EHCACHE

a product from TERRACOTTA

Ehcache 2.5.x Documentation

Ehcache 2.5.x Documentation

Ehcache 2.5.x Documentation

Ehcache 2.5.x Documentation

Ehcache 2.5.x Documentation

Ehcache 2.5.x Documentation

Ehcache 2.5.x Documentation

Ehcache 2.5.x Documentation

Ehcache 2.5.x Documentation

Ehcache 2.5.x Documentation

Getting Started Overview

The following sections provide a documentation Table of Contents and additional information sources for getting started with Ehcache.

"Getting Started" Table of Contents

Topic	Description
Hello, Ehcache	Introduction to caching with Ehcache and the benefits of caching.
Cache Topologies	This overview of the Ehcache caching topologies highlights the differences between distributed and replicated caching.
Key Classes and Methods	Definitions of terms, key Ehcache classes, and cache usage patterns, as well as explanations of the CacheManager and the Ehcache interface.
About Distributed Cache	Distributed Ehcache combines an in-process Ehcache with the Terracotta Server Array acting as a backing cache store. This page details architecture and development for Distributed Ehcache running in a Terracotta cluster.
Consistency Options	Explanation of the Distributed Ehcache consistency models in terms of standard distributed systems theory, plus use cases and recommended practices.
Storage Options	Description of the three storage options for Ehcache--MemoryStore, OffHeapStore, and DiskStore--plus suitable Element types, configuration examples, and performance considerations.
Using Ehcache	A quick guide to get you started with the packed Ehcache kit and using Ehcache with several popular platforms.
Building From Source	Create your own Ehcache or distributed Ehcache build, rather than using the packed kit.

Getting Started in Theory and Practice

The following pages cover general caching theory:

- Hello, Ehcache
- Cache Topologies
- About Distributed Cache

The following pages provide background information that will help you to make informed decisions when configuring Ehcache:

- Cache Concepts
- Cache Consistency Options
- Storage Options

The following pages get you up and running:

- Using Ehcache
- About Distributed Cache Code Samples
- Building From Source

Hello, Ehcache

Introduction

Ehcache is a cache library introduced in October 2003 with the key goal of improving performance by reducing the load on underlying resources. Ehcache is not just for general-purpose caching, however, but also for caching Hibernate (second-level cache), data access objects, security credentials, web pages. It can also be used for SOAP and RESTful server caching, application persistence, and distributed caching.

Definitions

- **cache**: 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.
- **cache-hit**: When a data element is requested of the cache and the element exists for the given key, it is referrred to as a cache hit (or simply 'hit').
- **cache-miss**: When a data element is requested of the cache and the element does not exist for the given key, it is referred to as a cache miss (or simply 'miss').
- **system-of-record**: The core premise of caching assumes that there is a source of truth for the data. This is often referred to as a system-of-record (SOR). The cache acts as a local copy of data retrieved from or stored to the system-of-record. This is often a traditional database, although it may be a specialized file system or some other reliable long-term storage. For the purposes of using Ehcache, the SOR is assumed to be a database.
- **SOR**: See system-of-record.

Why caching works

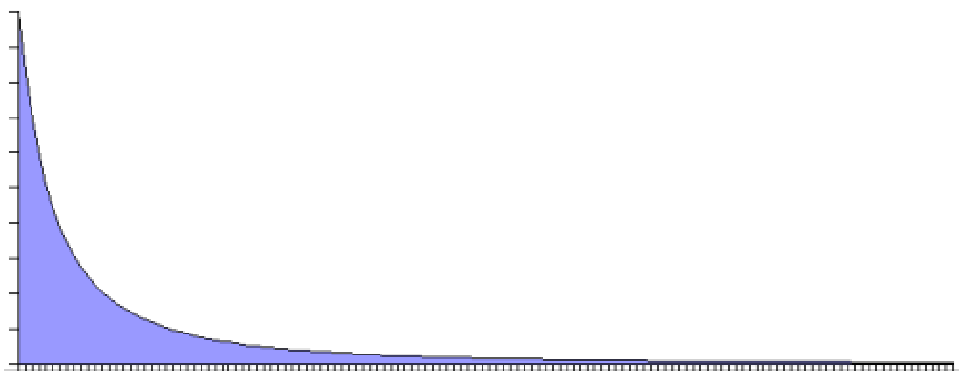
Locality of Reference

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.

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

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.

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.

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.

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.

Speeding up I/O-bound Applications

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.

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.

How much will an application speed up with Caching?

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.

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 + .0006)
= 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 + .0001)
    = 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 + .0004)
    = 1.6 times system speedup
```

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.

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 in Hits	Hit Ratio 1 Server	Hit Ratio 2 Servers	Hit Ratio 3 Servers	Hit Ratio 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.)

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)
```

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%. 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.

Cache Topologies

Introduction

Ehcache is used in the following topologies:

- Standalone – The cached data set is held in the application node. Any other application nodes are independent with no communication between them. If standalone caching is being used where there are multiple application nodes running the same application, then there is Weak Consistency between them. They contain consistent values for immutable data or after the time to live on an Element has completed and the Element needs to be reloaded.
- Distributed Ehcache – The data is held in a Terracotta Server Array with a subset of recently used data held in each application cache node. The distributed topology supports a very rich set of consistency modes.
    - ♦ More information on configuring consistency
    - ♦ More information on how consistency affects performance
    - ♦ More in-depth information on how consistency works
- Replicated â— The cached data set is held in each application node and data is copied or invalidated across the cluster without locking. Replication can be either asynchronous or synchronous, where the writing thread blocks while propagation occurs. The only consistency mode available in this topology is Weak Consistency.

Many production applications are deployed in clusters of multiple instances for availability and scalability. However, without a distributed or replicated cache, application clusters exhibit a number of undesirable behaviors, such as:

- **Cache Drift** -- if each application instance maintains its own cache, updates made to one cache will not appear in the other instances. This also happens to web session data. A distributed or replicated cache ensures that all of the cache instances are kept in sync with each other.
- **Database Bottlenecks** -- In a single-instance application, a cache effectively shields a database from the overhead of redundant queries. However, in a distributed application environment, each instance much load and keep its own cache fresh. The overhead of loading and refreshing multiple caches leads to database bottlenecks as more application instances are added. A distributed or replicated cache eliminates the per-instance overhead of loading and refreshing multiple caches from a database.

The following sections further explore distributed and replicated caching.

Distributed Caching (Distributed Ehcache)

Ehcache provides distributed caching using the Terracotta Server Array, enabling data sharing among multiple CacheManagers and their caches in multiple JVMs. By combining the power of the Terracotta Server Array with the ease of Ehcache application-data caching, you can:

- linearly scale your application to grow with requirements;
- rely on data that remains consistent across the cluster;
- offload databases to reduce the associated overhead;
- increase application performance with distributed in-memory data;
- access even more powerful APIs to leverage these capabilities.

Distributed Caching (Distributed Ehcache)

Using distributed caching is the recommended method of operating Ehcache in a clustered or scaled-out application environment. It provides the highest level of performance, availability, and scalability.

Adding distributed caching to Ehcache takes only two lines of configuration. To get started, see the tutorial for distributed caching with Terracotta.

Replicated Caching

In addition to the built-in distributed caching, Ehcache has a pluggable cache replication scheme which enables the addition of cache replication mechanisms. The following additional replicated caching mechanisms are available:

- RMI
- JGroups
- JMS
- Cache Server

Each of the is covered in its own chapter. One solution is to replicate data between the caches to keep them consistent, or coherent. Typical 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.

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.

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.

Potential Issues with Replicated Caching

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.

Use of Time To Idle

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

Key Classes and Methods

Introduction

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. The logical representations of these components are actualized mostly through the classes discussed below. The methods provided by these classes are largely responsible for providing programmatic access to working with Ehcache.

CacheManager

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

CacheManager Creation Modes

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

Versions of Ehcache before version 2.5 allowed any number of CacheManagers with the same name (same configuration resource) to exist in a JVM. Therefore, each time `new CacheManager(...)` was called, a new CacheManager was created without regard to existing CacheManagers. Calling `CacheManager.create(...)` returned the existing singleton CacheManager with the configured name (if it existed) or created the singleton based on the passed-in configuration.

Ehcache 2.5 and higher does not allow multiple CacheManagers with the same name to exist in the same JVM. `CacheManager()` constructors creating non-Singleton CacheManagers can violate this rule, causing a NullPointerException. If your code may create multiple CacheManagers of the same name in the same JVM, avoid this error by using the static `CacheManager.create()` methods, which always return the named (or default unnamed) CacheManager if it already exists in that JVM. If the named (or default unnamed) CacheManager does not exist, the `CacheManager.create()` methods create it.

NOTE: In Ehcache 2.5.0/2.5.1 `Cachemanager.create(...)` gets or creates the CacheManager regardless of whether it is a singleton or not. In Ehcache 2.5.2, calling `CacheManager.create(...)` returns the existing singleton CacheManager with the configured name (if it exists) or creates the singleton based on the passed-in configuration.

Ehcache 2.5.2 introduced the `CacheManager.newInstance(...)` method, which parses the passed-in configuration to either get the existing named CacheManager or create that CacheManager if it doesn't exist.

With Ehcache 2.5.2 and higher, the behavior of the CacheManager creation methods is as follows:

- `CacheManager.newInstance(Configuration configuration)` – Create a new CacheManager or return the existing one named in the configuration.
- `CacheManager.create()` – Create a new singleton CacheManager with default configuration, or return the existing singleton. This is the same as `CacheManager.getInstance()`.
- `CacheManager.create(Configuration configuration)` – Create a singleton CacheManager with the passed-in configuration, or return the existing singleton.
- `new CacheManager(Configuration configuration)` – Create a new CacheManager, or throw an exception if the CacheManager named in the configuration already exists.

See the Ehcache API documentation for more information on these methods, including options for passing in configuration. For examples, see Code Samples.

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.

Ehcache

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`.

Element

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.

About Distributed Cache

Introduction

Distributed Cache, formally called Terracotta Distributed Ehcache, is Ehcache running in a Terracotta cluster. Distributed caching is the recommended method of operating Ehcache in a clustered or scaled-out application environment, as it enables data sharing among multiple CacheManagers and their caches in multiple JVMs.

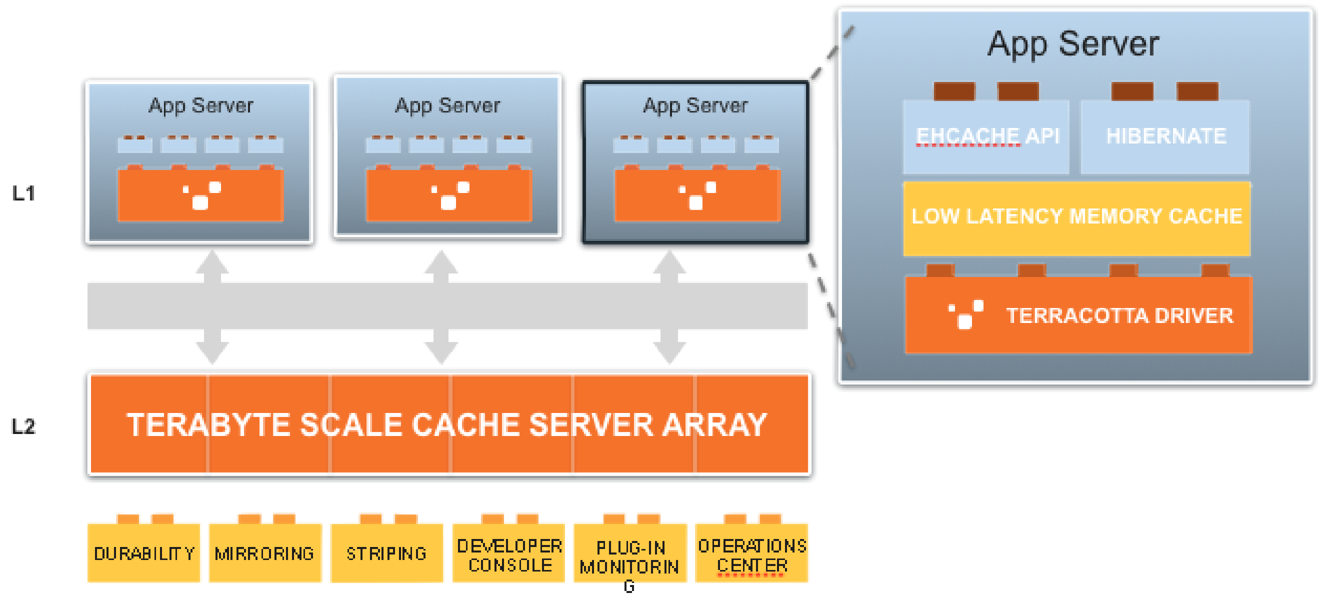
You can find tutorials, installation procedures, best practices, details on the Terracotta Server Array, and more in the Terracotta documentation.

Architecture

Distributed Ehcache combines an in-process Ehcache with the Terracotta Server Array acting as a backing cache store.

Logical View

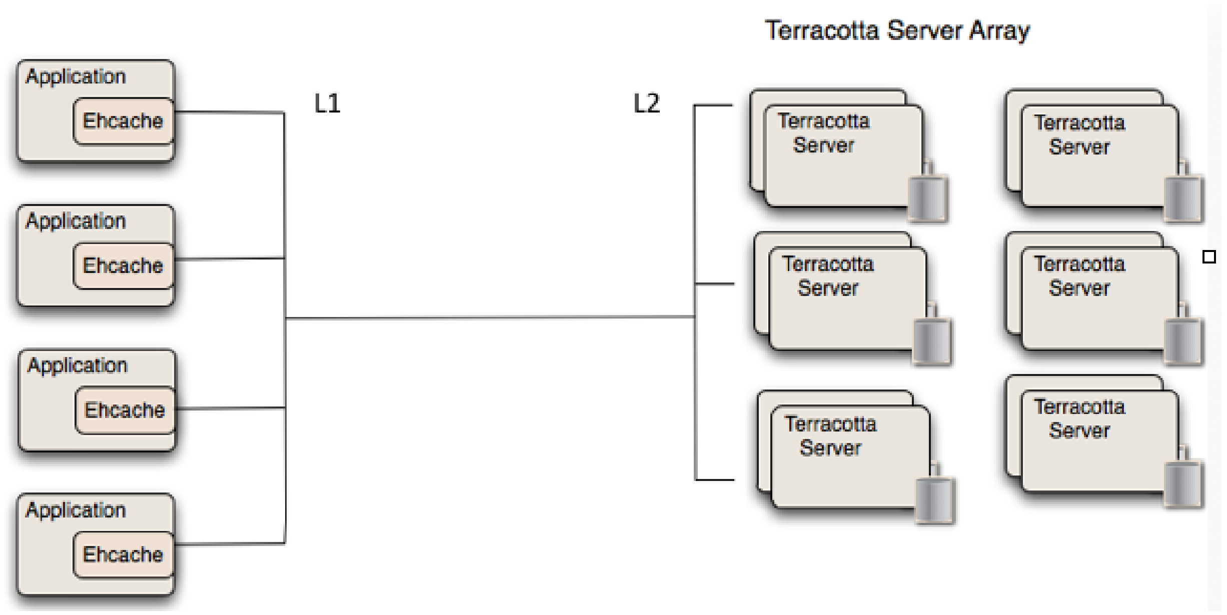
With Terracotta Server Array the data is split between an Ehcache node (the L1 cache) and the Terracotta Server Array itself (the L2 Cache). As with the other replication mechanisms, the L1 can hold as much data as is comfortable. But there is always a complete copy of all cache data in the L2. The L1 therefore acts as a hot-set of recently used data. Distributed Ehcache is persistent and highly available, leaving the cache unaffected by the termination of an Ehcache node. When the node comes back up it reconnects to the Terracotta Server Array L2 and as it uses data fills its local L1.



