. Multiple CacheManagers are not supported.

Resources are identified using a URI template. The value in parentheses should be substituted with a literal to specify a resource.

Response codes and response headers strictly follow HTTP conventions.

29.2.2 CacheManager Resource Operations

OPTIONS /{cache}}

Retrieves the WADL for describing the available CacheManager operations.

GET /

Lists the Caches in the CacheManager.

29.2.3 Cache Resource Operations

OPTIONS /{cache}}

Retrieves the WADL describing the available Cache operations.

HEAD /{cache}}

Retrieves the same metadata a GET would receive returned as HTTP headers. There is no body returned.

GET /{cache}

Gets a cache representation. This includes useful metadata such as the configuration and cache statistics.

PUT /{cache}

Creates a Cache using the defaultCache configuration.

DELETE / {cache}

Deletes the Cache.

29.2.4 Element Resource Operations

OPTIONS /{cache}}

Retrieves the WADL describing the available Element operations.

HEAD /{cache}/{element}

Retrieves the same metadata a GET would receive returned as HTTP headers. There is no body returned.

GET /{cache}/{element}

Gets the element value.

HEAD /{cache}/{element}

Gets the element's metadata.

PUT /{cache}/{element}

Puts an element into the Cache.

The time to live of new Elements defaults to that for the cache. This may be overridden by setting the HTTP request header `ehcacheTimeToLiveSeconds`. Values of 0 to 2147483647 are accepted. A value of 0 means eternal.

DELETE / {cache}/{element}

Deletes the element from the cache.

The resource representation for all elements is `*`. `DELETE/{cache}/* will call <<<cache.removeAll()`.

29.2.5 Resource Representations

We deal with resource representations rather than resources themselves.

Element Resource Representations

When Elements are PUT into the cache, a MIME Type should be set in the request header. The MIME Type is preserved for later use.

The new `MimeTypeByteArray` is used to store the `byte[]` and the `MimeType` in the value field of `Element`.

Some common MIME Types which are expected to be used by clients are:

text/plain	Plain text
text/xml	Extensible Markup Language. Defined in RFC 3023
application/json	JavaScript Object Notation JSON. Defined in RFC 4627
application/x-java-serialized-object	A serialized Java object

Because Ehcache is a distributed Java cache, in some configurations the Cache server may contain Java objects that arrived at the Cache server via distributed replication. In this case no MIME Type will be set and the Element will be examined to determine its MIME Type.

Because anything that can be PUT into the cache server must be Serializable, it can also be distributed in a cache cluster i.e. it will be Serializable.

29.2.6 RESTful Code Samples

These are RESTful code samples in multiple languages.

Curl Code Samples

These samples use the popular curl command line utility.

OPTIONS This example shows how calling OPTIONS causes Ehcache server to respond with the WADL for that resource

```
curl --request OPTIONS http://localhost:8080/ehcache/rest/sampleCache2/2
```

The server responds with:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<application xmlns="http://research.sun.com/wadl/2006/10">
<resources base="http://localhost:8080/ehcache/rest/">
<resource path="sampleCache2/2">
<method name="HEAD"><response><representation mediaType="
    ...
</resource>
</resources>
</application>
```

HEAD

```
curl --head  http://localhost:8080/ehcache/rest/sampleCache2/2
```

The server responds with:

```
HTTP/1.1 200 OK
X-Powered-By: Servlet/2.5
Server: GlassFish/v3
Last-Modified: Sun, 27 Jul 2008 08:08:49 GMT
ETag: "1217146129490"
Content-Type: text/plain; charset=iso-8859-1
Content-Length: 157
Date: Sun, 27 Jul 2008 08:17:09 GMT
```

PUT

```
echo "Hello World" |  curl -S -T -  http://localhost:8080/ehcache/rest/sampleCache2/3
```

The server will put `Hello World` into `sampleCache2` using key `3`.

GET

```
curl http://localhost:8080/ehcache/rest/sampleCache2/2
```

The server responds with:

```
<?xml version="1.0"?>
<oldjoke>
<burns>Say <quote>goodnight</quote>,
Gracie.</burns>
<allen><quote>Goodnight,
Gracie.</quote></allen>
<applause/>
```

Ruby Code Samples

GET

```
require 'rubygems'
require 'open-uri'
require 'rexml/document'


response = open('http://localhost:8080/ehcache/rest/sampleCache2/2')
xml = response.read
puts xml
```

The server responds with:

```
<?xml version="1.0"?>
<oldjoke>
<burns>Say <quote>goodnight</quote>,
Gracie.</burns>
<allen><quote>Goodnight,
Gracie.</quote></allen>
<applause/>
</oldjoke>
```

Python Code Samples

GET

```
import urllib2

f = urllib2.urlopen('http://localhost:8080/ehcache/rest/sampleCache2/2')
print f.read()
```

The server responds with:

```
<?xml version="1.0"?>
<oldjoke>
<burns>Say <quote>goodnight</quote>,
Gracie.</burns>
<allen><quote>Goodnight,
Gracie.</quote></allen>
<applause/>
</oldjoke>
```

Java Code Samples

Create and Get a Cache and Entry

```
package samples;

import java.io.InputStream;
import java.io.OutputStream;
import java.net.HttpURLConnection;
import java.net.URL;

/**
 * A simple example Java client which uses the built-in java.net.URLConnection.
 *
 * @author BryantR
 * @author Greg Luck
 */
public class ExampleJavaClient {

    private static String TABLE_COLUMN_BASE =
            "http://localhost:8080/ehcache/rest/tableColumn";
    private static String TABLE_COLUMN_ELEMENT =
            "http://localhost:8080/ehcache/rest/tableColumn/1";

    /**
```

```
 * Creates a new instance of EHCacheREST
 */
public ExampleJavaClient() {
}

public static void main(String[] args) {
    URL url;
    HttpURLConnection connection = null;
    InputStream is = null;
    OutputStream os = null;
    int result = 0;
    try {
        //create cache
        URL u = new URL(TABLE_COLUMN_BASE);
        HttpURLConnection urlConnection = (HttpURLConnection) u.openConnection();
        urlConnection.setRequestMethod("PUT");

        int status = urlConnection.getResponseCode();
        System.out.println("Status: " + status);
        urlConnection.disconnect();

        //get cache
        url = new URL(TABLE_COLUMN_BASE);
        connection = (HttpURLConnection) url.openConnection();
        connection.setRequestMethod("GET");
        connection.connect();
        is = connection.getInputStream();
        byte[] response1 = new byte[4096];
        result = is.read(response1);
        while (result != -1) {
            System.out.write(response1, 0, result);
            result = is.read(response1);
        }
        if (is != null) try {
            is.close();
        } catch (Exception ignore) {
        }
        System.out.println("reading cache: " + connection.getResponseCode()
                + " " + connection.getResponseMessage());
        if (connection != null) connection.disconnect();

        //create entry
        url = new URL(TABLE_COLUMN_ELEMENT);
        connection = (HttpURLConnection) url.openConnection();
        connection.setRequestProperty("Content-Type", "text/plain");
        connection.setDoOutput(true);
        connection.setRequestMethod("PUT");
        connection.connect();
        String sampleData = "Ehcache is way cool!!!";
        byte[] sampleBytes = sampleData.getBytes();
        os = connection.getOutputStream();
        os.write(sampleBytes, 0, sampleBytes.length);
        os.flush();
        System.out.println("result=" + result);
        System.out.println("creating entry: " + connection.getResponseCode()
                + " " + connection.getResponseMessage());
        if (connection != null) connection.disconnect();
```

```
            //get entry
            url = new URL(TABLE_COLUMN_ELEMENT);
            connection = (HttpURLConnection) url.openConnection();
            connection.setRequestMethod("GET");
            connection.connect();
            is = connection.getInputStream();
            byte[] response2 = new byte[4096];
            result = is.read(response2);
            while (result != -1) {
                System.out.write(response2, 0, result);
                result = is.read(response2);
            }
            if (is != null) try {
                is.close();
            } catch (Exception ignore) {
            }
            System.out.println("reading entry: " + connection.getResponseCode()
                    + " " + connection.getResponseMessage());
            if (connection != null) connection.disconnect();
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            if (os != null) try {
                os.close();
            } catch (Exception ignore) {
            }
            if (is != null) try {
                is.close();
            } catch (Exception ignore) {
            }
            if (connection != null) connection.disconnect();
        }
    }
}
```

Scala Code Samples

GET

```
import java.net.URL
import scala.io.Source.fromInputStream

object ExampleScalaGet extends Application {
  val url = new URL("http://localhost:8080/ehcache/rest/sampleCache2/2")
  fromInputStream(url.openStream).getLines.foreach(print)
}
```

Run it with:

```
scala -e ExampleScalaGet
```

The program outputs:

```
<?xml version="1.0"?>
<oldjoke>
<burns>Say <quote>goodnight</quote>,
Gracie.</burns>
```

```
     <allen><quote>Goodnight,
     Gracie.</quote></allen>
     <applause/>
```

PHP Code Samples

GET

```
   <?php

   $ch = curl_init();

   curl_setopt ($ch, CURLOPT_URL, "http://localhost:8080/ehcache/rest/sampleCache2/3");
   curl_setopt ($ch, CURLOPT_HEADER, 0);

   curl_exec ($ch);

   curl_close ($ch);
   ?>
```

The server responds with:

```
Hello Ingo
```

PUT

```
   <?php
$url = "http://localhost:8080/ehcache/rest/sampleCache2/3";
$localfile = "localfile.txt";

$fp = fopen ($localfile, "r");
$ch = curl_init();
curl_setopt($ch, CURLOPT_VERBOSE, 1);
curl_setopt($ch, CURLOPT_URL, $url);
curl_setopt($ch, CURLOPT_PUT, 1);
curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
curl_setopt($ch, CURLOPT_INFILE, $fp);
curl_setopt($ch, CURLOPT_INFILESIZE, filesize($localfile));

$http_result = curl_exec($ch);
$error = curl_error($ch);
$http_code = curl_getinfo($ch ,CURLINFO_HTTP_CODE);

curl_close($ch);
fclose($fp);

print $http_code;
print "<br /><br />$http_result";
if ($error) {
   print "<br /><br />$error";
}

?>
```

The server responds with:

```
* About to connect() to localhost port 8080 (#0)
*   Trying ::1... * connected
* Connected to localhost (::1) port 8080 (#0)
> PUT /ehcache/rest/sampleCache2/3 HTTP/1.1
Host: localhost:8080
Accept: */*
Content-Length: 11
Expect: 100-continue

< HTTP/1.1 100 Continue
< HTTP/1.1 201 Created
< Location: http://localhost:8080/ehcache/rest/sampleCache2/3
< Content-Length: 0
< Server: Jetty(6.1.10)
<
* Connection #0 to host localhost left intact
* Closing connection #0
```

29.3 Creating Massive Caches with Load Balancers and Partitioning

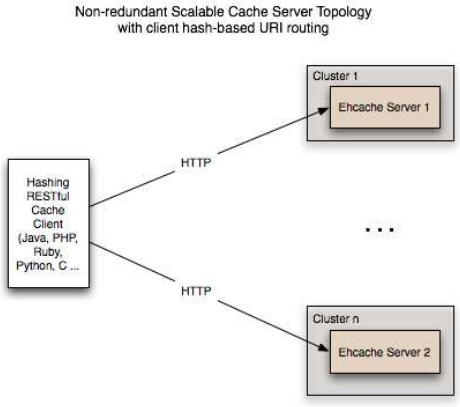
The RESTful Ehcache Server is designed to achieve massive scaling using data partitioning - all from a RESTful interface. The largest Ehcache single instances run at around 20GB in memory. The largest disk stores run at 100Gb each. Add nodes together, with cache data partitioned across them, to get larger sizes. 50 nodes at 20GB gets you to 1 Terabyte.

Two deployment choices need to be made:

- where is partitoning performed, and
- is redundancy required?

These choices can be mixed and matched with a number of different deployment topologies.

29.3.1 Non-redundant, Scalable with client hash-based routing



This topology is the simplest. It does not use a load balancer. Each node is accessed directly by the cache client using REST. No redundancy is provided.

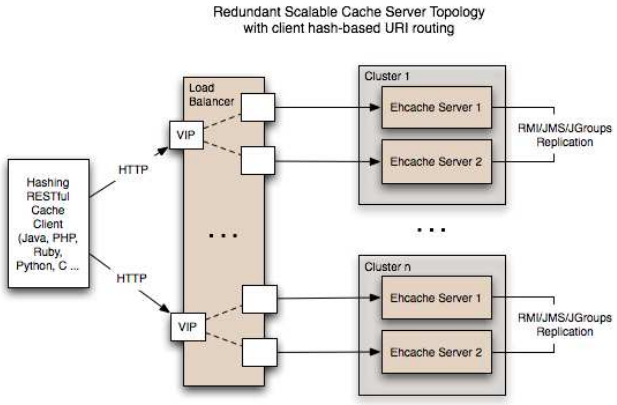
The client can be implemented in any language because it is simply a HTTP client.

It must work out a partitioning scheme. Simple key hashing, as used by memcached, is sufficient.

Here is a Java code sample:

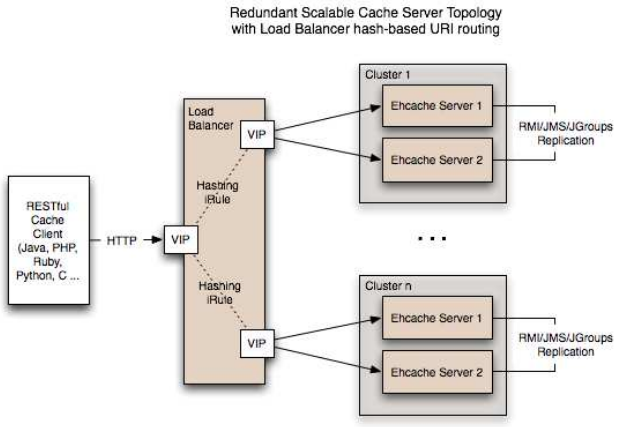
```
String[] cacheservers = new String[]{"cacheserver0.company.com", "cacheserver1.company.com",
    "cacheserver2.company.com", "cacheserver3.company.com", "cacheserver4.company.com",
    "cacheserver5.company.com"};
Object key = "123231";
int hash = Math.abs(key.hashCode());
int cacheserverIndex = hash % cacheservers.length;
String cacheserver = cacheservers[cacheserverIndex];
```

29.3.2 Redundant, Scalable with client hash-based routing



Redundancy is added as shown in the above diagram by: Replacing each node with a cluster of two nodes. One of the existing distributed caching options in Ehcache is used to form the cluster. Options in Ehcache 1.5 are RMI and JGroups-based clusters. Ehcache-1.6 will add JMS as a further option. Put each Ehcache cluster behind VIPs on a load balancer.

29.3.3 Redundant, Scalable with load balancer hash-based routing



Many content-switching load balancers support URI routing using some form of regular expressions.

So, you could optionally skip the client-side hashing to achieve partitioning in the load balancer itself.

For example:

```
/ehcache/rest/sampleCache1/[a-h]* => cluster1
/ehcache/rest/sampleCache1/[i-z]* => cluster2
```

Things get much more sophisticated with F5 load balancers, which let you create iRules in the TCL language. So rather than regular expression URI routing, you could implement key hashing-based URI routing. Remember in Ehcache's RESTful server, the key forms the last part of the URI. e.g. In the URI http://cacheserver.company.com/ehcache/rest/sampleCache1/3432 , 3432 is the key.

You hash using the last part of the URI.

See http://devcentral.f5.com/Default.aspx?tabid=63&PageID=153&ArticleID=135&articleType=ArticleView for how to implment a URI hashing iRule on F5 load balancers.

29.4 W3C (SOAP) Web Services

The W3C (http://www.w3.org/ is a standards body that defines Web Services as

```
The World Wide Web is more and more used for application to application communication.
The programmatic interfaces made available are referred to as Web services.
```

They provide a set of recommendations for achieving this. See http://www.w3.org/2002/ws/.

An interoperability organisation, WS-I http://www.ws-i.org/, seeks to achieve interoperabilty between W3C Web Services. The W3C specifications for SOAP and WSDL are required to meet the WS-I definition.

Ehcache is using Glassfish's libraries to provide it's W3C web services. The project known as Metro follows the WS-I definition.

Finally, OASIS (http://oasis-open.org), defines a Web Services Security specification for SOAP: WS-Security. The current version is 1.1. It provides three main security mechanisms: ability to send security tokens as part of a message, message integrity, and message confidentiality.

Ehcache's W3C Web Services support the stricter WS-I definition and use the SOAP and WSDL specfications.

Specifically:

- The method of operation is in the entity-body of the SOAP envelope and a HTTP header. POST is always used as the HTTP method.
- The scoping information, used to identify the resource to perform the method on, is contained in the SOAP entity-body. The URI path is always the same for a given Web Service - it is the service "endpoint".
- The Web Service is described by and exposes a WSDL (Web Services Description Language) file. It contains the methods, their arguments and what data types are used.
- The WS-Security SOAP extensions are supported

29.4.1 W3C Web Services API

The Ehcache RESTFul Web Services API exposes the singleton CacheManager, which typically has been configured in ehcache.xml or an IoC container. Multiple CacheManagers are not supported.

The API definition is as follows:

- WSDL - EhcacheWebServiceEndpointService.wsdl
- Types - EhcacheWebServiceEndpointService_schema1.xsd

29.4.2 Security

By default no security is configured. Because it is simply a Servlet 2.5 web application, it can be secured in all the usual ways by configuration in the web.xml.

In addition the cache server supports the use of XWSS 3.0 to secure the Web Service. See https://xwss.dev.java.net/. All required libraries are packaged in the war for XWSS 3.0.

A sample, commented out server_security_config.xml is provided in the WEB-INF directory. XWSS automatically looks for this configuration file.

A simple example, based on an XWSS example, `net.sf.ehcache.server.soap.SecurityEnvironmentHandler`, which looks for a password in a System property for a given username is included. This is not recommended for production use but is handy when you are getting started with XWSS.

To use XWSS:

Add configuration in accordance with XWSS to the server_security_config.xml file. Create a class which implements the `CallbackHandler` interface and provide its fully qualified path in the `SecurityEnvironmentHandler` element.

The integration test `EhcacheWebServiceEndpoint` test shows how to use the XWSS client side. On the client side, configuration must be provided in a file called `client_security_config.xml` must be in the root of the classpath.

To add client credentials into the SOAP request do:

```
cacheService = new EhcacheWebServiceEndpointService().getEhcacheWebServiceEndpointPort();
//add security credentials
((BindingProvider)cacheService).getRequestContext().put(BindingProvider.USERNAME_PROPERTY,
 "Ron");
((BindingProvider)cacheService).getRequestContext().put(BindingProvider.PASSWORD_PROPERTY,
 "noR");
String result = cacheService.ping();
```

29.5 Requirements

29.5.1 Java

Java 5 or 6

29.5.2 Web Container (WAR packaged version only)

The standalone server comes with its own embedded Glassfish web container.

The web container must support the Servlet 2.5 specification.

The following web container configuration have been tested:

- Glassfish V2/V3
- Tomcat 6
- Jetty 6

29.6 Downloading

The server is available as follows:

29.6.1 Sourceforge

Download here.

There are two tarball archives in tar.gz format:

- ehcache-server - this contains the WAR file which must be deployed in your own web container.
- ehcache-standalone-server - this contains a complete standalone directory structure with an embedded Glassfish V3 web container together with shell scripts for starting and stopping.

29.6.2 Maven

The Ehcache Server is in the central Maven repository packaged as type *war*. Use the following Maven pom snippet:

```
<dependency>
     <groupId>net.sf.ehcache</groupId>
     <artifactId>ehcache-server</artifactId>
     <version>enter_version_here</version>
     <type>war</type>
</dependency>
```

It is also available as a jaronly version, which makes it easier to embed. This version excludes all META-INF and WEB-INF configuration files, and also excludes the ehcache.xml. You need to provide these in your maven project.