Network View

From a network topology point of view Distributed Ehcache consists of:

- L1 - the Ehcache library is present in each app. An Ehcache instance, running in-process sits in each JVM.
- L2 - Each Ehcache instance (or node) maintains a connection with one or more Terracotta servers. These are arranged in pairs for high availability. A pair is known as a *mirror group*. For high availability each server runs on a dedicated server. For scale out multiple pairs are added. Consistent hashing is used by the Ehcache nodes to store and retrieve cache data in the correct server pair. The terms Stripe or Partition are then used to refer to each mirror group.



Memory Hierarchy View

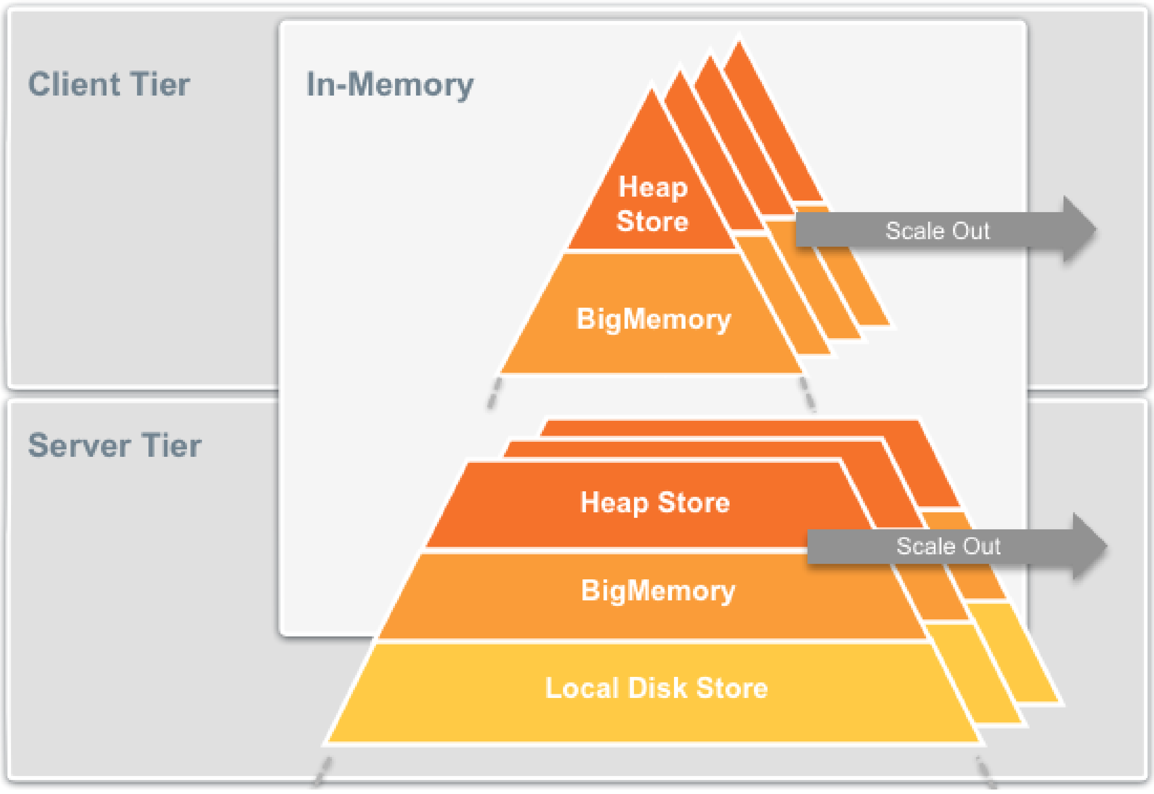
Another way to look at the architecture of Distributed Ehcache is as a tiered memory hierarchy. Each in-process Ehcache instance (L1s) can have:

- Heap memory
- Off-heap memory (BigMemory). This is stored in direct byte buffers.

The Terractta servers (L2s) run as Java processes with their own memory hierarchy:

- Heap memory
- Off-heap memory (BigMemory). This is stored in direct byte buffers.
- Disk storage. This is optional. It provides persistence in the event both servers in a mirror group suffer a crash or power outage at the same time.

Memory Hierarchy View



Differences Between Terracotta Distributed Cache and Standalone or Replicated Cache

Differences in behavior and available functionality between distributed cache and standalone and replicated caches are called out in the documentation. Some major differences are listed here:

- In distributed caches locking takes effect on individual keys, while in standalone caches locking takes effect on segments that include a number of keys.
- In distributed caches, all cache stores are shared.
- Only distributed caches can be made transactional caches (`<cache ... transactionalMode="xa">`).
- Standalone caches load very quickly and do not require a bulk-loading API.
- Distributed caches are "cluster safe" for Hibernate (locks are used for writing to distributed caches). There is no need for `session.refresh()` as with replicated caches.
- Extreme scaling using multiple server stripes is available for distributed cache.
- Replication requires use of CacheEventListeners.
- Distributed caching can be used to create a clustered message queue for updating a database in a way that keeps data consistent.
- Distributed caches come with Terracotta High Availability and durability, greatly benefitting use cases requiring features such as write-through (CacheWriter) queues (`<cacheWriter writeMode="write-behind">`).
- When using read-through with write-behind, distributed caches can add cluster-wide consistency to cache data.

Code Samples

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"/>
```

This tells Ehcache to load the Terracotta server config from localhost port 9510. For `url` configuration options, refer to "Adding an URL Attribute" in Terracotta Clustering Configuration Elements. 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"
     maxEntriesLocalHeap="1000"
     eternal="false"
     timeToIdleSeconds="3600"
     timeToLiveSeconds="1800"
     overflowToDisk="false">
    <terracotta/>
  &lt/cache>
```

That's it!

Development with Maven and Ant

With a Distributed Ehcache, there is a Terracotta Server Array. At development time, this necessitates running a server locally for integration and/or interactive testing. There are plugins for Maven and Ant to simplify and automate this process.

For Maven, Terracotta has a plugin available which makes this very simple.

Setting up for Integration Testing

```
<pluginRepositories>
   <pluginRepository>
       <id>terracotta-snapshots</id>
       <url>http://www.terracotta.org/download/reflector/maven2</url>
       <releases>
           <enabled>true</enabled>
       </releases>
       <snapshots>
           <enabled>true</enabled>
       </snapshots>
   </pluginRepository>
</pluginRepositories>
<plugin>
   <groupId>org.terracotta.maven.plugins</groupId>
   <artifactId>tc-maven-plugin</artifactId>
   <version>1.5.1</version>
   <executions>
       <execution>
           <id>run-integration</id>
           <phase>pre-integration-test</phase>
           <goals>
               <goal>run-integration</goal>
           </goals>
       </execution>
       <execution>
           <id>terminate-integration</id>
           <phase>post-integration-test</phase>
           <goals>
               <goal>terminate-integration</goal>
           </goals>
       </execution>
   </executions>
</plugin>
```

Interactive Testing

To start Terracotta:

```
mvn tc:start
```

To stop Terracotta:

```
mvn tc:stop
```

See the Terracotta Forge for a complete reference.

Cache Consistency Options

Introduction

This page explains the Distributed Ehcache consistency models in terms of standard distributed systems theory.

For a practical discussion of how performance and consistency interact in Distributed Ehcache, see this section.

Server-Side Consistency

Leaving aside the issue of data also held in the Ehcache nodes, let us look at the server side consistency of the Terracotta Server Array (TSA).

Server Deployment Topology

Large datasets are handled with partitions which are managed automatically using a consistent hashing algorithm once a set of "stripes" are defined in the <tc-config>. There is no dynamic resizing of clusters, so the consistent hash always resolves to the same stripe. The TSA is typically deployed with a pair of servers per partition of data, which is known in the <tc-config> as a Mirror Group. A mirror group has an active server which handles all requests for that partition and a mirror server, or hot standby, which does not service any requests. The active server propagates changes to the mirror server. In the language of consistency protocols, the active and mirror are replicas - they should contain the same data.

Restating in terms of Quorum based replicated-write protocols

To use the terminology from Gifford (1979), a storage system has *N* storage replicas. A write is a *W*. A read is an *R*. The server-side storage system will be strongly consistent if:

- R + W > N

  and
- W > N/2

In Terracotta, there is one active and one mirror. The acknowledgement is not sent until all have been written to. We always read from only one replica, the Active. So R = 1, W = 2, N = 2. Substituting the terms of R + W > N, we get 1 + 2 > 2, which is clearly true. And for W > N/2 we get 2 > 2/2 => 2 > 1 which is clearly true. Therefore we are strongly consistent server side.

Client-Side Consistency

Because data is also held in Ehcache nodes, and Ehcache nodes are what application code interacts with, there is more to the story than consistency in the TSA. Werner Vogel's seminal "Eventually Consistent" paper presented standard terms for client-side consistency and a way of reasoning about whether that consistency can be achieved in a distributed system. This paper in turn referenced Tannenbaum's Distributed Systems: Principles and Paradigms (2nd Edition).

Tannenbaum was popularising research work done on Bayou, a database system. See page 290 of "Distributed Systems, Principles and Paradigms" by Tannenbaum and Van Steen for detailed coverage of this material.

Model Components

Before explaining our consistency modes, we need to expain the standard components of the the reference model, which is an abstract model of a distributed system that can be used for studying interactions.

- A storage system. The storage system consists of data stored durably in one server or multiple servers connected via a network. In Ehcache, durability is optional and the storage system might simply be in memory.
- Client Process A. This is a process that writes to and reads from the storage system.
- Client Processes B and C. These two processes are independent of process A and write to and read from the storage system. It is irrelevant whether these are really processes or threads within the same process; what is important is that they are independent and need to communicate to share information. Client-side consistency has to do with how and when observers (in this case the processes A, B, or C) see updates made to a data object in the storage systems.

Mapping the Model to Distributed Ehcache

The model maps to Distributed Ehcache as follows:

- there is a Terracotta Server Array which is the 'storage system';
- there are three nodes connected to the Terracotta Server Array: Ehcache A, B and C, mapping to the processes in the standard model;
- a "write" in the standard model is a "put" or "remove" in Ehcache.

Standard Client-Side Consistency Modes

It then goes on to define the following consistencies where process A has made an update to a data object:

- Strong consistency. After the update completes, any subsequent access (by A, B, or C) will return the updated value.
- Weak consistency. The system does not guarantee that subsequent accesses will return the updated value.
- Eventual consistency. This is a specific form of weak consistency; the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value. If no failures occur, the maximum size of the inconsistency window can be determined based on factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme.

Within eventual consistency there are a number of desirable properties:

- Read-your-writes consistency. This is an important model where process A, after it has updated a data item, always accesses the updated value and will never see an older value. This is a special case of the causal consistency model.
- Session consistency. This is a practical version of the previous model, where a process accesses the storage system in the context of a session. As long as the session exists, the system guarantees read-your-writes consistency. If the session terminates because of a certain failure scenario, a new session needs to be created and the guarantees do not overlap the sessions.

- Monotonic read consistency. If a process has seen a particular value for the object, any subsequent accesses will never return any previous values.
- Monotonic write consistency. In this case the system guarantees to serialize the writes by the same process. Systems that do not guarantee this level of consistency are notoriously hard to program.

Finally, in eventual consistency, the period between the update and the moment when it is guaranteed that any observer will always see the updated value is dubbed the inconsistency window.

Consistency Modes in Distributed Ehcache

The consistency modes in Terracotta distributed Ehcache are "strong" and "eventual". Eventual consistency is the default mode.

Strong Consistency

In the distributed cache, strong consistency is configured as follows:

```
<cache name="sampleCache1"
...
 />
   <terracotta consistency="strong" />
</cache>
```

We will walk through how a write is done and show that it is strongly consistent.

1. A thread in Ehcache A performs a write.
2. Before the write is done, a write lock is obtained from the Terracotta Server (storage system). The write lock is granted only after all read locks have been surrendered.
3. The write is done to an in-process Transaction Buffer. Within the Java process the write is thread-safe. Any local threads in Ehcache A will have immediate visibility of the change.
4. Once the change has hit the Transaction Buffer which is a LinkedBlockingQueue, a notify occurs, and the Transaction Buffer initiates sending the write (update) asynchronously to the TSA (storage system).
5. The Terracotta Server is generally configured with multiple replicas forming a Mirror Group. Within the mirror group there is an active server, and one or more mirror servers. The write is to the active server. The active server does not acknowledge the write until it has written it to each of the mirror servers in the mirror group. It then sends back an acknowledgement to Ehcache A which then deletes the write from the Transaction Buffer.
6. A read or write request from Ehcache A is immediately available because a read lock is automatically granted when a write lock has already been acquired. A read or write request in Ehcache B or C requires the acquisition of a read or write lock, respectively, which will block until step 5 has occurred. In addition, if you have a stale copy locally, it is updated first. When the lock is granted, the write is present in all replicas. Because Ehcache also maintains copies of Elements in-process in potentially each node, if any of Ehcache A, B or C have a copy they are also updated before Step 5 completes.

Note: This analysis assumes that if the `nonstop` is being used, it is configured with the default of Exception, so that on a `clusterOffline` event no cache operations happen locally. Nonstop allows fine-grained tradeoffs to be made in the event of a network partition, including dropping consistency.

Eventual Consistency

Distributed Ehcache can have eventual consistency in the following ways:

- Configured with `consistency="eventual"`.
- Set programmatically with a bulk-loading mode, using `setNodeBulkLoadEnabled(boolean)`.
- Configured with <UnlockedReadsView>, a `CacheDecorator` that can be created like a view on a cache to show the latest writes visible to the local Ehcache node without respect for any locks.
- Using bulk-loading Cache methods putAll(), getAll(), and removeAll(). These can also be used with strong consistency. If you can use them, it's unnecessary to use bulk-load mode. See the API documentation for details.