```
 <dependency>
     <groupId>net.sf.ehcache</groupId>
     <artifactId>ehcache-server</artifactId>
     <version>enter_version_here</version>
     <type>jar</type>
     <classifier>jaronly</classifier>
</dependency>
```

29.7 Installation

29.7.1 Installing the WAR

Use your Web Container's instructions to install the WAR or include the WAR in your project with Maven's war plugin.

Web Container specific configuration is provided in the WAR as follows:

- sun-web.xml - Glassfish V2/V3 configuration
- jetty-web.xml - Jetty V5/V6 configuration

  Tomcat V6 passes all integration tests. It does not require a specific configuration.

29.7.2 Configuring the Web Application

Expand the WAR.

Edit the web.xml.

Disabling the RESTful Web Service

Comment out the RESTful web service section.

Disabling the SOAP Web Service

Comment out the RESTful web service section.

Configuring Caches

The ehcache.xml configuration file is located in WEB-INF/classes/ehcache.xml.

Follow the instructions in this config file, or the core Ehcache instructions to configure.

SOAP Web Service Security

29.8 Installing the Standalone Server

The WAR also comes packaged with a standalone server, based on Glassfish V3 Embedded.

The quick start is:

- Untar the download
- bin/start.sh to start. By default it will listen on port 8080, with JMX listening on port 8081, will have both RESTful and SOAP web services enabled, and will use a sample Ehcache configuration from the WAR module.
- bin/stop.sh to stop

29.8.1 Configuring the Standalone Server

Configuration is by editing the war/web.xml file as per the instructions for the WAR packaging.

29.8.2 Starting and Stopping the Standalone Server

Using Commons Daemon jsvc

jsvc creates a daemon which returns once the service is started. jsvc works on all common Unix-based operating systems including Linux, Solaris and Mac OS X.

It creates a pid file in the pid directory.

This is a Unix shell script that starts the server as a daemon.

To use jsvc you must install the native binary jsvc from the Apache Commons Daemon project. The source for this is distributed in the bin directory as jsvc.tar.gz. Untar it and follow the instructions for building it or download a binary from the Commons Daemon project.

Convenience shell scripts are provided as follows:

start - `daemon_start.sh`

stop - `daemon_stop.sh`

jsvc is designed to integrate with Unix System 5 initialization scripts. (/etc/rc.d)

You can also send Unix signals to it. Meaningful ones for the Ehcache Standalone Server are:

No	Meaning
1	HUP
2	INT
9	KILL
15	TERM

Executable jar

The server is also packaged as an executable jar for developer convenience which will work on all operating systems.

A convenience shell script is provided as follows:

start - startup.sh

From the bin directory you can also invoke the following command directly:

```
unix    - java -jar ../lib/ehcache-standalone-server-0.7.jar 8080 ../war
windows - java -jar ..\lib\ehcache-standalone-server-0.7.jar 8080 ..\war
```

29.9 Monitoring

The CacheServer registers Ehcache MBeans with the platform MBeanServer.

Remote monitoring of the MBeanServer is the responsibility of the Web Container or Application Server vendor.

For example, some instructions for Tomcat are here: https://wiki.internet2.edu/confluence/display/CPD/Monitoring+Tomcat+wi

See your Web Container documentation for how to do this for your web container.

29.9.1 Remotely Monitoring the Standalone Server with JMX

The standalone server automatically exposes the MBeanServer on a port 1 higher than the HTTP listening port.

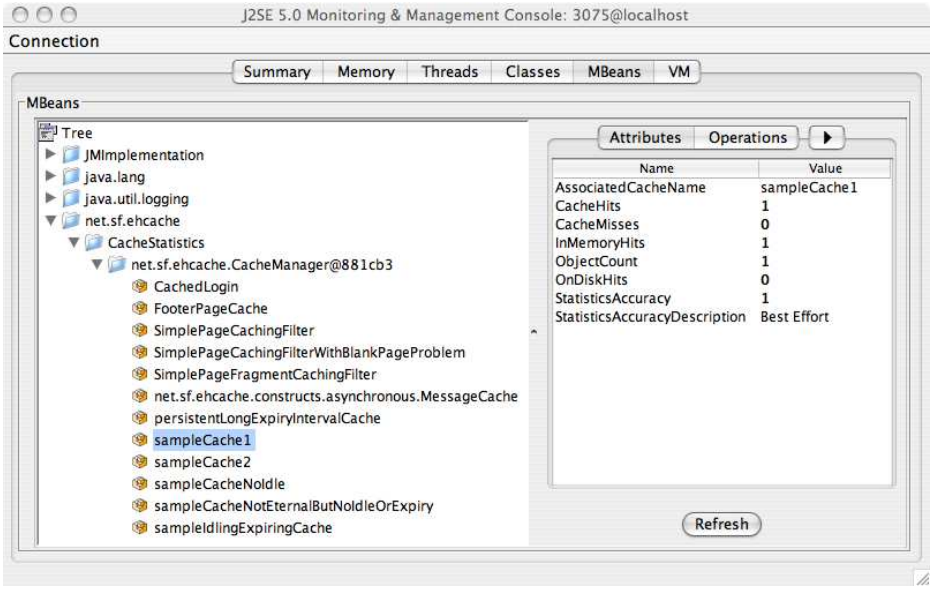
To connect with `JConsole` simply fire up JConsole, enter the host in the Remote field and portIn the above example that is

```
192.168.1.108:8686
```

Then click `Connect`.

To see the Ehcache MBeans, click on the `Mbeans` tab and expand the `net.sf.ehcache` tree node.

You will see something like the following.



CacheStatistics MBeans in JConsole

Of course, from there you can hook the Cache Server up to your monitoring tool of choice. See the chapter on JMX Management and Monitoring for more information.

Chapter 30

Hibernate Caching

Note these instructions are for Hibernate 3.1. Go to Guide for Version 1.1 for older instructions on how to use Hibernate 2.1.

Ehcache easily integrates with the Hibernate Object/Relational persistence and query service. Gavin King, the maintainer of Hibernate, is also a committer to the Ehcache project. This ensures Ehcache will remain a first class cache for Hibernate.

Since Hibernate 2.1, Ehcache has been the default cache, for Hibernate.

The net.sf.ehcache.hibernate package provides classes integrating Ehcache with Hibernate. Hibernate is an application of ehcache. Ehcache is also widely used a general-purpose Java cache.

To use Ehcache with Hibernate do the following:

- Ensure Ehcache is enabled in the Hibernate configuration.
- Add the cache element to the Hibernate mapping file, either manually, or via hibernatedoclet for each Domain Object you wish to cache.
- Add the cache element to the Hibernate mapping file, either manually, or via hibernatedoclet for each Domain Object collection you wish to cache.
- Add the cache element to the Hibernate mapping file, either manually, or via hibernatedoclet for each Hibernate query you wish to cache.
- Create a cache element in ehcache.xml

Each of these steps is illustrated using a fictional Country Domain Object.

For more about cache configuration in Hibernate see the Hibernate documentation. Parts of this chapter are drawn from Hibernate documentation and source code comments.

They are reproduced here for convenience in using ehcache.

30.1 Setting Ehcache as the cache provider

30.1.1 Using one of the two Ehcache providers from the Ehcache project

To ensure Ehcache is enabled, verify that the hibernate.cache.provider_class property is set to one of the following in the Hibernate configuration file, either hibernate.cfg.xml or hibernate.properties. The format given is for hibernate.cfg.xml.

```
net.sf.ehcache.hibernate.EhCacheProvider
```

for instance creation, or

```
net.sf.ehcache.hibernate.SingletonEhCacheProvider
```

to force Hibernate to use a singleton of Ehcache CacheManager.

30.1.2 Using multiple Hibernate instances

Each instance of Hibernate will need it's own instance of ehcache's CacheManager.

To do this use the following configuration, which a unique `configurationResourceName` per Hibernate instance.

```
hibernate.cache.provider_class=net.sf.ehcache.hibernate.EhCacheProvider
net.sf.ehcache.configurationResourceName=/name_of_ehcache.xml
```

The meaning of the properties is as follows:

hibernate.cache.provider_class - The fully qualified class name of the cache provider

net.sf.ehcache.configurationResourceName - The name of a configuration resource to use.

The resource is searched for in the root of the classpath. It is needed to support multiple CacheManagers in the same VM. It tells Hibernate which configuration to use. An example might be "ehcache-2.xml".

30.1.3 Using the Hibernate Ehcache provider

To use the one from the Hibernate project:

```
hibernate.cache.provider_class=org.hibernate.cache.EhCacheProvider
hibernate.cache.provider_configuration_file_resource_path=/name_of_configuration_resource
```

30.1.4 Programmatic setting of the Hibernate Cache Provider

The provider can also be set programmatically in Hibernate using Configuration.setProperty("hibernate.cache.provider_class", "net.sf.ehcache.hibernate.EhCacheProvider").

30.2 Hibernate Mapping Files

In Hibernate, each domain object requires a mapping file.

For example to enable cache entries for the domain object com.somecompany.someproject.domain.Country there would be a mapping file something like the following:

```
<hibernate-mapping>

<class
```

```
    name="com.somecompany.someproject.domain.Country"
    table="ut_Countries"
    dynamic-update="false"
    dynamic-insert="false"
>
...
</hibernate-mapping>
```

To enable caching, add the following element.