Regardless, Ehcache B and C will eventually see the change made by Ehcache A, generally with consistency window of 5ms (with no partitions or interruptions). If a GC happens on a TSA node, or Ehcache A or B, the inconsistency window is increased by the length of the GC.

If `setNodeBulkLoadEnabled(true)` is used, it causes the TSA to not update Ehcache B and C. Instead, they are set to a 5 minute fixed TTL. The inconsistency window thus increases to 5 minutes plus the above.

If a network partition occurs that is long enough to cause an Ehcache A to be ejected from the cluster, the only configurable option is to discard on rejoin. Once this happens Ehcache A or B gets the write. From the perspective of other threads in Ehcache A, all writes are thread-safe.

Java Memory Model Honored

In all modes the *happens-before* requirement of the Java Memory Model is honored. As a result the following is true:

- A thread in Ehcache A will see any writes made by another thread. => Read your writes consistency.
- Monotonic Read Consistency in Ehcache A is true.
- Monotonic Write Consistency is Ehcache A is true.

Consistency in Web Sessions

It should be noted that desirable characteristics of eventual consistency are from the point of view of Ehcache A. From the context of a web application, in order for an end user interacting with a whole application to see this behaviour use sticky sessions.

This way the user interacts with the same node (Ehcache A) for each step. If an application node falls over, a new session will be established. The time between the last write, failure, detection by the load balancer and allocation to a new application node will take longer than the 5ms+ that it takes for all Ehcache nodes in the cluster to get the write. So when the new application node is switched to, eventual consistency has occurred and no loss of consistency is observed by the user.

If you want to avoid sticky sessions, try relying on the time gap between a click or submit and the next one in a "click path" that takes much longer than the 5ms+ that it takes for other nodes to become eventually consistent.

In an Internet context the user is sufficiently distant from the server so that the response time is at least an order of magnitude greater than the inconsistency window. Probabilistically it is therefore unlikely that a user would see inconsistency.

Other Safety Features

Ehcache offers a rich set of data safety features. In this section we look at some of the others and how they interact with the `strong` and `eventual` consistency.

CAS Cache Operations

We support three Compare and Swap (CAS) operations:

- `cache.replace(Element old, Element new)`
- `cache.putIfAbsent(Element)`
- `cache.remove(Element)`

In each case the TSA will only perform the write if the old value is the same as that presented. This is guaranteed to be done atomically as required by the CAS pattern. CAS achieves strong consistency between A, B and C. The key difference is that it achieves it with optimistic locking rather than pessimistic locking. As with all optimistic locking approaches, the operations are not guaranteed to succeed. If someone else got in and changed the Element ahead of you, the methods will return `false`. You should read the new value, take that into account in your business logic and then retry your mutation.

CAS will work with both `strong` and `eventual` consistency modes, but because it does not use the locks it does not need `strong`. However, with eventual consistency two simultaneous `replace()` operations in different nodes (or threads) can both return true. But at the TSA, only one of the operations is allowed to succeed and all competing values are invalidated, eventually making the caches consistent in all nodes.

Use Cases And Recommended Practices

In this section we look at some common use cases and give advice on what consistency and safety options should be used. These serve as a useful starting point for your own analysis. We welcome commentary and further discussion on these use cases. Please post to the Ehcache mailing list or post your questions on the forums.

Shopping Cart - optimistic inventory

Problem

A user adds items to a shopping cart. Do not decrement inventory until checkout.

Solution

Use eventual consistency.

Shopping Cart with Inventory Decrementing

Problem

A user adds items to a shopping cart. There is limited inventory and the business policy is that the first user to add the inventory to their shopping cart can buy it. If the user does not proceed to checkout, a timer will release the inventory back. As a result, inventory must be decremented at the time the item is added to the shopping cart.

Solution

Use strong consistency with one of:

- explicit locking
- local transactions
- XA transactions

The key thing here is that two resources have to be updated: the shopping cart, which is only visible to one user, and on it's own has low consistency requirements, and an inventory which is transactiional in nature.

Financial Order Processing - write to cache and database

Problem

An order processing system sends a series of messages in a workflow, perhaps using Business Process Management software. The system involves multiple servers and the next step in the processing of an order may occur on any server. Let's say there are 5 steps in the process. To avoid continual re-reading from a database, the processing results are also written to a distributed cache. The next step could execute in a few ms to minutes depending on what other orders are going through and how busy the hardware is.

Solution

Use strong consistency plus XA transactions. Because the execution step cannot be replayed once completed, and may be under the control of a BPM, it is very important that the change in state gets to the cache cluster. Synchronous writes can also be used (at a high performance cost) so that the put to the cache does not return until the data has been applied. If an executing node failed before the data was transferred, the locks would still be in place preventing readers from reading stale data, but that will not help the next step in the process. XA transactions are needed because we want to keep the database and the cache in sync.

Immutable Data

Problem

The application uses data that once it comes into existence is immutable. Nothing is immutable forever. The key point is that it is immutable up until the time of the next software release. Some examples are:

- application constants
- reference data - zip and post codes, countries etc.

If you analyse database traffic commonly used reference data turns out to be a big hitter. As they are immutable they can only be appended or read, never updated.

Solution

In concurrent programming, immutable data never needs further concurrency protection. So we simply want to use the fastest mode. Here we would always use eventual consistency.

Financial Order Processing - write to cache as SOR

Problem

An order processing system sends a series of messages in a workflow, perhaps using Business Process Management software. The system involves multiple servers and the next step in the processing of an order may occur on any server. Let's say there are 50 steps in the process. To avoid overloading a database the processing results at each step only written to a distributed cache. The next step could execute in a few ms to minutes depending on what other orders are going through and how busy the hardware is.

Solution

Use one of:

- strong consistency and local transactions (if changes are needed to be applied to multiple caches or entries). Because the execution step, once completed cannot be replayed, and may be under the control of a BPM, it is very important that the change in state gets to the cache cluster. Synchronous writes can also be used (at a high performance cost) so that the put to the cache does not return until the data has been applied. If an executing node failed before the data was transferred, the locks would still be in place preventing readers from reading stale data, but that will not help the next step in the process.
- CAS operations with eventual consistency. The CAS methods will not return until the data has been applied to the server, so it is not necessary to use synchronous writes. In a 50 step process it is likely there are key milestones. Often it is desirable to record these in a database with the non-milestone steps recorded in the cache. For these key milestones use the "Financial Order Processing - write to cache and database" pattern.

E-commerce web app with Non-sticky sessions

Here a user makes reads and writes to a web application cluster. There are n servers where n > 1. The load balancer is non-sticky, so any of the n servers can be hit on the next HTTP operation. When a user submits using a HTML form, either a GET or POST is done based on the form action. And if it is an AJAx app, then requests are being done with `XMLHttpRequest` and any HTTP request method can be sent. If POST (form and AJAX) or PUT (AJAX) is used, no content is returned and a separate GET is required to refresh the view or AJAX app.

The key point is that sending a change and getting a view may happen with one request or two. If it happens with two, then the same server might respond to the second request or not. The probability that the second server will be the same as the first is 1/n. AJAX apps can further exacebate this situation. A page may make multiple requests to fill different panels. This opens up the possibility of, within a single page, having data come from multiple servers. Any lack of consistency could be glaring indeed.

Solution

Use one of:

- strong consistency
- CAS

Other options can be added depending on what is needed for the request. e.g. XA if a database plus the cache is updated.

E-commerce web app with sticky sessions

Problem

Here a user makes reads and writes to a web application cluster. The load balancer is sticky, so the same server should be hit on the next HTTP operation. There are different ways of configuring sticky sessions. The same server might be used for the length of a session, which is the standard meaning, or a browser's IP can permanently hash to a server. In any case, each request is guaranteed to hit the same server.

Solution

The same server is always hit. The consistency mode depends on whether only the user making the changes needs to see them applied (read your writes, monotonic reads, monotonic writes), or whether they are mutating shared-state, like inventory where write-write conflicts might occur. For mutating user-only consistency, use eventual consistency. For multi-user shared state, use strong consistency at a minimum plus further safety mechanisms depending on the type of mutation.

E-commerce Catalog

Problem

Catalogues display inventory. There are product details and pricing. There may be also be an inventory status of available or sold out. Catalogue changes are usually made by one user or process (for example a daily update load from a supplier) and usually do not have write-write conflicts. While the catalogue is often non-sticky, admin users are typically configured sticky. There is often tolerance for the displayed catalogue to lag behind the change made. Users following a click path are usually less tolerant about seeing inconsistencies.

Solution

The person making the changes can see a consistent view by virtue of the sticky session. So eventual consistency will often be enough. For end users following a click path, they need a consistent view. However, the network or Internet time, plus their think time to move along the path, adds up to seconds and minutes, while eventual consistency will propagate in the order of 2+ milliseconds. With eventual consistency, it is very unlikely they will see inconsistency. The general recommendation is therefore to use eventual consistency.

Storage Options

Introduction

Ehcache has three stores:

- a MemoryStore
- an OffHeapStore (BigMemory, Enterprise Ehcache only) and
- a DiskStore (two versions: open source and Ehcache Enterprise)

This page addresses relevant storage issues and provides the suitable element types for each storage option.

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:

- **Safe**
    - Thread safe for use by multiple concurrent threads.
    - Tested for memory leaks. The MemoryCacheTest 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 higher 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.

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.
- **Least 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 convenient method which can provide an estimate of the size in bytes of the `MemoryStore`, calculateInMemorySize(). It returns the serialized size of the cache. However, do not use this method in production, as it is very slow. It is only meant to provide a rough estimate.

An alternative would be to have an expiry thread. This is a trade-off between lower memory use and short locking periods and CPU utilization. The design is in favour of the latter. For those concerned with memory use, simply reduce the cache's size in memory (see How to Size Caches for more information).

BigMemory (Off-Heap Store)

BigMemory is a pure Java product from Terracotta that permits caches to use an additional type of memory store outside the object heap. It is packaged for use in Enterprise Ehcache as a snap-in job store called the "off-heap store."

This off-heap store, which is not subject to Java GC, is 100 times faster than the DiskStore and allows very large caches to be created (we have tested this up to 350GB). Because off-heap data is stored in bytes, there are two implications:

- Only Serializable cache keys and values can be placed in the store, similar to DiskStore.
- Serialization and deserialization take place on putting and getting from the store. This means that the off-heap store is slower in an absolute sense (around 10 times slower than the MemoryStore), but this theoretical difference disappears due to two effects:
    - the MemoryStore holds the hottest subset of data from the off-heap store, already in deserialized form
    - when the GC involved with larger heaps is taken into account, the off-heap store is faster on average

Suitable Element Types

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

See the BigMemory chapter for more details.

DiskStore

The `DiskStore` provides a disk spooling facility.

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 when 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.

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.

Enterprise DiskStore

The commercial version of Ehcache 2.4 introduced an upgraded disk store. Improvements include:

- Upgraded fragmentation control/management to be the same as offheap
- No Heap used for fragmentation management or keys
- Much more predictable write latency up to caches over half a terabyte.
- SSD aware and optimised.

Throughput is approximately 110,000 operations/s which translates to around 60MB/sec on a 10k rpm hard drive with even higher rates on SSD drives, for which the Disk

Storage

Files

The disk store creates a data file for each cache on startup called ".data". If the `DiskStore` is configured to be persistent, an index file called "**cache name**.index" is created 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> Configuration 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.dir - A system property you would normally specify on the command line—for example, java -Dehcache.disk.store.dir=/u01/myapp/diskdir ...

Subdirectories can be specified below the system property, for example:

```
java.io.tmpdir/one
```

becomes, on a Unix system:

```
/tmp/one
```

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 utilization. The default value is 120 seconds.

Eviction

If a cache's disk store has a limited size, Elements will be evicted from the `DiskStore` when it exceeds this limit. The LFU algorithm is used for these evictions. It is not configurable or changeable.

Serializable Objects

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

Safety

`DiskStore`s are thread safe.

Persistence

`DiskStore` persistence is controlled by the diskPersistent configuration element. If false or omitted, `DiskStores` 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 in 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 had been a dirty shutdown, Ehcache will delete the dirty index and data files upon restart.
- 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 deserialized, it is deleted, and null is returned. These measures prevent corrupt and inconsistent data from being returned.

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 other 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.

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.

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 that: * The serialization time for a Java object consisting of a large Map of String arrays was 126ms, where the 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.

Some Configuration Examples

These examples show how to allocate 8GB of machine memory to different stores. It assumes a data set of 7GB - say for a cache of 7M items (each 1kb in size).

Those who want minimal application response-time variance (or minimizing GC pause times), will likely want all the cache to be off-heap. Assuming that 1GB of heap is needed for the rest of the app, they will set their Java config as follows:

```
java -Xms1G -Xmx1G -XX:maxDirectMemorySize=7G
```

And their Ehcache config as:

```
<cache
 maxEntriesLocalHeap=100
 overflowToOffHeap="true"
 maxBytesLocalOffHeap="6G"
... />
```

NOTE: Direct Memory and Off-heap Memory Allocations To accommodate server communications layer requirements, the value of maxDirectMemorySize must be greater than the value of maxBytesLocalOffHeap. The exact amount greater depends upon the size of maxBytesLocalOffHeap. The minimum is 256MB, but if you allocate 1GB more to the maxDirectMemorySize, it will certainly be sufficient. The server will only use what it needs and the rest will remain available.

Those who want best possible performance for a hot set of data, while still reducing overall application repsonse time variance, will likely want a combination of on-heap and off-heap. The heap will be used for the hot set, the offheap for the rest. So, for example if the hot set is 1M items (or 1GB) of the 7GB data. They will set their Java config as follows

```
java -Xms2G -Xmx2G -XX:maxDirectMemorySize=6G
```

And their Ehcache config as:

```
<cache
  maxEntriesLocalHeap=1M
  overflowToOffHeap="true"
  maxBytesLocalOffHeap="5G"
  ...>
```

This configuration will compare VERY favorably against the alternative of keeping the less-hot set in a database (100x slower) or caching on local disk (20x slower).

Where the data set is small and pauses are not a problem, the whole data set can be kept on heap:

```
<cache
  maxEntriesLocalHeap=1M
  overflowToOffHeap="false"
  ...>
```

Where latency isn't an issue overflow to disk can be used:

```
<cache
  maxEntriesLocalHeap=1M
  overflowToOffDisk="true"
  ...>
```

Performance Considerations

Relative Speeds

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.

Always use some amount of Heap

For performance reasons, Ehcache should always use as much heap memory as possible without triggering GC pauses. Use BigMemory (the off-heap store) to hold the data that cannot fit in heap without causing GC pauses.

Using Ehcache

Introduction

Ehcache can be used directly. It can also be used with the popular Hibernate Object/Relational tool and Java EE Servlet Caching. This page is a quick guide to get you started. The rest of the documentation can be explored for a deeper understanding.