```
<cache usage="read-write|nonstrict-read-write|read-only" />
```

e.g.

```
<cache usage="read-write" />
```

30.2.1 read-write

Caches data that is sometimes updated while maintaining the semantics of "read committed" isolation level. If the database is set to "repeatable read", this concurrency strategy almost maintains the semantics. Repeatable read isolation is compromised in the case of concurrent writes.

This is an "asynchronous" concurrency strategy.

30.2.2 nonstrict-read-write

Caches data that is sometimes updated without ever locking the cache. If concurrent access to an item is possible, this concurrency strategy makes no guarantee that the item returned from the cache is the latest version available in the database. Configure your cache timeout accordingly! This is an "asynchronous" concurrency strategy.

This policy is the fastest. It does not use synchronized methods whereas read-write and read-only both do.

30.2.3 read-only

Caches data that is never updated.

30.3 Hibernate Doclet

Hibernate Doclet, part of the XDoclet project, can be used to generate Hibernate mapping files from markup in JavaDoc comments.

Following is an example of a Class level JavaDoc which configures a read-write cache for the Country Domain Object:

```
/**
* A Country Domain Object
*
```

```
* @hibernate.class table="COUNTRY"
* @hibernate.cache usage="read-write"
*/
public class Country implements Serializable
{
  ...
}
```

The @hibernate.cache usage tag should be set to one of read-write, nonstrict-read-write and read-only.

30.4 Configuration with ehcache.xml

Because ehcache.xml has a defaultCache, caches will always be created when required by Hibernate. However more control can be exerted by specifying a configuration per cache, based on its name.

In particular, because Hibernate caches are populated from databases, there is potential for them to get very large. This can be controlled by capping their maxElementsInMemory and specifying whether to overflowToDisk beyond that.

Hibernate uses a specific convention for the naming of caches of Domain Objects, Collections, and Queries.

30.4.1 Domain Objects

Hibernate creates caches named after the fully qualified name of Domain Objects.

So, for example to create a cache for com.somecompany.someproject.domain.Country create a cache configuration entry similar to the following in ehcache.xml.

```
<cache
    name="com.somecompany.someproject.domain.Country"
    maxElementsInMemory="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="true"
/>
```

30.4.2 Hibernate

CacheConcurrencyStrategy read-write, nonstrict-read-write and read-only policies apply to Domain Objects.

30.4.3 Collections

Hibernate creates collection caches named after the fully qualified name of the Domain Object followed by "." followed by the collection field name.

For example, a Country domain object has a set of advancedSearchFacilities. The Hibernate doclet for the accessor looks like:

```
 /**
  * Returns the advanced search facilities that should appear for this country.
  * @hibernate.set cascade="all" inverse="true"
  * @hibernate.collection-key column="COUNTRY_ID"
```

```
  * @hibernate.collection-one-to-many class="com.wotif.jaguar.domain.AdvancedSearchFacility"
  * @hibernate.cache usage="read-write"
  */
 public Set getAdvancedSearchFacilities() {
    return advancedSearchFacilities;
 }
```

You need an additional cache configured for the set. The ehcache.xml configuration looks like:

```
<cache name="com.somecompany.someproject.domain.Country"
     maxElementsInMemory="50"
     eternal="false"
     timeToLiveSeconds="600"
     overflowToDisk="true"
 />
<cache
     name="com.somecompany.someproject.Country.advancedSearchFacilities"
     maxElementsInMemory="450"
     eternal="false"
     timeToLiveSeconds="600"
     overflowToDisk="true"
 />
```

30.4.4 Hibernate CacheConcurrencyStrategy

read-write, nonstrict-read-write and read-only policies apply to Domain Object collections.

30.4.5 Queries

Hibernate allows the caching of query results using two caches.

"net.sf.hibernate.cache.StandardQueryCache" and "net.sf.hibernate.cache.UpdateTimestampsCache" in versions 2.1 to 3.1 and "org.hibernate.cache.StandardQueryCache" and "org.hibernate.cache.UpdateTimestampsCache" in version 3.2. are always used.

30.4.6 StandardQueryCache

This cache is used if you use a query cache without setting a name. A typical ehcache.xml configuration is:

```
<cache
    name="org.hibernate.cache.StandardQueryCache"
    maxElementsInMemory="5"
    eternal="false"
    timeToLiveSeconds="120"
    overflowToDisk="true"/>
```

30.4.7 UpdateTimestampsCache

Tracks the timestamps of the most recent updates to particular tables. It is important that the cache timeout of the underlying cache implementation be set to a higher value than the timeouts of any of the query caches. In fact, it is recommend that the the underlying cache not be configured for expiry at all.

A typical ehcache.xml configuration is:

```
<cache
    name="org.hibernate.cache.UpdateTimestampsCache"
    maxElementsInMemory="5000"
    eternal="true"
    overflowToDisk="true"/>
```

30.4.8 Named Query Caches

In addition, a QueryCache can be given a specific name in Hibernate using Query.setCacheRegion(String name). The name of the cache in ehcache.xml is then the name given in that method. The name can be whatever you want, but by convention you should use "query." followed by a descriptive name.

E.g.

```
<cache name="query.AdministrativeAreasPerCountry"
    maxElementsInMemory="5"
    eternal="false"
    timeToLiveSeconds="86400"
    overflowToDisk="true"/>
```

30.4.9 Using Query Caches

For example, let's say we have a common query running against the Country Domain.

Code to use a query cache follows:

```
public List getStreetTypes(final Country country) throws HibernateException {
 final Session session = createSession();
 try {
        final Query query = session.createQuery(

        "select st.id, st.name"
        + " from StreetType st "
        + " where st.country.id = :countryId "
        + " order by st.sortOrder desc, st.name");
        query.setLong("countryId", country.getId().longValue());
        query.setCacheable(true);
        query.setCacheRegion("query.StreetTypes");
        return query.list();
     } finally {
        session.close();
     }
 }
```

The `query.setCacheable(true)` line caches the query.

The `query.setCacheRegion("query.StreetTypes")` line sets the name of the Query Cache.

30.4.10 Hibernate CacheConcurrencyStrategy

None of read-write, nonstrict-read-write and read-only policies apply to Domain Objects. Cache policies are not configurable for query cache. They act like a non-locking read only cache.

30.5 Hibernate Caching Performance Tips

To get the most out of Ehcache with Hibernate, Hibernate's use of it's in-process cache is important to understand.

30.5.1 In-Process Cache

From Hibernate's point of view, Ehcache is an in-process cache. Cached objects are accessible across different sessions. They are common to the Java process.

30.5.2 Object Id

Hibernate identifies cached objects via an object id. This is normally the primary key of a database row.

30.5.3 Session.load

Session.load will always try to use the cache.

30.5.4 Session.find and Query.find

Session.find does not use the cache for the primary object. Hibernate will try to use the cache for any associated objects. Session.find does however cause the cache to be populated.

Query.find works in exactly the same way.

Use these where the chance of getting a cache hit is low.

30.5.5 Session.iterate and Query.iterate

Session.iterate always uses the cache for the primary object and any associated objects.

Query.iterate works in exactly the same way.

Use these where the chance of getting a cache hit is high.

30.6 Hibernate FAQ

30.6.1 TBC OpenJPA Caching Provider

Ehcache easily integrates with the OpenJPA persistence framework.

30.7 Installing

To use it, add a Maven dependency for ehcache-openjpa.

```
<groupId>net.sf.ehcache</groupId>
<artifactId>ehcache-openjpa</artifactId>
<version>0.1</version>
```

or download from downloads.

30.8 Configuration

Set OpenJPA's `openjpa.QueryCache` to `ehcache` and `openjpa.DataCacheManager` to `ehcache`. That's it!

See http://openjpa.apache.org/builds/1.0.2/apache-openjpa-1.0.2/docs/manual/ref_guide_caching.html for more on caching in OpenJPA.

Chapter 31

JSR107 (JCACHE) Support

31.1 JSR107 Implementation

Ehcache provides a preview implementation of JSR107 via the `net.sf.cache.jcache` package.

WARNING: JSR107 is still being drafted with the Ehcache maintainer as Co Spec Lead. This package will continue to change until JSR107 is finalised. No attempt will be made to maintain backward compatiblity between versions of the package. It is therefore recommended to use Ehcache's proprietary API directly.

31.2 Using JCACHE

31.2.1 Creating JCaches

JCaches can be created in two ways:

- as an Ehcache decorator
- from JCache's CacheManager

Creating a JCache using an Ehcache decorator

`manager` in the following sample is an net.sf.ehcache.CacheManager

```
net.sf.jsr107cache.Cache cache = new JCache(manager.getCache("sampleCacheNoIdle"), null);
```

Creating a JCache from an existing Cache in Ehcache's CacheManager

This is the recommended way of using JCache. Caches can be configured in ehcache.xml and wrapped as JCaches with the getJCache method of CacheManager.

`manager` in the following sample is an net.sf.ehcache.CacheManager

```
net.sf.jsr107cache.Cache cache = manager.getJCache("sampleCacheNoIdle");
```

Adding a JCache to Ehcache's CacheManager

`manager` in the following sample is an net.sf.ehcache.CacheManager

```
Ehcache Ehcache = new net.sf.ehcache.Cache(...);
net.sf.jsr107cache.Cache cache = new JCache(ehcache);
manager.addJCache(cache);
```

Creating a JCache using the JCache CacheManager

Warning: The JCache CacheManager is unworkable and will very likely be dropped in the final JCache as a Class. It will likely be replaced with a CacheManager interface.

The JCache CacheManager only works as a singleton. You obtain it with `getInstance`

The CacheManager uses a CacheFactory to create Caches. The CacheFactory is specified using the Service Provider Interface mechanism introduced in JDK1.3.

The factory is specified in the `META-INF/services/net.sf.jsr107cache.CacheFactory` resource file. This would normally be packaged in a jar. The default value for the Ehcache implementation is `net.sf.ehcache.jcache.JCacheFactory`

The configuration for a cache is assembled as a map of properties. Valid properties can be found in the JavaDoc for the JCacheFactory.createCache() method.

See the following full example.

```
CacheManager singletonManager = CacheManager.getInstance();
CacheFactory cacheFactory = singletonManager.getCacheFactory();
assertNotNull(cacheFactory);

Map config = new HashMap();
config.put("name", "test");
config.put("maxElementsInMemory", "10");
config.put("memoryStoreEvictionPolicy", "LFU");
config.put("overflowToDisk", "true");
config.put("eternal", "false");
config.put("timeToLiveSeconds", "5");
config.put("timeToIdleSeconds", "5");
config.put("diskPersistent", "false");
config.put("diskExpiryThreadIntervalSeconds", "120");

Cache cache = cacheFactory.createCache(config);
singletonManager.registerCache("test", cache);
```

31.2.2 Getting a JCache

Once a cache is registered in CacheManager, you get it from there.

The following example shows how to get a Cache.

```
manager = CacheManager.getInstance();
Ehcache Ehcache = new net.sf.ehcache.Cache("UseCache", 10,
MemoryStoreEvictionPolicy.LFU,
 false, null, false, 10, 10, false, 60, null);
manager.registerCache("test", new JCache(ehcache, null));
Cache cache = manager.getCache("test");
```

31.2.3 Using a JCache

The JavaDoc is the best place to learn how to use a JCache.

The main point to remember is that JCache implements Map and that is the best way to think about it.

JCache also has some interesting asynchronous methods such as `load` and `loadAll` which can be used to preload the JCache.

31.3 Problems and Limitations in the early draft of JSR107

If you are used to the richer API that Ehcache provides, you need to be aware of some problems and limitations in the draft specification.

You can generally work around these by getting the `Ehcache` backing cache. You can then access the extra features available in ehcache.

Of course the biggest limitation is that JSR107 (as of Augut 2007) is a long way from final.

```
/**
 * Gets the backing Ehcache
 */
public Ehcache getBackingCache() {
    return cache;
}
```

The following is both a critique of JCache and notes on the Ehcache implementation. As a member of the JSR107 Expert Group these notes are also intended to be used to improve the specification.

31.3.1 net.sf.jsr107cache.CacheManager

CacheManager does not have the following features:

- shutdown the CacheManager - there is no way to free resources or persist. Implementations may utilise a shutdown hook, but that does not work for application server redeployments, where a shutdown listener must be used.
- List caches in the CacheManager. There is no way to iterate over, or get a list of caches.
- remove caches from the CacheManager - once its there it is there until JVM shutdown. This does not work well for dynamic creation, destruction and recreation of caches.
- CacheManager does not provide a standard way to configure caches. A Map can be populated with properties and passed to the factory, but there is no way a configuration file can be configured. This should be standardised so that declarative cache configuration, rather than programmatic, can be achieved.

31.3.2 net.sf.jsr107cache.CacheFactory

A property is specified in the resource services/net.sf.jsr107cache.CacheFactory for a CacheFactory.

The factory then resolves the CacheManager which must be a singleton.

A singleton CacheManager works in simple scenarios. But there are many where you want multiple CacheManagers in an application. Ehcache supports both singleton creation semantics and instances and defines the way both can coexist.

The singleton CacheManager is a limitation of the specification.

(Alternatives: Some form of annotation and injection scheme)

Pending a final JSR107 implementation, the Ehcache configuration mechanism is used to create JCaches from ehcache.xml config.

31.3.3 net.sf.jsr107cache.Cache

- The spec is silent on whether a Cache can be used in the absence of a CacheManager. Requiring a CacheManager makes a central place where concerns affecting all caches can be managed, not just a way of looking them up. For example, configuration for persistence and distribution.
- Cache does not have a lifecycle. There is no startup and no shutdown. There is no way, other than a shutdown hook, to free resources or perform persistence operations. Once again this will not work for redeployment of applications in an app server.
- There is no mechanism for creating a new cache from a default configuration such as a `public void registerCache(String cacheName)` on CacheManager. This feature is considered indispensable by frameworks such as Hibernate.
- Cache does not have a `getName()` method. A cache has a name; that is how it is retrieved from the CacheManager. But it does not know its own name. This forces API users to keep track of the name themselves for reporting exceptions and log messages.
- Cache does not support setting a TTL override on a put. e.g. `put(Object key, Object value, long timeToLive)`. This is a useful feature.
- The spec is silent on whether the cache accepts null keys and elements. Ehcache allows all implementations. i.e.

```
cache.put(null, null);
assertNull(cache.get(null));
cache.put(null, "value");
assertEquals("value", cache.get(null));
cache.put("key", null);
assertEquals(null, cache.get("key"));
```

  `null` is effectively a valid key. However because `null` id not an instance of `Serializable` null-keyed entries will be limited to in-process memory.
- The `load(Object key)`, `loadAll(Collection keys)` and `getAll(Collection collection)` methods specify in the javadoc that they should be asynchronous. Now, most load methods work off a database or some other relatively slow resource (otherwise there would be no need to have a cache in the first place).

  To avoid running out of threads, these load requests need to be queued and use a finite number of threads. The Ehcache implementation does that. However, due to the lack of lifecycle management, there is no immediate way to free resources such as thread pools.
- The `load` method ignores a request if the element is already loaded in for that key.
- `get` and `getAll` are inconsistent. `getAll` throws CacheException, but `get` does not. They both should.

```
/**
 * Returns a collection view of the values contained in this map.  The
 * collection is backed by the map, so changes to the map are reflected in
 * the collection, and vice-versa.  If the map is modified while an
 * iteration over the collection is in progress (except through the
 * iterator's own <tt>remove</tt> operation), the results of the
 * iteration are undefined.  The collection supports element removal,
 * which removes the corresponding mapping from the map, via the
 * <tt>Iterator.remove</tt>, <tt>Collection.remove</tt>,
 * <tt>removeAll</tt>, <tt>retainAll</tt> and <tt>clear</tt> operations.
```

```
     * It does not support the add or <tt>addAll</tt> operations.
     * <p/>
     *
     * @return a collection view of the values contained in this map.
     */
    public Collection values() {
```

It is not practical or desirable to support this contract. Ehcache has multiple maps for storage of elements so there is no single backing map. Allowing changes to propagate from a change in the collection maps would break the public interface of the cache and introduce subtle threading issues.

The Ehcache implementation returns a new collection which is not connected to internal structures in ehcache.

31.3.4 net.sf.jsr107cache.CacheEntry

- getHits() returns int. It should return long because production cache systems have entries hit more than Integer.MAX_VALUE times.

  Once you get to Integer.MAX_VALUE the counter rolls over. See the following test:

```
    @Test public void testIntOverflow() {
        long value = Integer.MAX_VALUE;
        value += Integer.MAX_VALUE;
        value += 5;
        LOG.log(Level.INFO, "" + value);
        int valueAsInt = (int) value;
        LOG.log(Level.INFO, "" + valueAsInt);
        assertEquals(3, valueAsInt);
    }
```

- `getCost()` requirs the CacheEntry to know where it is. If it is in a DiskStore then its cost of retrieval could be higher than if it is in heap memory. Ehcache elements do not have this concept, and it is not implemented. i.e. getCost always returns 0. Also, if it is in the DiskStore, when you retrieve it is in then in the MemoryStore and its retrieval cost is a lot lower. I do not see the point of this method.

- `getLastUpdateTime()` is the time the last "update was made". JCACHE does not support updates, only puts

31.3.5 net.sf.jsr107cache.CacheStatistics

- getObjectCount() is a strange name. How about getSize()? If a cache entry is an object graph each entry will have more than one "object" in it. But the cache size is what is really meant, so why not call it that?

- Once again `getCacheHits` and `getCacheMisses` should be longs.