General-Purpose Caching

- Download Ehcache ›

  Beginning with Ehcache 1.7.1, Ehcache depends on SLF4J (http://www.slf4j.org) for logging. SLF4J is a logging framework with a choice of concrete logging implementations. See the chapter on Logging for configuration details.
- Use Java 1.5 or 1.6.
- Place the Ehcache jar into your 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.

Cache Usage Patterns

There are several common access patterns when using a cache. Ehcache supports the following patterns:

- cache-aside (or direct manipulation)
- cache-as-sor (a combination of read-through and write-through or write-behind patterns)
- read-through
- write-through
- write-behind (or write-back)

cache-aside

Here, application code uses the cache directly.

This means that application code which accesses the system-of-record (SOR) should consult the cache first, and if the cache contains the data, then return the data directly from the cache, bypassing the SOR.

Otherwise, the application code must fetch the data from the system-of-record, store the data in the cache, and then return it.

When data is written, the cache must be updated with the system-of-record. This results in code that often looks like the following pseudo-code:

```
public class MyDataAccessClass
{
  private final Ehcache cache;
  public MyDataAccessClass(Ehcache cache)
  {
    this.cache = cache;
  }

  /* read some data, check cache first, otherwise read from sor */
  public V readSomeData(K key)
  {
     Element element;
     if ((element = cache.get(key)) != null) {
         return element.getValue();
     }
      // note here you should decide whether your cache
     // will cache 'nulls' or not
     if (value = readDataFromDataStore(key)) != null) {
         cache.put(new Element(key, value));
     }
     return value;
  }
  /* write some data, write to sor, then update cache */
  public void writeSomeData(K key, V value)
  {
     writeDataToDataStore(key, value);
     cache.put(new Element(key, value);
  }
```

cache-as-sor

The cache-as-sor pattern implies using the cache as though it were the primary system-of-record (SOR). The pattern delegates SOR reading and writing activies to the cache, so that application code is absolved of this responsibility.

To implement the cache-as-sor pattern, use a combination of the following read and write patterns:

- read-through
- write-through or write-behind

Advantages of using the cache-as-sor pattern are:

- less cluttered application code (improved maintainability)
- easily choose between write-through or write-behind strategies on a per-cache basis (use only configuration)
- allow the cache to solve the "thundering-herd" problem

A disadvantage of using the cache-as-sor pattern is:

- less directly visible code-path

read-through

The read-through pattern mimics the structure of the cache-aside pattern when reading data. The difference is that you must implement the `CacheEntryFactory` interface to instruct the cache how to read objects on a cache miss, and you must wrap the Ehcache instance with an instance of `SelfPopulatingCache`. Compare the appearance of the read-through pattern code to the code provided in the cache-aside pattern. (The full example is provided at the end of this document that includes a read-through and write-through implementation).

write-through

The write-through pattern mimics the structure of the cache-aside pattern when writing data. The difference is that you must implement the `CacheWriter` interface and configure the cache for write-through or write-behind. A write-through cache writes data to the system-of-record in the same thread of execution, therefore in the common scenario of using a database transaction in context of the thread, the write to the database is covered by the transaction in scope. More details (including configuration settings) can be found in the User Guide chapter on Write-through and Write-behind Caching.

write-behind

The write-behind pattern changes the timing of the write to the system-of-record. Rather than writing to the System of Record in the same thread of execution, write-behind queues the data for write at a later time.

The consequences of the change from write-through to write-behind are that the data write using write-behind will occur outside of the scope of the transaction.

This often-times means that a new transaction must be created to commit the data to the system-of-record that is separate from the main transaction. More details (including configuration settings) can be found in the User Guide chapter on Write-through and Write-behind Caching.

cache-as-sor example

```
public class MyDataAccessClass
{
private final Ehcache cache;
public MyDataAccessClass(Ehcache cache)
{
   cache.registerCacheWriter(new MyCacheWriter());
   this.cache = new SelfPopulatingCache(cache);
}
/* read some data - notice the cache is treated as an SOR.
* the application code simply assumes the key will always be available
*/
public V readSomeData(K key)
{
   return cache.get(key);
}
/* write some data - notice the cache is treated as an SOR, it is
* the cache's responsibility to write the data to the SOR.
*/
public void writeSomeData(K key, V value)
{
   cache.put(new Element(key, value);
}
/**
* Implement the CacheEntryFactory that allows the cache to provide
* the read-through strategy
*/
private class MyCacheEntryFactory implements CacheEntryFactory
{
   public Object createEntry(Object key) throws Exception
   {
       return readDataFromDataStore(key);
   }
}
```

cache-as-sor example

```
/**
* Implement the CacheWriter interface which allows the cache to provide
* the write-through or write-behind strategy.
*/
private class MyCacheWriter implements CacheWriter
   public CacheWriter clone(Ehcache cache) throws CloneNotSupportedException;
   {
       throw new CloneNotSupportedException();
   }
    public void init() { }
   void dispose() throws CacheException { }
    void write(Element element) throws CacheException;
   {
       writeDataToDataStore(element.getKey(), element.getValue());
   }
    void writeAll(Collection elements) throws CacheException
   {
       for (Element element : elements) {
           write(element);
       }
   }
    void delete(CacheEntry entry) throws CacheException
   {
       deleteDataFromDataStore(element.getKey());
   }
    void deleteAll(Collection entries) throws CacheException
   {
       for (Element element : elements) {
           delete(element);
       }
   }
}
}
```

Copy Cache

A Copy Cache can have two behaviors: it can copy Element instances it returns, when `copyOnRead` is true and copy elements it stores, when `copyOnWrite` to true.

A copy on read cache can be useful when you can't let multiple threads access the same Element instance (and the value it holds) concurrently. For example, where the programming model doesn't allow it, or you want to isolate changes done concurrently from each other.

Copy on write also lets you determine exactly what goes in the cache and when. i.e. when the value that will be in the cache will be in state it was when it actually was put in cache. *All mutations to the value, or the element, after the put operation will not be reflected in the cache*.

A concrete example of a copy cache is a Cache configured for `XA`. It will always be configured `copyOnRead` and `copyOnWrite` to provide proper transaction isolation and clear transaction boundaries (the state the objects are in at commit time is the state making it into the cache). By default, the copy operation will be performed using standard Java object serialization. We do recognize though that for some applications this might not be good (or fast) enough. You can configure your own `CopyStrategy` which will be used to perform these copy operations. For example, you could easily implement use cloning rather than Serialization.

More information on configuration can be found here: copyOnRead and copyOnWrite cache configuration.

Specific Technologies

Distributed Caching

Distributed Ehcache combines the power of the Terracotta platform with the ease of Ehcache application-data caching. Ehcache supports distributed caching with two lines of configuration.

By integrating Enterprise Ehcache with the Terracotta platform, you can take advantage of BigMemory and expanded Terracotta Server Arrays to greatly scale your application and cluster.

The distributed-cache documentation covers how to configure Ehcache in a Terracotta cluster and how to use its API in your application.

Hibernate

- Perform the same steps as for general-purpose caching (above).
- Create caches in ehcache.xml.

See the Hibernate Caching chapter for more information.

Java EE Servlet Caching

- Perform the same steps as for general-purpose caching above.
- 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.

RESTful and SOAP Caching with the Cache Server

- Download the ehcache-standalone-server from https://sourceforge.net/projects/ehcache/files/ehcache-server.
- 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 on the Cache Server page. You can use Java or any other programming language with the Cache Server.

See the Cache Server page for more information.

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.

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.

Building and Testing Ehcache

Introduction

This page is intended for those who want to create their own Ehcache or distributed Ehcache build rather than use the packed kit.

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.

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`

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.

Java Requirements and Dependencies

Java Requirements

- Current Ehcache releases require Java 1.5 and 1.6 at runtime.
- Ehcache 1.5 requires Java 1.4. Java 1.4 is not supported with Terracotta distributed Ehcache.
- The ehcache-monitor module, which provides management and monitoring, will work with Ehcache 1.2.3 but only for Java 1.5 or higher.

Mandatory Dependencies

- Ehcache core 1.6 through to 1.7.0 has no dependencies.
- Ehcache core 1.7.1 depends on SLF4J (http://www.slf4j.org), an increasingly commonly used logging framework which provides a choice of concrete logging implementation. See the page on Logging for configuration details.

Other modules have dependencies as specified in their maven POMs.

Maven Snippet

To include Ehcache in your project, use:

```
<dependency>
  <groupId>net.sf.ehcache</groupId>
  <artifactId>ehcache</artifactId>
  <version>2.3.1</version>
  <type>pom</type>
</dependency>
```

Note: Be sure to substitute the version number above with the version number of Ehcache that you want to use.

If using Terracotta Distributed Ehcache, also add:

```
<dependency>
  <groupId>org.terracotta</groupId>
  <artifactId>terracotta-toolkit-1.4-runtime</artifactId>
  <version>4.0.0</version>
</dependency>

<repositories>
  <repository>
    <id>terracotta-repository</id>
    <url>http://www.terracotta.org/download/reflector/releases</url>
    <releases>
      <enabled>true</enabled>
    </releases>
  </repository>
</repositories>
```

Be sure to check the dependency versions for compatibility. Versions released in a single kit are guaranteed compatible.

Distributed Cache Development with Maven and Ant

With a Distributed Ehcache, there is a Terracotta Server Array. At development time, this necessitates running a server locally for integration and/or interactive testing. There are plugins for Maven and Ant to simplify and automate this process.

For Maven, Terracotta has a plugin available which makes this very simple.

Setting up for Integration Testing

```
<pluginRepositories>
   <pluginRepository>
       <id>terracotta-snapshots</id>
       <url>http://www.terracotta.org/download/reflector/maven2</url>
       <releases>
           <enabled>true</enabled>
       </releases>
       <snapshots>
```

```
            <enabled>true</enabled>
        </snapshots>
    </pluginRepository>
</pluginRepositories>
<plugin>
    <groupId>org.terracotta.maven.plugins</groupId>
    <artifactId>tc-maven-plugin</artifactId>
    <version>1.5.1</version>
    <executions>
        <execution>
            <id>run-integration</id>
            <phase>pre-integration-test</phase>
            <goals>
                <goal>run-integration</goal>
            </goals>
        </execution>
        <execution>
            <id>terminate-integration</id>
            <phase>post-integration-test</phase>
            <goals>
                <goal>terminate-integration</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Note: Be sure to substitute the version number above with the current version number.

Interactive Testing

To start Terracotta:

```
mvn tc:start
```

To stop Terracotta:

```
mvn tc:stop
```

See the Terracotta Forge for a complete reference.

Configuration Overview

The following sections provide a documentation Table of Contents and additional information sources about Ehcache configuration.

Configuration Table of Contents

Topic	Description
Introduction	The basics of cache configuration with Ehcache, including dynamically changing cache configuration, cache warming, and copyOnRead/copyOnWrite cache configuration.
BigMemory	Introduction to BigMemory, how to configure Ehcache with BigMemory, performance comparisons, an FAQ, and more.
Sizing Caches	Tuning Ehcache often involves sizing cached data appropriately. Ehcache provides a number of ways to size the different data tiers using simple cache-configuration sizing attributes. This page explains simplified tuning of cache size by configuring dynamic allocation of memory and automatic load balancing.
Expiration, Pinning, and Eviction	The architecture of an Ehcache node can include a number of tiers that store data. One of the most important aspects of managing cached data involves managing the life of the data in each tier. This page covers managing data life in Ehcache and the Terracotta Server Array, including the pinning features of Automatic Resource Control (ARC).
Nonstop Cache	A nonstop (non-blocking) cache allows certain cache operations to proceed on clients that have become disconnected from the cluster, or to proceed when cache operations cannot complete by the nonstop timeout value. This can be useful in meeting SLA requirements, responding to node failures, building a more robust High Availability cluster, and more. This page covers configuring nonstop cache, nonstop timeouts and behaviors, and nonstop exceptions.
UnlockedReadsView	With this API, you can have both the unlocked view and a strongly consistent cache at the same time. UnlocksReadView provides an eventually consistent view of a strongly consistent cache. Views of data are taken without regard to that data's consistency, and writes are not affected by UnlockedReadsView. This page covers creating an UnlockedReadsView and provides a download link and an FAQ.
Distributed-Cache Configuration	The basic configuration guide for Distributed Ehcache (Ehcache with Terracotta clustering), this page also includes CacheManager configuration and Terracotta clustering configuration elements.
Distributed-Cache Default Configuration	A number of properties control the way the Terracotta Server Array and Ehcache clients perform in a Terracotta cluster. Some of these properties are set in the Terracotta configuration file, others are set in the Ehcache configuration file, and a few must be set programmatically. This page details the most important of these properties and shows their default values.

Hit the Ground Running

Popular topics in Configuration:

- Cache Warming
- Handling JVM startup and shutdown with BigMemory

- Sizing Distributed Caches
- Terracotta Clustering Configuration Elements

Additional Information about Configuration

The following pages provide additional information about Ehcache configuration:

- Discussion of Data Freshness and Expiration
- Enabling Terracotta Support Programmatically
- Adding and Removing Caches Programmatically
- Creating Caches Programmatically

Cache Configuration

Introduction

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 all in 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.
- A defaultCache configuration exists and will always be loaded.

  While a defaultCache configuration is not required, an error is generated if caches are created by name (programmatically) with no defaultCache loaded.

The Ehcache documentation focuses on XML declarative configuration. Programmatic configuration is explored in certain examples and is documented in Javadocs.

Ehcache is redistributed by lots of projects, some of which may not provide a sample Ehcache XML configuration file (or they provide an outdated one). If one is not provided, download Ehcache. The latest ehcache.xml and ehcache.xsd are provided in the distribution.

Dynamically Changing Cache Configuration

After a Cache has been started, its configuration is not generally changeable. However, since Ehcache 2.0, certain cache configuration parameters can be modified dynamically at runtime. In the current version of Ehcache, this includes the following:

- timeToLive

  The maximum number of seconds an element can exist in the cache regardless of use. The element expires at this limit and will no longer be returned from the cache. The default value is 0, which means no TTL eviction takes place (infinite lifetime).
- timeToIdle

  The maximum number of seconds an element can exist in the cache without being accessed. The element expires at this limit and will no longer be returned from the cache. The default value is 0, which means no TTI eviction takes place (infinite lifetime).
- Local sizing attributes maxEntriesLocalHeap, maxBytesLocalHeap, maxBytesLocalOffHeap, maxEntriesLocalDisk, maxBytesLocalDisk.
- memory-store eviction policy
- CacheEventListeners can be added and removed dynamically

Note that the `eternal` attribute, when set to "true", overrides `timeToLive` and `timeToIdle` so that no expiration can take place. This example shows how to dynamically modify the cache configuration of running cache:

```
Cache cache = manager.getCache("sampleCache");
CacheConfiguration config = cache.getCacheConfiguration();
config.setTimeToIdleSeconds(60);
config.setTimeToLiveSeconds(120);
config.setmaxEntriesLocalHeap(10000);
config.setmaxEntriesLocalDisk(1000000);
```

Dynamic cache configurations can also be frozen to prevent future changes:

```
Cache cache = manager.getCache("sampleCache");
cache.disableDynamicFeatures();
```

In `ehcache.xml`, you can disable dynamic configuration by setting the `<ehcache>` element's `dynamicConfig` attribute to "false".

Dynamic Configuration Changes for Distributed Cache

Just as for a standalone cache, mutating the configuration of a distributed cache requires access to the set methods of `cache.getCacheConfiguration().`

The following table provides information dynamically changing common cache configuration options in a Terracotta cluster. The table's Scope column, which specifies where the configuration is in effect, can have one of the following values:

- Client â— The Terracotta client where the CacheManager runs.
- TSA â— The Terracotta Server Array for the cluster.
- BOTH â— Both the client and the TSA.

Note that configuration options whose scope covers "BOTH" are distributed and therefore affect a cache on all clients.

Configuration Option	**Dynamic**	**Scope**	**Notes**
Cache name	NO	TSA	
Nonstop	NO	Client	Enable High Availability
Timeout	YES	Client	For nonstop.
Timeout Behavior	YES	Client	For nonstop.
Immediate Timeout When Disconnected	YES	Client	For nonstop.
Time to Idle	YES	BOTH	
Maximum Entries or Bytes in Local Stores	YES	Client	This and certain other sizing attributes that are part of ARC may be pooled by the CacheManager, creating limitations on how they can be changed.
Time to Live	YES	BOTH	
Maximum Elements on Disk	YES	TSA	
Overflow to Disk	N/A	N/A	
Persist to Disk	N/A	N/A	
Disk Expiry Thread Interval	N/A	N/A	
Disk Spool Buffer Size	N/A	N/A	
Overflow to Off-Heap	N/A	N/A	
Maximum Off-heap	N/A	N/A	Maximum off-heap memory allotted to the TSA.
Eternal	YES	BOTH	
Clear on Flush	NO	Client	
Copy on Read	NO	Client	
Copy on Write	NO	Client	
Memory Store Eviction Policy	NO	Client	
Statistics	YES	Client	Cache statistics. Change dynamically with `cache.setStatistics(boolean)` method.
Logging	NO	Client	Ehcache and Terracotta logging is specified in configuration. However, cluster events can be set dynamically.
Consistency	NO	Client	It is possible to switch to and from bulk mode.
Synchronous Writes	NO	Client	

To apply non-dynamic L1 changes, remove the existing cache and then add (to the same CacheManager) a new cache with the same name as the removed cache, and which has the new configuration. Restarting the CacheManager with an updated configuration, where all cache names are the same as in the previous configuration, will also apply non-dynamic L1 changes.

Memory-Based Cache Sizing (Ehcache 2.5 and higher)

Historically Ehcache has only permitted sizing of caches in the Java heap (the OnHeap store) and the disk (DiskStore). BigMemory introduced the OffHeap store, where sizing of caches is also allowed.

To learn more about sizing caches, see How to Size Caches.

Pinning of Caches and Elements in Memory (2.5 and higher)

Pinning of caches or specific elements is discussed in Pinning, Expiration, and Eviction.

Cache Warming for multi-tier Caches

(Ehcache 2.5 and higher)

When a cache starts up, the On-Heap and Off-Heap stores are always empty. Ehcache provides a BootstrapCacheLoader mechanism to overcome this. The BootstrapCacheLoader is run before the cache is set to alive. If synchronous, loading completes before the CacheManager starts, or if asynchronous, the CacheManager starts but loading continues agressively rather than waiting for elements to be requested, which is a lazy loading approach.

Replicated caches provide a boot strap mechanism which populates them. For example following is the JGroups bootstrap cache loader:

```
<bootstrapCacheLoaderFactory class="net.sf.ehcache.distribution.jgroups.JGroupsBootstrapCacheLoad
```

There are two new bootstrapCacheLoaderFactory implementations: one for standalone caches with DiskStores, and one for Terracotta Distributed caches.

DiskStoreBootstrapCacheLoaderFactory

The DiskStoreBootstrapCacheLoaderFactory loads elements from the DiskStore to the On-Heap Store and the Off-Heap store until either:

- the memory stores are full
- the DiskStore has been completely loaded

Configuration

The DiskStoreBootstrapCacheLoaderFactory is configured as follows:

```
<bootstrapCacheLoaderFactory class="net.sf.ehcache.store.DiskStoreBootstrapCacheLoaderFactory" pr
```

TerracottaBootstrapCacheLoaderFactory

The TerracottaBootstrapCacheLoaderFactory loads elements from the Terracotta L2 to the L1 based on what it was using the last time it ran. If this is the first time it has been run it has no effect.

It works by periodically writing the keys used by the L1 to disk.

Configuration

The TerracottaStoreBootstrapCacheLoaderFactory is configured as follows:

```
<bootstrapCacheLoaderFactory class="net.sf.ehcache.terracotta.TerracottaBootstrapCacheLoaderFacto
properties="bootstrapAsynchronously=true,
           directory=dumps,
           interval=5,
           immediateShutdown=false,
           snapshotOnShutDown=true,
           doKeySnapshot=false,
           useDedicatedThread=false"/>
```

The configuration properties are:

- bootstrapAsynchronously: Whether to bootstrap asynchronously or not. Asynchronous bootstrap will allow the cache to start up for use while loading continues.
- directory: the directory that snapshots are created in. By default this will use the CacheManager's DiskStore path.
- interval: interval in seconds between each key snapshot. Default is every 10 minutes (600 seconds). Cache performance overhead increases with more frequent snapshots and is dependent on such factors as cache size and disk speed. Thorough testing with various values is highly recommended.
- immediateShutdown: whether, when shutting down the Cache, it should let the keysnapshotting (if in progress) finish or terminate right away. Defaults to true.
- snapshotOnShutDown: Whether to take the local key-set snapshot when the Cache is disposed. Defaults to false.
- doKeySnapshot : Set to false to disable keysnapshotting. Default is true. Enables loading from an existing snapshot without taking new snapshots after the existing one been loaded (stable snapshot). Or to only snapshot at cache disposal (see snapshotOnShutdown).
- useDedicatedThread : By default, each CacheManager uses a thread pool of 10 threads to do the snapshotting. If you want the cache to use a dedicated thread for the snapshotting, set this to true

Key snapshots will be in the diskstore directory configured at the cachemanager level.

One file is created for each cache with the name `<cacheName>.keySet`.

In case of a abrupt termination, while new snapshots are being written they are written using the extension `.temp` and then after the write is complete the existing file is renamed to `.old`, the `.temp` is renamed to `.keyset` and finally the `.old` file is removed. If an abrupt termination occurs you will see some of these files in the directory which will be cleaned up on the next startup.

Like other DiskStore files, keyset snapshot files can be migrated to other nodes for warmup.

If between restarts, the cache can't hold the entire hot set locally, the Loader will stop loading as soon as the on-heap (or off-heap) store has been filled.

copyOnRead and copyOnWrite cache configuration

A cache can be configured to copy the data, rather than return reference to it on get or put. This is configured using the `copyOnRead` and `copyOnWrite` attributes of cache and defaultCache elements in your configuration or programmatically as follows:

copyOnRead and copyOnWrite cache configuration

```
CacheConfiguration config = new CacheConfiguration("copyCache", 1000).copyOnRead(true).copyOnWrit
Cache copyCache = new Cache(config);
```

The default configuration will be false for both options.

In order to copy elements on put()-like and/or get()-like operations, a CopyStrategy is being used. The default implementation uses serialization to copy elements. You can provide your own implementation of `net.sf.ehcache.store.compound.CopyStrategy` like this:

```
<cache name="copyCache"
    maxEntriesLocalHeap="10"
    eternal="false"
    timeToIdleSeconds="5"
    timeToLiveSeconds="10"
    overflowToDisk="false"
    copyOnRead="true"
    copyOnWrite="true">
  <copyStrategy class="com.company.ehcache.MyCopyStrategy"/>
</cache>
```

Per cache, a single instance of your `CopyStrategy` is used. Therefore, in your implementation of `CopyStrategy.copy(T)`, T has to be thread-safe.

A copy strategy can be added programmatically in the following:

```
CacheConfiguration cacheConfiguration = new CacheConfiguration("copyCache", 10);

CopyStrategyConfiguration copyStrategyConfiguration = new CopyStrategyConfiguration();
copyStrategyConfiguration.setClass("com.company.ehcache.MyCopyStrategy");

cacheConfiguration.addCopyStrategy(copyStrategyConfiguration);
```

Special System Properties

net.sf.ehcache.disabled

Setting this system property to `true` (using `java -Dnet.sf.ehcache.disabled=true` in the Java command line) disables caching in ehcache. If disabled, no elements can be added to a cache (puts are silently discarded).

net.sf.ehcache.use.classic.lru

When LRU is selected as the eviction policy, set this system property to `true` (using `java -Dnet.sf.ehcache.use.classic.lru=true` in the Java command line) to use the older LruMemoryStore implementation. This is provided for ease of migration.

ehcache.xsd

Ehcache configuration files must be comply with the Ehcache XML schema, `ehcache.xsd`. It can be downloaded from http://ehcache.org/ehcache.xsd.

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
     maxEntriesLocalHeap="10000"
     eternal="false"
     timeToIdleSeconds="120"
     timeToLiveSeconds="120"
     overflowToDisk="true"
     maxEntriesLocalDisk="10000000"
     diskPersistent="false"
     diskExpiryThreadIntervalSeconds="120"
     memoryStoreEvictionPolicy="LRU"
  />
</ehcache>
```

Update Checker

The update checker is used to see if you have the latest version of Ehcache. It is also used to get non-identifying feedback on the OS architectures using Ehcache. To disable the check, do one of the following:

By System Property

```
-Dnet.sf.ehcache.skipUpdateCheck=true
```

By Configuration

The outer `ehcache` element takes an `updateCheck` attribute, which is set to false as in the following example.

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="ehcache.xsd"
    updateCheck="false" monitoring="autodetect"
    dynamicConfig="true">
```

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.

ehcache.xml and Other Configuration Files

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` is included in the Ehcache distribution. It 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.

Ehcache Configuration With Terracotta Clustering

See the distributed-cache configuration guidelines for more information on configuration with distributed caches in a Terracotta cluster.

BigMemory

Introduction

BigMemory is a pure Java product from Terracotta that permits caches to use an additional type of memory store outside the object heap. It is packaged for use in Enterprise Ehcache as a snap-in job store called the "off-heap store." If Enterprise Ehcache is distributed in a Terracotta cluster, you can configure BigMemory in both Ehcache (the Terracotta client or L1) and in the Terracotta Server Array (the L2).

This off-heap store, which is not subject to Java GC, is 100 times faster than the DiskStore and allows very large caches to be created (we have tested this with over 350GB).

Because off-heap data is stored in bytes, there are two implications:

- Only Serializable cache keys and values can be placed in the store, similar to DiskStore.
- Serialization and deserialization take place on putting and getting from the store. This means that the off-heap store is slower in an absolute sense (around 10 times slower than the MemoryStore), but this theoretical difference disappears due to two effects:
    - the MemoryStore holds the hottest subset of data from the off-heap store, already in deserialized form
    - when the GC involved with larger heaps is taken into account, the off-heap store is faster on average.

For a tutorial on Ehcache BigMemory, see BigMemory for Enterprise Ehcache Tutorial.

Configuration

Configuring Caches to Overflow to Off-heap

Configuring a cache to use an off-heap store can be done either through XML or programmatically.

If using distributed cache with strong consistency, a large number of locks may need to be stored in client and server heaps. In this case, be sure to test the cluster with the expected data set to detect situations where OutOfMemory errors are likely to occur. In addition, the overhead from managing the locks is likely to reduce performance.

Declarative Configuration

The following XML configuration creates an off-heap cache with an in-heap store (maxEntriesLocalHeap) of 10,000 elements which overflow to a 1-gigabyte off-heap area.

```
<ehcache updateCheck="false" monitoring="off" dynamicConfig="false">
    <defaultCache maxEntriesLocalHeap="10000"
                  eternal="true"
                  memoryStoreEvictionPolicy="LRU"
                  statistics="false" />

    <cache name="sample-offheap-cache"
           maxEntriesLocalHeap="10000"
           eternal="true"
           memoryStoreEvictionPolicy="LRU"
           overflowToOffHeap="true"
           maxBytesLocalOffHeap="1G"/>
</ehcache>
```

The configuration options are:

overflowToOffHeap

Values may be true or false. When set to true, enables the cache to utilize off-heap memory storage to improve performance. Off-heap memory is not subject to Java GC cycles and has a size limit set by the Java property MaxDirectMemorySize. The default value is false.

maxBytesLocalOffHeap

Sets the amount of off-heap memory available to the cache and is in effect only if `overflowToOffHeap` is true. The minimum amount that can be allocated is 128MB. There is no maximum.

For more information about sizing caches, refer to How To Size Caches.
NOTE: Heap Configuration When Using an Off-heap Store You should set maxEntriesLocalHeap to at least 100 elements when using an off-heap store to avoid performance degradation. Lower values for maxEntriesLocalHeap trigger a warning to be logged.

Programmatic Configuration

The equivalent cache can be created using the following programmatic configuration:

```
public Cache createOffHeapCache()
{
  CacheConfiguration config = new CacheConfiguration("sample-offheap-cache", 10000)
  .overflowToOffHeap(true).maxBytesLocalOffHeap("1G");
  Cache cache = new Cache(config);
  manager.addCache(cache);
  return cache;
}
```

Adding The License

Enterprise Ehcache trial download comes with a license key good for 30 days. Use this key to activate the off-heap store. It can be added to the classpath or via a system property.

Configuring the License in the Classpath

Add the `terracotta-license.key` to the root of your classpath, which is also where you add ehcache.xml. It will be automatically found.

Configuring the License as a Java System Property

Add a `com.tc.productkey.path=/path/to/key` system property which points to the key location.

For example,

```
java -Dcom.tc.productkey.path=/path/to/key
```

Allocating Direct Memory in the JVM

In order to use these configurations, you must then use the ehcache-core-ee jar on your classpath and modify your JVM command-line to increase the amount of direct memory allowed by the JVM. You must allocate at least 256MB more to direct memory than the total off-heap memory allocated to caches.

For example, to allocate 2GB of memory in the JVM:

```
java -XX:MaxDirectMemorySize=2G ..."
```

NOTE: Direct Memory and Off-heap Memory Allocations To accommodate server communications layer requirements, the value of maxDirectMemorySize must be greater than the value of maxBytesLocalOffHeap. The exact amount greater depends upon the size of maxBytesLocalOffHeap. The minimum is 256MB, but if you allocate 1GB more to the maxDirectMemorySize, it will certainly be sufficient. The server will only use what it needs and the rest will remain available.

Advanced Configuration Options

There are some additional configuration options which can be used for fine grained control.

-XX:+UseLargePages

This is a JVM flag which is meant to improve performance of memory-hungry applications. In testing, this option gives a 5% speed improvement with a 1GB off-heap cache.

See http://andrigoss.blogspot.com/2008/02/jvm-performance-tuning.html for a discussion.

Increasing the Maximum Serialized Size of an Element that can be Stored in the OffHeapStore

While the MemoryStore and the DiskStore do not have any limits, by default the OffHeapStore has a 4MB limit for classes with high quality hashcodes, and 256KB for those with pathologically bad hashcodes. The built-in classes such as the `java.lang.Number` subclasses such as Long, Integer etc and and `String` have high quality hashcodes.

You can increase the size by setting a system property `net.sf.ehcache.offheap.cache_name.config.idealMaxSegmentSize` to the size you require.

For example,

```
net.sf.ehcache.offheap.com.company.domain.State.config.idealMaxSegmentSize=30M
```

Avoiding OS Swapping

Operating systems use swap partitions for virtual memory and are free to move less frequently used pages of memory to the swap partition. This is generally not what you want when using the OffHeapStore, as the time it takes to swap a page back in when demanded will add to cache latency.

It is recommended that you minimize swap use for maximum performance.

Avoiding OS Swapping

On Linux, you can set `/proc/sys/vm/swappiness` to reduce the risk of memory pages being swapped out. See http://lwn.net/Articles/83588/ for details of tuning this parameter. Note that there are bugs in this which were fixed in kernel 2.6.30 and higher.

Another option is to configure HugePages. See http://unixfoo.blogspot.com/2007/10/hugepages.html

Although there's a swappiness kernel parameter that can be set to zero in Linux, it is usually not enough to avoid swapping altogether. The only surefire way to avoid any kind of swapping is either (a) disabling the swap partition, with the undesirable consequences which that may bring, or (b) using HugePages, which are always mapped to physical memory and cannot be swapped out to disk.

Compressed References

The following setting applies to Java 6 or higher. Its use should be considered to make the most efficient use of memory in 64 bit mode. For the Oracle HotSpot, compressed references are enabled using the option `-XX:+UseCompressedOops`. For IBM JVMs, use `-Xcompressedrefs`. See https://wikis.oracle.com/display/HotSpotInternals/CompressedOops for details.

Controlling Over-allocation of Memory to the OffHeapStore

It is possible to over-allocate memory to the off-heap store and overrun the physical and even virtual memory available on the OS. See Slow Off-Heap Allocation for how to handle this situation.

Sample Application

The easiest way to get started is to play with a simple, sample app. Download from here a simple Maven-based application that uses the ehcache off-heap functionality.

Note: You will need to get a license key and install it as discussed above to run this.

Performance Comparisons

Checkout http://svn.terracotta.org/svn/forge/offHeap-test/ for a Maven-based performance comparison test between different store configurations.

Note: To run the test, you will need to get a demo license key and install it as discussed above.

Here are some charts from tests we have run on BigMemory.

The test machine was a Cisco UCS box running with Intel(R) Xeon(R) Processors. It had 6 2.93Ghz Xeon(R) CPUs for a total of 24 cores, with 128GB of RAM, running RHEL5.1 with Sun JDK 1.6.0_21 in 64 bit mode.

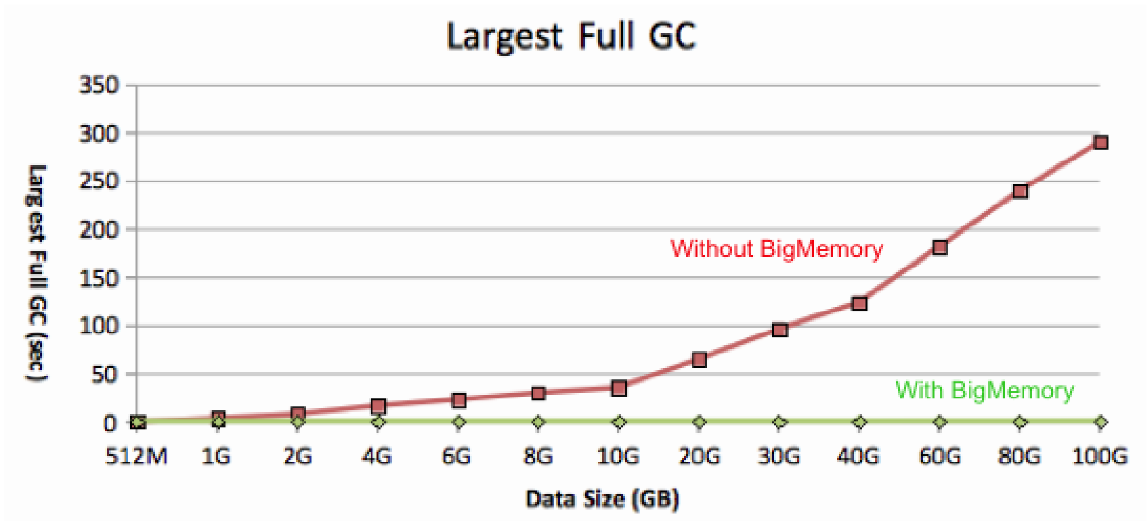
We used 50 threads doing an even mix of reads and writes with 1KB elements. We used the default garbage collection settings.

The tests all go through a load/warmup phase and then start a performance run. You can use the tests in your own environments and extend them to cover different read/write ratios, data sizes, -Xmx settings and hot sets. The full suite, which is done with `run.sh` takes 4-5 hours to complete.

The following charts show the most common caching use case. The read/write ratio is 90% reads and 10% writes. The hot set is that 90% of the time `cache.get()` will access 10% of the key set. This is representative of the the familiar Pareto distribution that is very commonly observed.

There are of course many other caching use cases. Further performance results are covered on the Further Performance Analysis page.

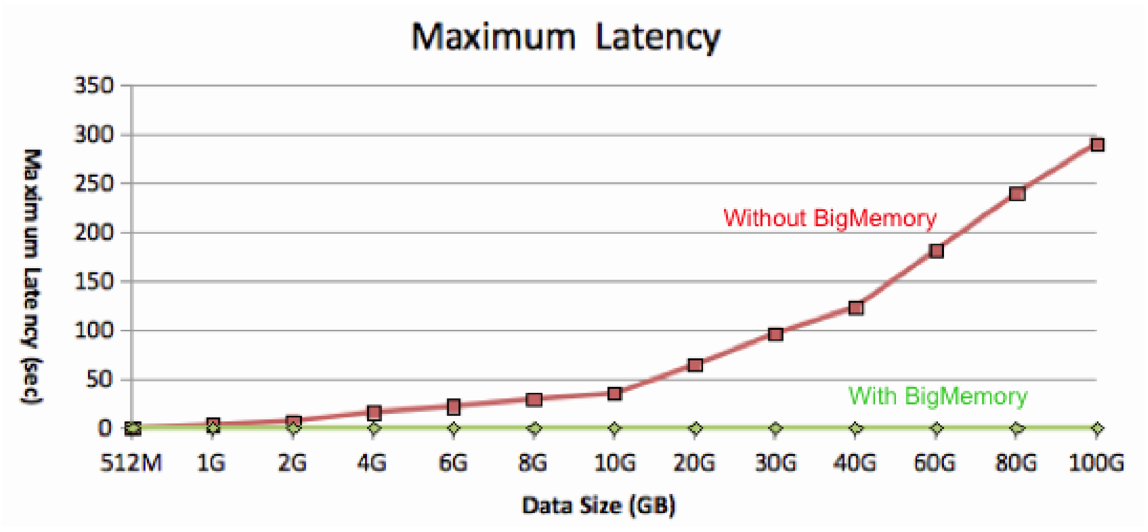
Largest Full GC



This chart shows the largest observed full GC duration which occurred during the performance run. Most non-batch applications have maximum response time SLAs. As can be seen in the chart, as data sizes grow the full GC gets worse and worse for cache held on heap, whereas off-heap remains a low constant.

The off-heap store will therefore enable applications with maximum response time SLAs to reliably meet those SLAs.

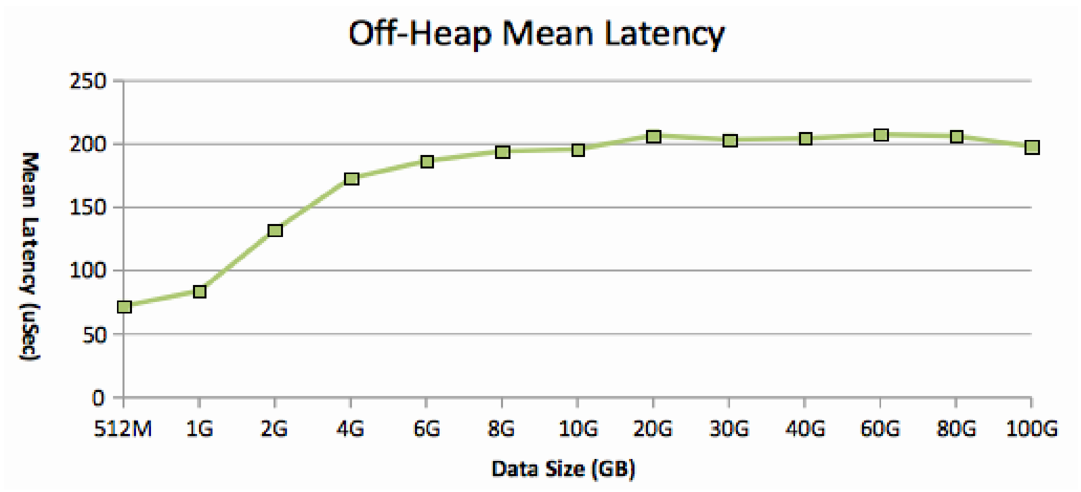
Latency



Latency

This chart shows the maximum observed latency while performing either a `cache.put()` or a `cache.get()`. It is very similar to the Full GC chart because the full GCs are causing the on-heap latencies to rise dramatically, as all threads in the test app get frozen.

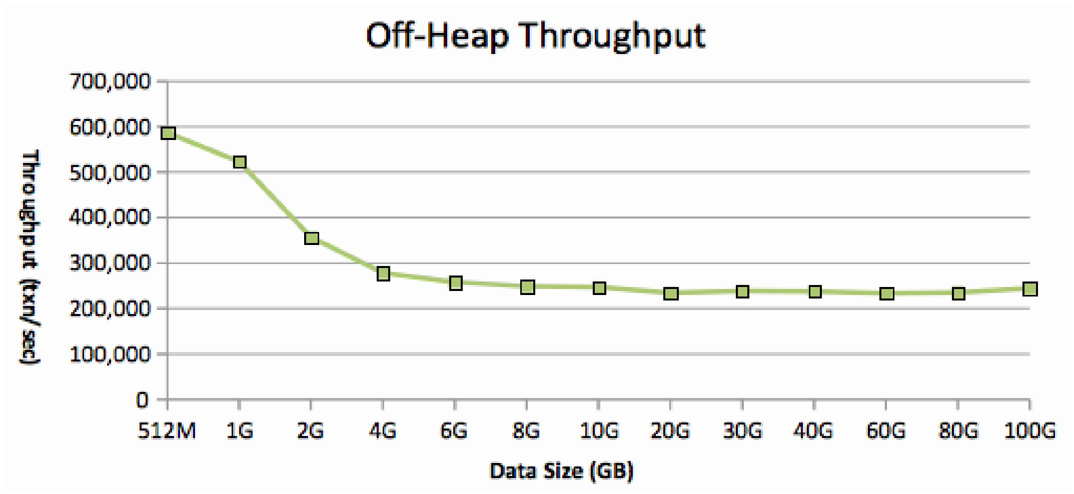
Once again the off-heap store can be observed to have a flat, low maximum latency, because any full GCs are tiny, and the cache has excellent concurrency properties.



This chart shows the off-heap mean latency in microseconds. It can be observed to be flat from 2GB up to 40GB. Further in-house testing shows that it continues to remain flat to the limits that we have tested.

Lower latencies are observed at smaller data set sizes because we use a `maxEntriesLocalHeap` setting which approximates to 200MB of on-heap store. On-heap, excluding GC effects, is faster than off-heap because there is no deserialization on gets. At lower data sizes, there is a higher probability that the small on-heap store will be hit, which is reflected in the lower average latencies.

Throughput



This chart shows the cache operations per second achieved with off-heap. It is the inverse of average latency and shows much the same thing. Once the effect of the on-heap store becomes marginal, throughput remains constant, regardless of cache size.

Throughput

Note that much larger throughputs than those shown in this chart are achievable. Throughput is affected by:

- the number of threads (more threads -> more throughput)
- the read/write ratio (reads are slightly faster)
- data payload per operation (more data implies a lower throughput in TPS but similar in bytes)
- CPU cores available and their speed (our testing shows that the CPU is always the limiting factor with enough threads. In other words, cache throughput can be increased by adding threads until all cores are utilised and then adding CPU cores - an ideal situation where the software can use as much hardware as you can throw at it.)

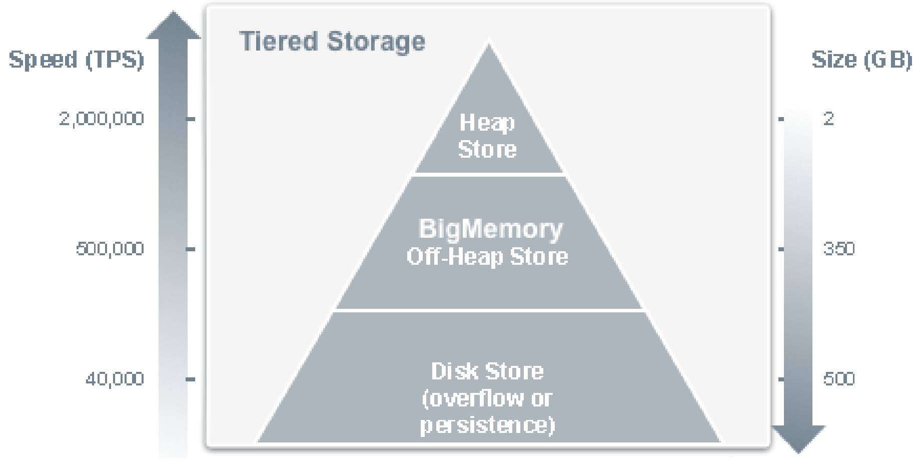
Storage

Storage Hierarchy

With the OffHeapStore, Enterprise Ehcache has three stores:

- MemoryStore - very fast storage of Objects on heap. Limited by the size of heap you can comfortably garbage collect.
- OffHeapStore - fast (one order of magnitude slower than MemoryStore) storage of Serialized objects off-heap. Limited only by the amount of RAM on your hardware and address space. You need a 64 bit OS to address higher than 2-4GB.
- DiskStore - speedy storage on disk. It is two orders of magnitude slower than the OffHeapStore but still much faster than a database or a distributed cache.

The relationship between speed and size for each store is illustrated below:



Memory Use in Each Store

As a performance optimization, and because storage gets much cheaper as you drop down through the hierarchy, we write each put to as many stores as are configured. So, if all three are configured, the Element may be present in MemoryStore, OffHeapStore and DiskStore.

The result is that each store consumes storage for itself and the other stores higher up the hierarchy. So, if the MemoryStore has 1,000,000 Elements which consume 2GB, and the OffHeapStore is configured for 8GB, then 2GB of that will be duplicate of what is in the MemoryStore. And the 8GB will also be duplicated on the

DiskStore plus the DiskStore will have what cannot fit in any of the other stores.

This needs to be taken into account when configuring the OffHeap and Disk stores.

It has the great benefit, which pays for the duplication, of not requiring copy on eviction. On eviction from a store, an Element can simply be removed. It is already in the next store down.

One further note: the MemoryStore is only populated on a read. Puts go to the OffHeapStore and then when read, are held in the MemoryStore. The MemoryStore thus holds hot items of the OffHeapStore. This will result in a difference in what can be expected to be in the MemoryStore between this implementation and the Open Source one. A "usage" for the purposes of the eviction algorithms is either a put or a get. As only gets are counted in this implementation, some differences will be observed.

Handling JVM Startup and Shutdown

So you can have a huge in-process cache. But this is not a distributed cache, so when you shut down you will lose what is in the cache. And when you start up, how long will it take to load the cache?

In caches up to a GB or two, these issues are not hugely problematic. You can often pre-load the cache on start-up before you bring the application online. Provided this only takes a few minutes, there is minimal operations impact.

But when we go to tens of GBs, these startup times are O(n), and what took 2 minutes now takes 20 minutes.

To solve this problem, we provide a new implementation of Ehcache's DiskStore, available in the Enterprise version.

You simply mark the cache `diskPersistent=true` as you normally would for a disk persistent cache.

It works as follows:

- on startup, which is immediate, the cache will get elements from disk and gradually fill the MemoryStore and the OffHeapStore.
- when running elements are written to the OffHeapStore, they are already serialized. We write these to the DiskStore asynchronously in a write-behind pattern. Tests show they can be written at a rate of 20MB/s on server-class machines with fast disks. If writes get behind, they will back up and once they reach the `diskSpoolBufferSizeMB` cache puts will be slowed while the DiskStore writer catches up. By default this buffer is 30MB but can be increased through configuration.
- When the Cache is disposed, only a final sync is required to shut the DiskStore down.

Using OffHeapStore with 32-bit JVMs

On a 32-bit operating system, Java will always start with a 32-bit data model. On 64-bit OSs, it will default to 64-bit, but can be forced into 32-bit mode with the Java command-line option `-d32`. The problem is that this limits the size of the process to 4GB. Because garbage collection problems are generally manageable up to this size, there is not much point in using the OffHeapStore, as it will simply be slower.

If you are suffering GC issues with a 32-bit JVM, then OffHeapStore can help. There are a few points to keep in mind.

- Everything has to fit in 4GB of addressable space. If you allocate 2GB of heap (with `-Xmx2g`) then you have at most 2GB left for your off-heap caches.
- Don't expect to be able to use all of the 4GB of addressable space for yourself. The JVM process requires some of it for its code and shared libraries plus any extra Operating System overhead.
- If you allocate a 3GB heap with -Xmx as well as 2047MB of off-heap memory, the virtual machine certainly won't complain at startup, but when it's time to grow the heap you will get an OutOfMemoryError.
- If you use both -Xms3G and -Xmx3G with 2047MB of off-heap memory, the virtual machine will start but then complain as soon as the OffHeapStore tries to allocate the off-heap buffers.
- Some APIs, such as java.util.zip.ZipFile on Sun 1.5 JVMs, may <mmap> files in memory. This will also use up process space and may trigger an OutOfMemoryError.

For these reasons we issue a warning to the log when OffHeapStore is used with 32-bit JVMs.

Slow Off-Heap Allocation

Based on its configuration and memory requirements, each cache may attempt to allocate off-heap memory multiple times. If off-heap memory comes under pressure due to over-allocation, the system may begin paging to disk, thus slowing down allocation operations. As the situation worsens, an off-heap buffer too large to fit in memory can quickly deplete critical system resources such as RAM and swap space and crash the host operating system.

To stop this situation from degrading, off-heap allocation time is measured to avoid allocating buffers too large to fit in memory. If it takes more than 1.5s to allocate a buffer, a warning is issued. If it takes more than 15s, then the JVM is halted with `System.exit()` (or a different method if the Security Manager prevents this).

To prevent a JVM shutdown after a 15s delay has occurred, set the `net.sf.ehcache.offheap.DoNotHaltOnCriticalAllocationDelay` system property to true. In this case, an error is logged instead.

Reducing Cache Misses

While the MemoryStore holds a hotset (a subset) of the entire data set, the off-heap store should be large enough to hold the entire data set. The frequency of cache misses begins to rise when the data is too large to fit into off-heap memory, forcing gets to fetch data from the DiskStore. More misses in turn raise latency and lower performance.

For example, tests with a 4GB data set and a 5GB off-heap store recorded no misses. With the off-heap store reduced to 4GB, 1.7 percent of cache operations resulted in misses. With the off-heap store at 3GB, misses reached 15 percent.

FAQ

What Eviction Algorithms are supported?

The pluggable MemoryStore eviction algorithms work as normal. The OffHeapStore and DiskStore use a Clock Cache, a standard paging algorithm which is an approximation of LRU.

Why do I see performance slow down and speed up in a cyclical pattern when I am filling a cache?

This is due to repartitioning in the OffHeapStore which is normal. Once the cache is fully filled the performance slow-downs cease.

What is the maximum serialized size of an object when using OffHeapStore?

Refer to "Increasing the maximum serialized size of an Element that can be stored in the OffHeapStore" in the Advanced Configuration Options section above.

Why is my application startup slower?

On startup the CacheManager will calculate the amount of off-heap storage required for all caches using off-heap stores. The memory will be allocated from the OS and zeroed out by Java. The time taken will depend on the OS. A server-class machine running Linux will take approximately half a second per GB.

We print out log messages for each 10% allocated, and also report the total time taken.

This time is incurred only once at startup. The pre-allocation of memory from the OS is one of the reasons that runtime performance is so fast.

How can I do Maven testing with BigMemory?

Maven starts java for you. You cannot add the required `-XX` switch in as a `mvn` argument.

Maven provides you with a MAVEN_OPTS environment variable you can use for this on Unix systems.

For example, to specify 1GB of MaxDirectMemorySize and then to run jetty:

```
export MAVEN_OPTS=-XX:MaxDirectMemorySize=1G
mvn jetty:run-war
```

How to Size Caches

Introduction

Tuning Ehcache often involves sizing cached data appropriately. Ehcache provides a number of ways to size the different data tiers using simple cache-configuration sizing attributes. These sizing attributes affect local memory and disk resources, allowing them to be set differently on each node.

Cache Configuration Sizing Attributes

The following table summarizes cache-sizing attributes for standalone Ehcache.

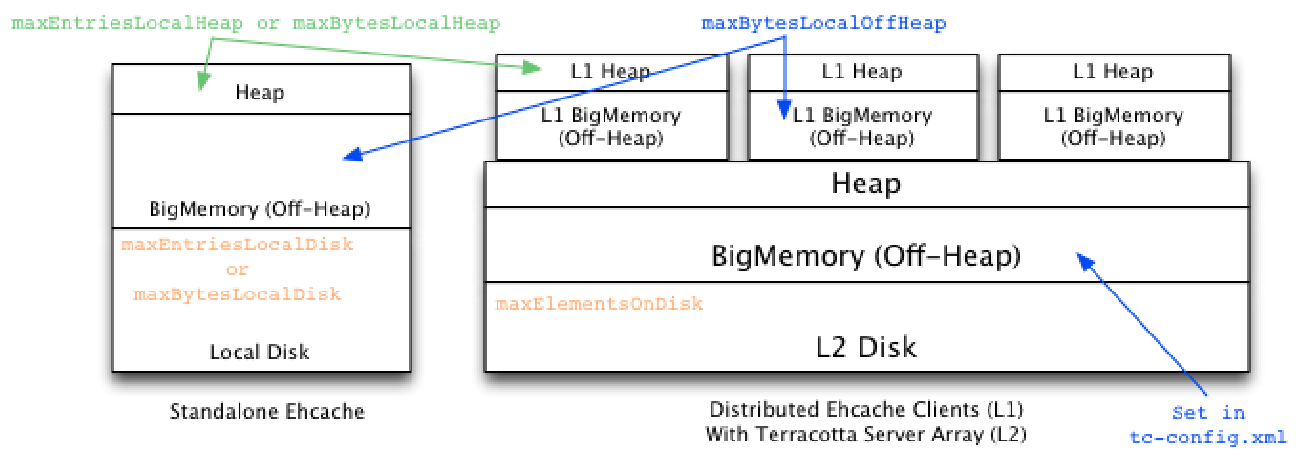
Tier	**Attribute**	**Pooling available at CacheManager Level?**	**Description**
Heap	`maxEntriesLocalHeap` `maxBytesLocalHeap`	`maxBytesLocalHeap` only	The maximum number of cache entries or bytes a cache can use in local heap memory, or, when set at the CacheManager level (maxBytesLocalHeap only), a local pool available to all caches under that CacheManager. This setting is required for every cache or at the CacheManager level.
Off-heap	`maxBytesLocalOffHeap`	Yes	The maximum number of bytes a cache can use in off-heap memory, or, when set at the CacheManager level, as a pool available to all caches under that CacheManager. This setting requires BigMemory.
Local disk	`maxEntriesLocalDisk` `maxBytesLocalDisk`	`maxBytesLocalDisk` only	The maximum number of cache entries or bytes a standalone cache can use on the local disk, or, when set at the CacheManager level (maxBytesLocalDisk only), a local pool available to all caches under that CacheManager. Distributed caches cannot use the local disk. Note that this setting is *separate from* disk settings for overflow and persistence.

The following table summarizes cache-sizing attributes for Terracotta distributed Ehcache.

Tier	**Attribute**	**Pooling available at CacheManager Level?**	**Description**
Heap	`maxEntriesLocalHeap` `maxBytesLocalHeap`	`maxBytesLocalHeap` only	The maximum number of cache entries or bytes a cache can use in local heap memory, or, when set at the CacheManager level (maxBytesLocalHeap only), a local pool available to all caches under that CacheManager. This setting is required for every cache or at the CacheManager level.
Off-heap	`maxBytesLocalOffHeap`	Yes	The maximum number of bytes a cache can use in off-heap memory, or, when set at the CacheManager level, as a pool available to all caches under that CacheManager. This setting requires BigMemory.
Local disk	N/A	N/A	Distributed caches cannot use the local disk.
Terracotta Server Array disk	`maxElementsOnDisk`	No	The number of cache elements that the Terracotta Server Array will store for a distributed cache. This value can be exceeded under certain circumstances (see below). Set on individual distributed caches only, this setting is unlimited by default.

Attributes that set a number of entries or elements take an integer. Attributes that set a memory size (bytes) use the Java -Xmx syntax (for example: "500k", "200m", "2g") or percentage (for example: "20%"). Percentages, however, can be used only in the case where a CacheManager-level pool has been configured (see below).

The following diagram illustrates the tiers and their effective sizing attributes.



Pooling Resources Versus Sizing Individual Caches

You can constrain the size of any cache on a specific tier in that cache's configuration. You can also constrain the size of all of a CacheManager's caches in a specific tier by configuring an overall size at the CacheManager level.

If there is no CacheManager-level pool specified for a tier, an individual cache claims the amount of that tier specified in its configuration. If there is a CacheManager-level pool specified for a tier, an individual cache claims that amount *from the pool*. In this case, caches with no size configuration for that tier receive an equal share of the remainder of the pool (after caches with explicit sizing configuration have claimed their portion).

For example, if CacheManager with eight caches pools one gigabyte of heap, and two caches each explicitly specify 200MB of heap while the remaining caches do not specify a size, the remaining caches will share 600MB of heap equally. Note that caches must use bytes-based attributes to claim a portion of a pool; entries-based attributes such as `maxEntriesLocal` cannot be used with a pool.

On startup, the sizes specified by caches are checked to ensure that any CacheManager-level pools are not over-allocated. If over-allocation occurs for any pool, an InvalidConfigurationException is thrown. Note that percentages should not add up to more than 100% of a single pool.

If the sizes specified by caches for any tier take exactly the entire CacheManager-level pool specified for that tier, a warning is logged. In this case, caches that do not specify a size for that tier cannot use the tier as nothing is left over.

Local Heap

A size must be provided for local heap, either in the CacheManager (`maxBytesLocalHeap` only) or in each individual cache (`maxBytesLocalHeap` or `maxEntriesLocalHeap`). Not doing so causes an InvalidConfigurationException.

If pool is configured, it can be combined with a local-heap setting in an individual cache. This allows the cache to claim a specified portion of the heap setting configured in the pool. However, in this case the cache setting must use `maxBytesLocalHeap` (same as the CacheManager).

In any case, every cache **must** have a local-heap setting, either configured explicitly or taken from the pool configured in the CacheManager.

BigMemory (Local Off-Heap)

If you are using BigMemory, off-heap sizing is available. Off-heap sizing can be configured in bytes only, never by entries.

If a CacheManager has a pool configured for off-heap, your application cannot add caches dynamically that have off-heap configuration — doing so generates an error. In addition, if any caches that used the pool are removed programmatically or through the Developer Console, other caches in the pool cannot claim the unused portion. To allot the entire off-heap pool to the remaining caches, remove the unwanted cache from the Ehcache configuration and then reload the configuration.

To use local off-heap as a data tier, a cache must have `overflowToOffHeap` set to "true". If a CacheManager has a pool configured for using off-heap, the `overflowToOffHeap` attribute is automatically set to "true" for all caches. In this case, you can prevent a specific cache from overflowing to off-heap by explicitly setting its `overflowToOffHeap` attribute to "false".

Local Disk

The local disk can be used as a data tier. Note the following:

- Distributed caches cannot use the local disk.
- To use the local disk as a data tier, a cache must have `overflowToDisk` set to "true".
- The local disk is the slowest local tier.

Cache Sizing Examples

The following examples illustrate both pooled and individual cache-sizing configurations.

Pooled Resources

The following configuration sets pools for all of this CacheManager's caches:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="100M"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="50G">
...

<cache name="Cache1" ... </cache>
<cache name="Cache2" ... </cache>
<cache name="Cache3" ... </cache>

</ehcache>
```

CacheManager CM1 automatically allocates these pools equally among its three caches. Each cache gets one third of the allocated heap, off-heap, and local disk. Note that at the CacheManager level, resources can be allocated in bytes only.