```
public interface CacheStatistics {

    public static final int STATISTICS_ACCURACY_NONE = 0;
    public static final int STATISTICS_ACCURACY_BEST_EFFORT = 1;
    public static final int STATISTICS_ACCURACY_GUARANTEED = 2;

    public int getStatisticsAccuracy();
```

```
        public int getObjectCount();

        public int getCacheHits();

        public int getCacheMisses();

        public void clearStatistics();
```

- There is a `getStatisticsAccuracy()` method but not a corresponding setStatisticsAccuracy method on Cache, so that you can alter the accuracy of the Statistics returned.

  Ehcache supports this behaviour.

- There is no method to estimate memory use of a cache. Ehcache serializes each Element to a byte[] one at a time and adds the serialized sizes up. Not perfect but better than nothing and works on older JDKs.

- CacheStatistics is obtained using `cache.getCacheStatistics()` It then has getters for values. In this way it feels like a value object. The Ehcache implementation is Serializable so that it can act as a DTO. However it also has a clearStatistics() method. This method clear counters on the Cache. Clearly CacheStatistics must hold a reference to Cache to enable this to happen.

  But what if you are really using it as a value object and have serialized it? The Ehcache implementation marks the Cache reference as `transient`. If clearStatistics() is called when the cache reference is no longer there, an IllegalStateException is thrown.

  A much better solution would be to move clearStatistics() to Cache.

31.3.6 net.sf.jsr107cache.CacheListener

```
/**
 *  Interface describing various events that can happen as elements are added to
 *  or removed from a cache
 */
public interface CacheListener {
    /** Triggered when a cache mapping is created due to the cache loader being consulted */
    public void onLoad(Object key);

    /** Triggered when a cache mapping is created due to calling Cache.put() */
    public void onPut(Object key);

    /** Triggered when a cache mapping is removed due to eviction */
    public void onEvict(Object key);

    /** Triggered when a cache mapping is removed due to calling Cache.remove() */
    public void onRemove(Object key);

    public void onClear();
}
```

- Listeners often need not just the key, but the cache Entry itself. This listener interface is extremely limiting.

- There is no onUpdate notification method. These are mapped to JCACHE's onPut notification.

- There is no onExpired notification method. These are mapped to JCACHE's onEvict notification.

31.3.7 net.sf.jsr107cache.CacheLoader

- JCache can store null values against a key. In this case, on JCache#`get` or `getAll` should an implementation attempt to load these values again? They might have been null in the system the CacheLoader loads from, but now aren't. The Ehcache implementation will still return nulls, which is probably the correct behaviour. This point should be clarified.

31.4 Other Areas

31.4.1 JMX

JSR107 is silent on JMX which has been included in the JDK since 1.5.

Chapter 32

Glassfish HowTo & FAQ

The maintainer uses Ehcache in production with Glassfish. This chapter provides a Glassfish HOWTO.

32.1 Versions

Ehcache is used in production with Glassfish V1 and V2.

32.2 HowTo

32.2.1 HowTo Get A Sample Application using Ehcache packaged and Deployed to Glassfish

Ehcache comes with a sample web application which is used to test the page caching. The page caching is the only area that is sensitive to the Application Server. For Hibernate and general caching, it is only dependent on your Java version.

From a checkout of Ehcache run the following from the `core` directory:

You need:

- a Glassfish installation.
- a `GLASSFISH_HOME` environment variable defined.
- `$GLASSFISH_HOME/bin` added to your `PATH`

  Do the following:

```
# To package and deploy to domain1:
ant deploy-default-web-app-glassfish

# Start domain1:
asadmin start-domain domain1

# Stop domain1:
asadmin stop-domain domain1

# Overwrite the config with our own which changes the port to 9080:
ant glassfish-configuration
```

```
# Start domain1:
asadmin start-domain domain1
```

You can then run the web tests in the web package or point your browser at `http://localhost:9080`.

See for a quickstart to Glassfish.

32.2.2 How to get around the EJB Container restrictions on thread creation

When Ehcache is running in the EJB Container, for example for Hibernate caching, it is in technical breach of the EJB rules. Some app servers let you override this restriction.

I am not exactly sure how this in done in Glassfish. For a number of reasons we run Glassfish without the Security Manager, and we do not have any issues.

In domain.xml ensure that the following is not included.

```
<jvm-options>-Djava.security.manager</jvm-options>
```

32.2.3 How To Enable Read Behind Page Caching in Glassfish

The read behind page caching feature requires that HTTP1.1 keepalives are turned off.

To do this in Glassfish:

```
Not sure if this is possible in Glassfish. Not in the documentation
```

32.3 Glassfish FAQ

32.3.1 Ehcache page caching versions below Ehcache 1.3 get an IllegalStateException in Glassfish.

This issue was fixed in Ehcache 1.3.

32.3.2 I get a `Could not ungzip. Heartbeat will not be working. Not in GZIP format` reported from PayloadUtil exception when using Ehcache with my Glassfish cluster. Why?

Ehcache and Glassfish clustering have nothing to do with each other. The error is caused because Ehcache has received a multicast message from the Glassfish cluster. Ensure that Ehcache clustering has its own unique multicast address different to Glassfish.

Chapter 33

Google App Engine HowTo

33.1 Why?

- Speed - Ehcache cache operations take a few $\mu s, versusaround 60ms for Google's provided client-server cache, memcache g.$

- Cost - Because it uses way less resources, it is also cheaper.
- Object Storage - Ehcache in-process cache works with Objects that are not Serializable.

33.2 Compatibility

Ehcache is compatible and works with Google App Engine.

Google App Engine provides a constrained runtime which restricts networking, threading and file system access.

33.3 Limitations

All features of Ehcache can be used except for the DiskStore and replication. Having said that, there are workarounds for these limitations. See the Recipes section below.

As of June 2009, Google App Engine appears to be limited to a heap size of 100MB. (See http://gregluck.com/blog/archives/2009
for the evidence of this).

33.4 Versions

Version 1.6 of Ehcache is compatible with Google App Engine. 1.6.0-rc1 is not. Use a snapshot or the released version (which will be available soon).

Older versions will not work.

33.5 Configuring ehcache.xml

Make sure the following elements are commented out:

- *diskStore path="java.io.tmpdir"/*
- *cacheManagerPeerProviderFactory class= ../*
- *cacheManagerPeerListenerFactory class= ../*

  Within each cache element, ensure that:
- overFlowToDisk=false or overFlowToDisk is omitted
- diskPersistent=false or diskPersistent is omitted
- no replicators are added
- there is no bootstrapCacheLoaderFactory

  Copy and past this one to get started.

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

<Ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="ehcache.xsd" >

    <cacheManagerEventListenerFactory class="" properties=""/>

    <defaultCache
            maxElementsInMemory="10000"
            eternal="false"
            timeToIdleSeconds="120"
            timeToLiveSeconds="120"
            overflowToDisk="false"
            diskPersistent="false"
            memoryStoreEvictionPolicy="LRU"
            />

    <!--Example sample cache-->
    <cache name="sampleCache1"
           maxElementsInMemory="10000"
           maxElementsOnDisk="1000"
           eternal="false"
           timeToIdleSeconds="300"
           timeToLiveSeconds="600"
           memoryStoreEvictionPolicy="LFU"
            />
</ehcache>
```

33.6 Recipes

33.6.1 Setting up Ehcache as a local cache in front of memcacheg

The idea here is that your caches are set up in a cache hierarchy. Ehcache sits in front and memcacheg behind. Combining the two lets you elegantly work around limitations imposed by Googe App Engine. You get the benefits of the $\mu s speed of Ehcache together with the umlimited size of memcached.$

Ehcache contains the hooks to easily do this.

To update memcached, use a `CacheEventListener`.

To search against memcacheg on a local cache miss, use `cache.getWithLoader()` together with a `CacheLoader` for memcacheg.

33.6.2 Using memcacheg in place of a `DiskStore`

In the `CacheEventListener`, ensure that when `notifyElementEvicted()` is called, which it will be when a put exceeds the MemoryStore's capacity, that the key and value are put into memcacheg.

33.6.3 Distributed Caching

Configure all notifications in `CacheEventListener` to proxy throught to memcacheg.

Any work done by one node can then be shared by all others, with the benefit of local caching of frequently used data.

33.6.4 Dynamic Web Content Caching

Google App Engine provides acceleration for files declared static in appengine-web.xml.

e.g.

```
<static-files>
    <include path="/**" />
    <exclude path="/data/**" />
</static-files>
```

You can get acceleration for dynamic files using Ehcache's caching filters as you usually would.

See the Web Caching chapter.

33.7 Google App Engine FAQ

33.7.1 I get an error `java.lang.NoClassDefFoundError: java.rmi.server.UID is a restricted class`

You are using a version of Ehcache prior to 1.6.

Chapter 34

Tomcat Issues and Best Practices

Ehcache is probably used most commonly with Tomcat. This chapter documents some known issues with Tomcat and recommended practices.

Ehcache's own caching and gzip filter integration tests run against Tomcat 5.5 and Tomcat 6. Tomcat will continue to be tested against ehcache. Accordingly Tomcat is tier one for ehcache.

34.1 Tomcat Known Issues

Because Tomcat is so widely used, over time a list of known issues has been compiled. These issues and their solutions are listed below.

34.1.1 Problem rejoining a cluster after a reload

If I restart/reload a web application in Tomcat that has a CacheManager that is part of a cluster, the CacheManager is unable to rejoin the cluster. If I set logging for net.sf.ehcache.distribution to FINE I see the following exception: "FINE: Unable to lookup remote cache peer for .... Removing from peer list. Cause was: error unmarshalling return; nested exception is: java.io.EOFException.

The Tomcat and RMI class loaders do not get along that well. Move ehcache.jar to $TOMCAT_HOME/common/lib. This fixes the problem. This issue happens with anything that uses RMI, not just ehcache.

34.1.2 In development, there appear to be class loader memory leak as I continually redeploy my web application.

There are lots of causes of memory leaks on redeploy. Moving Ehcache out of the WAR and into $TOMCAT/common/lib fixes this leak.

34.1.3 net.sf.ehcache.CacheException: Problem starting listener for RMICachePeer ...

I get net.sf.ehcache.CacheException: Problem starting listener for RMICachePeer ... java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is: java.net.MalformedURLException: no protocol: Files/Apache. What is going on?

This issue occurs to any RMI listener started on Tomcat whenever Tomcat has spaces in its installation path.

It is is a JDK bug which can be worked around in Tomcat.

See http://archives.java.sun.com/cgi-bin/wa?A2=ind0205&L=rmi-users&P=797 and http://www.ontotext.com/kim/doc/sys-doc/faq-howto-bugs/known-bugs.html.

The workaround is to remove the spaces in your tomcat installation path.

34.1.4 Multiple Host Entries in Tomcat's server.xml stops replication from occurring

The presence of multiple *Host* entries in Tomcat's server.xml prevents replication from occuring. The issue is with adding multiple hosts on a single Tomcat connector. If one of the hosts is localhost and another starts with v, then the caching between machines when hitting localhost stops working correctly.

The workaround is to use a single *Host* entry or to make sure they don't start with "v".

Why this issue occurs is presently unknown, but is Tomcat specific.

Chapter 35

Building from Source

These instructions work for each of the modules, except for JMS Replication, which requires installation of a message queue. See that module for details.

35.1 Building an Ehcache distribution from source

To build Ehcache from source:

1. Check the source out from the subversion repository.
2. Ensure you have a valid JDK and Maven 2 installation.
3. From within the ehcache/core directory, type `mvn -Dmaven.test.skip=true install`

35.2 Running Tests for Ehcache

To run the test suite for Ehcache:

1. Check the source out from the subversion repository.
2. Ensure you have a valid JDK and Maven 2 installation.
3. From within the ehcache/core directory, type `mvn test`
4. If some performance tests fail, add a `-D net.sf.ehcache.speedAdjustmentFactor=x` System property to your command line, where x is how many times your machine is slower than the reference machine. Try setting it to 5 for a start.

35.3 Deploying Maven Artifacts

Ehcache has a repository and snapshot repository at oss.sonatype.org.

The repository is synced with the Maven Central Repository.

To deploy:

```
mvn deploy
```

This will fail because SourceForge has disabled ssh exec. You need to create missing directories manually using sftp access `sftp gregluck,ehcache@web.sourceforge.net`

35.4 Building the Site

(These instructions are for project maintainers)

You need the following unix utilities installed:

- maven 2.2.1
- a latex distribution (e.g. Tex Live 2008)
- ghostscript
- pdftk
- aptconvert
- netpbm
- xfig

  You also need a yDoc license.

  With all that, build the site as below:

  ```
  mvn -Dmaven.test.skip=true package site
  ```

  The site needs to be deployed from the target/site directory using:

  rsync -v -r * ehcache-stage.terracotta.lan:/export1/ehcache.org

  sudo -u maven -H /usr/local/bin/syncEHcache.sh

35.5 Deploying a release

35.5.1 Maven Release

```
mvn deploy
```

35.5.2 Sourceforge Release

```
mvn assembly:assembly
```

then manually upload to SourceForge

`sftp gregluck@frs.sourceforge.net`

and complete the file release process

Chapter 36

Frequently Asked Questions

36.1 There are a lot of product choices? Which one should I use?

Terracotta Ehcache is available in DX, EX and FX versions.

There is an Ehcache product brochure which explains the choices. That is a good start.

36.2 What are the software licenses used.

Terracotta Open Source is available with TPL.

The Ehcache open source modules are released under an Apache license.

There are some modules which are part of the commercial product offering e.g. monitoring server. These are available under a Terracotta Commercial License.

36.3 Does Ehcache run on JDK1.3/JDK1.4?

Older versions run on 1.3. Ehcache 1.5 runs on 1.4. Ehcache 1.6 required JDK 1.5.

36.4 Can you use more than one instance of Ehcache in a single VM?

As of ehcache-1.2, yes. Create your CacheManager using new CacheManager(...) and keep hold of the reference. The singleton approach accessible with the getInstance(...) method is still available too. Remember that Ehcache can supports hundreds of caches within one CacheManager. You would use separate CacheManagers where you want quite different configurations.

The Hibernate EhCacheProvider has also been updated to support this behaviour.

36.5 Can you use Ehcache with Hibernate and outside of Hibernate at the same time?

Yes. You use 1 instance of Ehcache and 1 ehcache.xml. You configure your caches with Hibernate names for use by Hibernate. You can have other caches which you interact with directly outside of Hibernate.

That is how I use Ehcache in the original project it was developed in. For Hibernate we have about 80 Domain Object caches, 10 StandardQueryCaches, 15 Domain Object Collection caches.

We have around 5 general caches we interact with directly using BlockingCacheManager. We have 15 general caches we interact with directly using SelfPopulatingCacheManager. You can use one of those or you can just use CacheManager directly.

I have updated the documentation extensively over the last few days. Check it out and let me know if you have any questions. See the tests for example code on using the caches directly. Look at CacheManagerTest, CacheTest and SelfPopulatingCacheTest.

36.6 What happens when maxElementsInMemory is reached? Are the oldest items are expired when new ones come in?

When the maximum number of elements in memory is reached, the least recently used ("LRU") element is removed. Used in this case means inserted with a put or accessed with a get.

If the overflowToDisk cache attribute is false, the LRU Element is discarded. If true, it is transferred asynchronously to the DiskStore.

36.7 Is it thread safe to modify Element values after retrieval from a Cache?

Remember that a value in a cache element is globally accessible from multiple threads. It is inherently not thread safe to modify the value. It is safer to retrieve a value, delete the cache element and then reinsert the value.

The UpdatingCacheEntryFactory does work by modifying the contents of values in place in the cache. This is outside of the core of Ehcache and is targeted at high performance CacheEntryFactories for SelfPopulatingCaches.

36.8 Can non-Serializable objects be stored in a cache?

As of ehcache-1.2, they can be stored in caches with MemoryStores.

Elements attempted to be replicated or overflowed to disk will be removed and a warning logged if not Serializable.

36.9 Why is there an expiry thread for the DiskStore but not for the MemoryStore?

Because the memory store has a fixed maximum number of elements, it will have a maximum memory use equal to the number of elements * the average size. When an element is added beyond the maximum size, the LRU element gets pushed into the DiskStore.

While we could have an expiry thread to expire elements periodically, it is far more efficient to only check when we need to. The tradeoff is higher average memory use.

The expiry thread keeps the disk store clean. There is hopefully less contention for the DiskStore's locks because commonly used values are in the MemoryStore. We mount our DiskStore on Linux using RAMFS

so it is using OS memory. While we have more of this than the 2GB 32 bit process size limit it is still an expensive resource. The DiskStore thread keeps it under control.

If you are concerned about cpu utilisation and locking in the DiskStore, you can set the diskExpiryThreadIntervalSeconds to a high number - say 1 day. Or you can effectively turn it off by setting the diskExpiryThreadIntervalSeconds to a very large value.

36.10 What elements are mandatory in ehcache.xml?

The documentation has been updated with comprehensive coverage of the schema for Ehcache and all elements and attributes, including whether they are mandatory. See the Declarative Configuration chapter.

36.11 Can I use Ehcache as a memory cache only?

Yes. Just set the overflowToDisk attribute of cache to false.

36.12 Can I use Ehcache as a disk cache only?

Yes. Set the maxElementsInMemory attribute of cache to 0.

This is strongly not recommended however. The minimum recommended value is 1. Performance is as much as 10 times higher when to one rather than 0. If not set to at least 1 a warning will be issued at Cache creation time.

36.13 Where is the source code? The source code is distributed in the root directory of the download.

It is called ehcache-x.x.zip. It is also available from SourceForge online or through SVN.

36.14 How do you get statistics on an Element without affecting them?

Use the Cache.getQuiet() method. It returns an Element without updating statistics.

36.15 How do you get WebSphere to work with ehcache?

It has been reported that IBM Websphere 5.1 running on IBM JDK1.4 requires commons-collection.jar in its classpath even though Ehcache will not use it for JDK1.4 and JDK5. (This is for versions of Ehcache lower than 1.6)

36.16 Do you need to call CacheManager.getInstance().shutdown() when you finish with ehcache?

Yes, it is recommended. If the JVM keeps running after you stop using ehcache, you should call CacheManager.getInstance().shutdown() so that the threads are stopped and cache memory released back to the

JVM. Calling shutdown also insures that your persistent disk stores get written to disk in a consistent state and will be usable the next time they are used.

If the CacheManager does not get shutdown it should not be a problem. There is a shutdown hook which calls the shutdown on JVM exit. This is explained in the documentation here.

36.17 Can you use Ehcache after a CacheManager.shutdown()?

Yes. When you call CacheManager.shutdown() is sets the singleton in CacheManager to null. If you try an use a cache after this you will get a CacheException.

You need to call CacheManager.create(). It will create a brand new one good to go. Internally the CacheManager singleton gets set to the new one. So you can create and shutdown as many times as you like.

There is a test which expliciyly confirms this behaviour. See CacheManagerTest#testCreateShutdownCreate()

36.18 I have created a new cache and its status is STATUS_UNINITIALISED. How do I initialise it?

You need to add a newly created cache to a CacheManager before it gets intialised. Use code like the following:

```
CacheManager manager = CacheManager.create();
Cache myCache = new Cache("testDiskOnly", 0, true, false, 5, 2);
manager.addCache(myCache);
```

36.19 Is there a simple way to disable Ehcache when testing?

Yes. There is a System Property based method of disabling ehcache. If disabled no elements will be added to a cache. Set the property "net.sf.ehcache.disabled=true" to disable ehcache.

This can easily be done using `-Dnet.sf.ehcache.disabled=true`> in the command line.

36.20 How do I dynamically change Cache attributes at runtime?

You can’t but you can achieve the same result as follows:

Cache cache = new Cache("test2", 1, true, true, 0, 0, true, 120, ...); cacheManager.addCache(cache);

See the JavaDoc for the full parameters, also reproduced here:

Having created the new cache, get a list of keys using cache.getKeys, then get each one and put it in the new cache. None of this will use much memory because the new cache element have values that reference the same data as the original cache. Then use cacheManager.removeCache("oldcachename") to remove the original cache.

36.21 I get net.sf.ehcache.distribution.RemoteCacheException: Error doing put to remote peer. Message was: Error unmarshaling return header; nested exception is: java.net.SocketTimeoutException: Read timed out. What does this mean.

It typically means you need to increase your socketTimeoutMillis. This is the amount of time a sender should wait for the call to the remote peer to complete. How long it takes depends on the network and the size of the Elements being replicated.

The configuration that controls this is the socketTimeoutMillis setting in cacheManagerPeerListenerFactory. 120000 seems to work well for most scenarios.

```
    <cacheManagerPeerListenerFactory
            class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"
            properties="hostName=fully_qualified_hostname_or_ip,
                        port=40001,
                        socketTimeoutMillis=120000"/>
```

36.22 Should I use this directive when doing distributed caching? *cacheManagerEventListenerFactory class="" properties=""/*

No. It is unrelated. It is for listening to changes in your local CacheManager.

36.23 What is the minimum config to get distributed caching going?

The minimum configuration you need to get distributed caching going is:

```
    <cacheManagerPeerProviderFactory
            class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
            properties="peerDiscovery=automatic,
                        multicastGroupAddress=230.0.0.1,
                        multicastGroupPort=4446"/>


<cacheManagerPeerListenerFactory
class="net.sf.ehcache.distribution.RMICacheManagerPeerListenerFactory"/>
```

and then at least one cache declaration with

```
<cacheEventListenerFactory
    class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>>>>
```

in it. An example cache is:

```
    <cache name="sampleDistributedCache1"
           maxElementsInMemory="10"
           eternal="false"
           timeToIdleSeconds="100"
```

```
        timeToLiveSeconds="100"
        overflowToDisk="false">
    <cacheEventListenerFactory
        class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>
</cache>
```

Each server in the cluster can have the same config.

36.24 How can I see if distributed caching is working?

You should see the listener port open on each server.

You can use the distributed debug tool to see what is going on. (See http://ehcache.org/documentation/remotedebugger.html).

36.25 Why can't I run multiple applications using Ehcache on one machine?

Because of an RMI bug, in JDKs before JDK1.5 such as JDK1.4.2, Ehcache is limited to one CacheManager operating in distributed mode per virtual machine. (The bug limits the number of RMI registries to one per virtual machine). Because this is the expected deployment configuration, however, there should be no practical effect. The tell tail error is `java.rmi.server.ExportException: internal error: ObjID already in use`

On JDK1.5 and higher it is possible to have multiple CacheManagers per VM each participating in the same or different clusters. Indeed the replication tests do this with 5 CacheManagers on the same VM all run from JUnit.

36.26 How many threads does Ehcache use, and how much memory does that consume?

The amount of memory consumed per thread is determined by the Stack Size. This is set using -Xss. The amount varies by OS. It is 512KB for Linux. I tend to override the default and set it to 100kb.

The threads are created per cache as follows:

- DiskStore expiry thread - if DiskStore is used
- DiskStore spool thread - if DiskStore is used
- Replication thread - if asynchronous replication is configured.

  If you are not doing any of the above, no extra threads are created

36.27 I am using Tomcat 5, 5.5 or 6 and I am having a problem. What can I do?

Tomcat is such a common deployment option for applications using Ehcache that there is a chapter on known issues and recommended practices.

See the Using Ehcache with Tomcat chapter. (http://ehcache.org/documentation/tomcat.html)

36.28 I am using Java 6 and getting a java.lang.VerifyError on the Backport Concurrent classes. Why?

The backport-concurrent library is used in Ehcache to provide java.util.concurrency facilities for Java 4 - Java 6. Use either the Java 4 version which is compatible with Java 4-6 or use the version for your JDK.

36.29 How do I get a memory only cache to persist to disk between VM restarts?

While disk persistence between restarts is a feature of the DiskStore only, you can get the same behaviour for a memory only cache by setting up a cache with maxElementsInMemory set to Integer.MAX_VALUE, 2147483647, overflowToDisk set to true and diskPersistent set to true.

36.30 I get a javax.servlet.ServletException: Could not initialise servlet filter when using SimplePageCachingFilter. Why?

If you use this default implementation, the cache name is called "SimplePageCachingFilter". You need to define a cache with that name in ehcache.xml. If you override CachingFilter you are required to set your own cache name.

36.31 I see, in my application's log:

```
WARN  CacheManager ...  Creating a new instance of CacheManager using the diskStorePath
 "C:\temp\tempcache" which is already used by an existing CacheManager.
```

This means, that for some reason, your application is trying to create a second or more instance of Ehcache's CacheManager with the same configuration. Ehcache is automatically resolving the Disk path conflict, which works fine.

To eliminate the warning:

- Use a separate configuration per instance
- If you only want one instance use the singleton creation methods i.e `CacheManager.getInstance()`. In Hibernate there is a special provider for this called

  `net.sf.ehcache.hibernate.SingletonEhCacheProvider`.

  See the Hibernate page for details.

36.32 How do I add a CacheReplicator to a cache that already exists? The cache event listening works but it does not get plumbed into the peering mechanism.

The current API does not have a CacheManager event for cache configuration change. You can however make it work by calling the notifyCacheAdded event.

```
getCache().getCacheManager().getCacheManagerEventListenerRegistry()
    .notifyCacheAdded("cacheName");
```

36.33 I am using the RemoteDebugger to monitor cluster messages but all I see is "Cache size: 0"

If you see nothing happening, but cache operations should be going through, enable trace (LOG4J) or finest (JDK) level logging on *code*net.sf.ehcache.distribution*/code* in the logging configuration being used by the debugger. A large volume of log messages will appear. The normal problem is that the CacheManager has not joined the cluster. Look for the list of cache peers.

Finally, the debugger in ehcache-1.5 has been improved to provide far more information on the caches that are replicated and events which are occurring.

36.34 With distributed replication on Ubuntu or Debian, I see the following warning,

```
WARN [Replication Thread] RMIAsynchronousCacheReplicator.flushReplicationQueue(324)
| Unable to send message to remote peer.
Message was: Connection refused to host: 127.0.0.1; nested exception is:

java.net.ConnectException: Connection refused

java.rmi.ConnectException: Connection refused to host: 127.0.0.1; nested exception is:

java.net.ConnectException: Connection refused
```

This is caused by a 2008 change to the Ubuntu/Debian linux default network configuration.

Essentially, this java call: `InetAddress.getLocalHost();` always returns the loopback address, which is 127.0.0.1. Why? Because in these recent distros, a system call of $ hostname always returns an address mapped onto the loopback device. Which causes ehcache's RMI Peer creation logic to always assign the loopback address, which causes the error you are seeing.

All you need to do is crack open the network config and make sure that the hostname of the machine returns a valid network address accessible by other peers on the network.

36.35 I see log messages about SoftReferences. What are these about and how do I stop getting the messages?

Ehcache uses SoftReferences with asynchronous RMI based replication, so that replicating caches do not run out of memory if the network is interrupted. Elements scheduled for replication will be collected instead. If this is happening, you will see warning messages from the replicator. It is also possible that a SoftReference can be reclaimed during the sending in which case you will see a debug level message in the receiving CachePeer.

Some things you can do to fix them:

- Set -Xms equal to -Xms. SoftReferences are also reclaimed in preference to increasing the heap size, which is a problem when an application is warming up.
- Set the -Xmx to a high enough value so that SoftReferences do not get reclaimed.

  Having done the above, SoftReferences will then only be reclaimed if there is some interruption to replication and the message queue gets dangerously high.

36.36 My Hibernate Query caches entries are replicating but the other caches in the cluster are not using them.

This is a Hibernate 3 bug. See http://opensource.atlassian.com/projects/hibernate/browse/HHH-3392 for tracking. It is fixed in 3.3.0.CR2 which was released in July 2008.

36.37 Active MQ Temporary Destinatons

ActiveMQ seems to have a bug in at least ActiveMQ 5.1 where it does not cleanup temporary queues, even though they have been deleted. That bug appears to be long standing but was though to have been fixed.

See:

- http://www.nabble.com/Memory-Leak-Using-Temporary-Queues-td11218217.html#a11218217
- http://issues.apache.org/activemq/browse/AMQ-1255

  The JMSCacheLoader uses temporary reply queues when loading. The Active MQ issue is readily reproduced in Ehcache integration testing. Accordingly, use of the JMSCacheLoader with ActiveMQ is not recommended. Open MQ tests fine.

36.38 Is Ehcache compatible with Google App Engine?

Version 1.6 is compatible. See the Google App Engine Howto

36.39 Can my app server use JMS Replication?

Some App Servers do not permit the creation of message listeners. This issue has been reported on Websphere 5. Websphere 4 did allow it. Tomcat allows it. Glassfish Allows it. Jetty allows it.

Usually there is a way to turn off strict EJB compliance checks in your app server. See your vendor documentation.

36.40 Why does Ehcache 1.6 use more memory than 1.5?

ConcurrentHashMap does not provide an eviction mechanism. We add that ourselves. For caches larger than 5000 elements, we create an extra ArrayList equal to the size of the cache which holds keys. This can be an issue with larger keys. An optimisation which cache clients can use is:

```
http://www.codeinstructions.com/2008/09/instance-pools-with-weakhashmap.html

To reduce the number of key instances in memory to just one per logical
key, all puts to the underlying ConcurrentHashMap could be replaced by
map.put(pool.replace(key), value), as well as keyArray.set(index,
pool.replace(key))

You can take this approach when producing the keys before handing them over to EhCache.
```

Even with this approach there is still some added overhead consumed by a reference consumed by each ArrayList element.

Index