Explicitly Sizing Caches

You can explicitly allocate resources to specific caches:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="100M"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="60G">
...

<cache name="Cache1" ...
        maxBytesLocalHeap="50M"
        ...
  </cache>

<cache name="Cache2" ...
       maxBytesLocalOffHeap="5G"
       ...
  </cache>
<cache name="Cache3" ... </cache>

</ehcache>
```

In the example above, Cache1 reserves 50Mb of the 100Mb local-heap pool; the other caches divide the remaining portion of the pool equally. Cache2 takes half of the local off-heap pool; the other caches divide the remaining portion of the pool equally. Cache3 receives 25Mb of local heap, 2.5Gb of off-heap, and 20Gb of the local disk.

Caches that reserve a portion of a pool are not required to use that portion. Cache1, for example, has a fixed portion of the local heap but may have any amount of data in heap up to the configured value of 50Mb.

Note that caches must use the same sizing attributes used to create the pool. Cache1, for example, cannot use `maxEntriesLocalHeap` to reserve a portion of the pool.

Mixed Sizing Configurations

If a CacheManager does not pool a particular resource, that resource can still be allocated in cache configuration, as shown in the following example.

```
<ehcache xmlns...
        Name="CM2"
        maxBytesLocalHeap="100M">
...

<cache name="Cache4" ...
        maxBytesLocalHeap="50M"
        maxEntriesLocalDisk="100000"
        ...
  </cache>

<cache name="Cache5" ...
       maxBytesLocalOffHeap="10G"
       ...
  </cache>
<cache name="Cache6" ... </cache>
```

```
</ehcache>
```

CacheManager CM2 creates one pool (local heap). Its caches all use the local heap and are constrained by the pool setting, as expected. However, cache configuration can allocate other resources as desired. In this example, Cache4 allocates disk space for its data, and Cache5 allocates off-heap space for its data. Cache6 gets 25Mb of local heap only.

Using Percents

The following configuration sets pools for each tier:

```
<ehcache xmlns...
        Name="CM1"
        maxBytesLocalHeap="1G"
        maxBytesLocalOffHeap="10G"
        maxBytesLocalDisk="50G">
...

<!-- Cache1 gets 400Mb of heap, 2.5Gb of off-heap, and 5Gb of disk. -->
<cache name="Cache1" ...
maxBytesLocalHeap="40%">
</cache>

<!-- Cache2 gets 300Mb of heap, 5Gb of off-heap, and 5Gb of disk. -->
<cache name="Cache2" ...
maxBytesLocalOffHeap="50%">
</cache>

<!-- Cache2 gets 300Mb of heap, 2.5Gb of off-heap, and 40Gb of disk. -->
<cache name="Cache3" ...
maxBytesLocalDisk="80%">
</cache>
</ehcache>
```

NOTE: Configuring Cache Sizes with Percentages You can use a percentage of the total JVM heap for the CacheManager maxBytesLocalHeap. The CacheManager percentage, then, is a portion of the total JVM heap, and in turn, the Cache percentage is the portion of the CacheManager pool for that tier.

Sizing Caches Without a Pool

The CacheManager in this example does not pool any resources.

```
<ehcache xmlns...
        Name="CM3"
       ... >
...

<cache name="Cache7" ...
        maxBytesLocalHeap="50M"
        maxEntriesLocalDisk="100000"
        ...
  </cache>

<cache name="Cache8" ...
       maxEntriesLocalHeap="1000"
       maxBytesLocalOffHeap="10G"
       ...
  </cache>
<cache name="Cache9" ...
```

```
       maxBytesLocalHeap="50M"
...
</cache>

</ehcache>
```

Caches can be configured to use resources as necessary. Note that every cache in this example must declare a value for local heap. This is because no pool exists for the local heap; implicit (CacheManager configuration) or explicit (cache configuration) local-heap allocation is required.

Overflows

Caches that do not specify overflow will overflow if a pool is set for off-heap and disk.

```
<ehcache maxBytesLocalHeap="1g" maxBytesLocalOffHeap="4g"
       maxBytesLocalDisk="100g" >

<cache name="explicitlyAllocatedCache1"
       maxBytesLocalHeap="50m"
       maxBytesLocalOffHeap="200m"
       timeToLiveSeconds="100">
</cache>

<!-- Does not overflow to disk because overflow has been explicitly
disabled. -->
<cache name="explicitlyAllocatedCache2"
       maxLocalHeap="10%"
       maxBytesLocalOffHeap="200m"
       timeToLiveSeconds="100"
       overflowToDisk="false">
</cache>

<!-- Overflows automatically to off-heap and disk because no specific override and resources are
<cache name="automaticallyAllocatedCache1"
       timeToLiveSeconds="100">
</cache>

<!-- Does not use off-heap because overflow has been explicitly
disabled. -->
<cache name="automaticallyAllocatedCache2"
       timeToLiveSeconds="100"
       overflowToOffHeap="false">
</cache>
</ehcache>
```

Sizing Distributed Caches

Terracotta distributed caches can be sized as noted above, except that they do not use the local disk and therefore cannot be configured with *LocalDisk sizing attributes. Distributed caches use the storage resources (BigMemory and disk) available on the Terracotta Server Array.

Cache-configuration sizing attributes behave as local configuration, which means that every node can load its own sizing attributes for the same caches. That is, while some elements and attributes are fixed by the first Ehcache configuration loaded in the cluster, cache-configuration sizing attributes can vary across nodes for the same cache.

For example, a cache may have the following configuration on one node:

```
<cache name="myCache"
     maxEntriesOnHeap="10000"
     maxBytesLocalOffHeap="8g"
     eternal="false"
     timeToIdleSeconds="3600"
     timeToLiveSeconds="1800"
     overflowToDisk="false">
    <terracotta/>
</cache>
```

The same cache may have the following size configuration on another node:

```
<cache name="myCache"
     maxEntriesOnHeap="10000"
     maxBytesLocalOffHeap="10g"
     eternal="false"
     timeToIdleSeconds="3600"
     timeToLiveSeconds="1800"
     overflowToDisk="false">
    <terracotta/>
</cache>
```

If the cache exceeds its size constraints on a node, then with this configuration the Terracotta Server Array provides myCache with an unlimited amount of disk space for spillover and backup. To impose a limit, you must set `maxElementsOnDisk` to a positive non-zero value:

```
<cache name="myCache"
     maxEntriesOnHeap="10000"
     maxBytesLocalOffHeap="10g"
     eternal="false"
     timeToIdleSeconds="3600"
     timeToLiveSeconds="1800"
     overflowToDisk="false"
     maxElementsOnDisk="1000000">
    <terracotta/>
</cache>
```

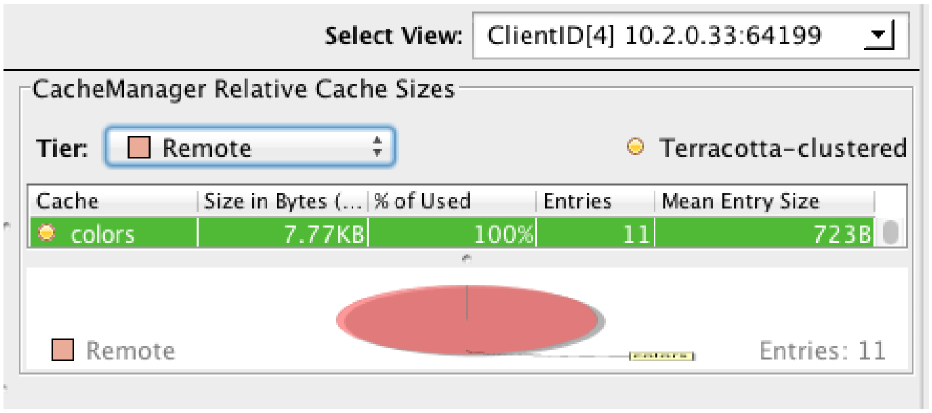
The Terracotta Server Array will now evict myCache entries to stay within the limit set by `maxElementsOnDisk`. However, for any particular cache, eviction on the Terracotta Server Array is based on the largest size configured for that cache. In addition, the Terracotta Server Array will *not* evict any cache entries that exist on at least one client node, regardless of the limit imposed by `maxElementsOnDisk`.

Sizing the Terracotta Server Array

Since `maxElementsOnDisk` is based on entries, you must size the Terracotta Server Array based on the expected average size of an entry. One way to discover this value is by using the average cache-entry size reported by the Terracotta Developer Console.

To find the average cache-entry size in the Terracotta Developer Console:

1. Set up a test cluster with the expected data set.
2. Connect the console to the cluster
3. Go to the Ehcache **Sizing** panel.
4. Choose **Remote** from the **CacheManager Relative Cache Sizes** subpanel's **Tier** drop-down menu.


Select View: ClientID[4] 10.2.0.33:64199

CacheManager Relative Cache Sizes

Tier: Remote   Terracotta-clustered

Cache	Size in Bytes (...	% of Used	Entries	Mean Entry Size
colors	7.77KB	100%	11	723B

Note that the average cache-entry size reported for the Terracotta Server Array (the *remote*) is an estimate.

Overriding Size Limitations

Pinned caches can override the limits set by cache-configuration sizing attributes, potentially causing OutOfMemory errors. This is because pinning prevents flushing of cache entries to lower tiers. For more information on pinning, see Pinning, Eviction, and Expiration.

Built-In Sizing Computation and Enforcement

Internal Ehcache mechanisms track data-element sizes and enforce the limits set by CacheManager sizing pools.

Sizing of cached entries

Elements put in a memory-limited cache will have their memory sizes measured. The entire Element instance added to the cache is measured, including key and value, as well as the memory footprint of adding that instance to internal data structures. Key and value are measured as object graphs – each reference is followed and the object reference also measured. This goes on recursively.

Shared references will be measured by each class that references it. This will result in an overstatement. Shared references should therefore be ignored.

Ignoring for Size Calculations

For the purposes of measurement, references can be ignored using the `@IgnoreSizeOf` annotation. The annotation may be declared at the class level, on a field, or on a package. You can also specify a file containing the fully qualified names of classes, fields, and packages to be ignored.

This annotation is not inherited, and must be added to any subclasses that should also be excluded from sizing.

The following example shows how to ignore the `Dog` class.

```
@IgnoreSizeOf
public class Dog {
  private Gender gender;
  private String name;
}
```

The following example shows how to ignore the `sharedInstance` field.

```
public class MyCacheEntry {
  @IgnoreSizeOf
  private final SharedClass sharedInstance;
    ...
}
```

Packages may be also ignored if you add the @IgnoreSizeOf annotation to appropriate package-info.java of the desired package. Here is a sample package-info.java for and in the com.pany.ignore package:

```
@IgnoreSizeOf
package com.pany.ignore;
import net.sf.ehcache.pool.sizeof.filter.IgnoreSizeOf;
```

Alternatively, you may declare ignored classes and fields in a file and specify a `net.sf.ehcache.sizeof.filter` system property to point to that file.

```
# That field references a common graph between all cached entries
com.pany.domain.cache.MyCacheEntry.sharedInstance

# This will ignore all instances of that type
com.pany.domain.SharedState

# This ignores a package
com.pany.example
```

Note that these measurements and configurations apply only to on-heap storage. Once Elements are moved to off-heap memory, disk, or the Terracotta Server Array, they are serialized as byte arrays. The serialized size is then used as the basis for measurement.

Configuration for Limiting the Traversed Object Graph

As noted above, sizing caches involves traversing object graphs, a process that can be limited with annotations. This process can also be controlled at both the CacheManager and cache levels.

Note that the following configuration has no effect on distributed caches.

Size-Of Limitation at the CacheManager Level

Control how deep the size-of engine can go when sizing on-heap elements by adding the following element at the CacheManager level:

```
<sizeOfPolicy maxDepth="100" maxDepthExceededBehavior="abort"/>
```

This element has the following attributes

- `maxDepth` – Controls how many linked objects can be visited before the size-of engine takes any action. This attribute is required.
- `maxDepthExceededBehavior` – Specifies what happens when the max depth is exceeded while sizing an object graph:

- "continue" – DEFAULT Forces the size-of engine to log a warning and continue the sizing operation. If this attribute is not specified, "continue" is the behavior used.
- "abort" – Forces the SizeOf engine to abort the sizing, log a warning, and mark the cache as not correctly tracking memory usage. With this setting, `Ehcache.hasAbortedSizeOf()` returns true.

The SizeOf policy can be configured at the cache manager level (directly under `<ehcache>`) and at the cache level (under `<cache>` or `<defaultCache>`). The cache policy always overrides the cache manager one if both are set. This element has no effect on distributed caches.

Size-Of Limitation at the Cache level

Use the `<sizeOfPolicy>` as a subelement in any `<cache>` block to control how deep the size-of engine can go when sizing on-heap elements belonging to the target cache. This cache-level setting overrides the CacheManager size-of setting.

Debugging of Size-Of Related Errors

If warnings or errors appear that seem related to size-of measurement (usually caused by the size-of engine walking the graph), generate more log information on sizing activities:

- Set the `net.sf.ehcache.sizeof.verboseDebugLogging` system property to true.
- Enable debug logs on `net.sf.ehcache.pool.sizeof` in your chosen implementation of SLF4J.

Eviction When Using CacheManager-Level Storage

When a CacheManager-level storage pool is exhausted, a cache is selected on which to perform eviction to recover pool space. The eviction from the selected cache is performed using the cache's configured eviction algorithm (LRU, LFU, etc...). The cache from which eviction is performed is selected using the "minimal eviction cost" algorithm described below:

```
eviction-cost = mean-entry-size * drop-in-hit-rate
```

Eviction cost is defined as the increase in bytes requested from the underlying SOR (the database for example) per unit time used by evicting the requested number of bytes from the cache.

If we model the hit distribution as a simple power-law then:

```
P(hit n'th element) ~ 1/n^{alpha}
```

In the continuous limit this means the total hit rate is proportional to the integral of this distribution function over the elements in the cache. The change in hit rate due to an eviction is then the integral of this distribution function between the initial size and the final size. Assuming that the eviction size is small compared to the overall cache size we can model this as:

```
drop ~ access * 1/x^{alpha} * Delta(x)
```

Where 'access' is the overall access rate (hits + misses) and x is a unit-less measure of the 'fill level' of the cache. Approximating the fill level as the ratio of hit rate to access rate, and substituting in to the eviction-cost expression we get:

```
eviction-cost = mean-entry-size * access * 1/(hits/access)^{alpha} * (eviction / (byteSize / (hi
```

Simplifying:

```
eviction-cost = (byteSize / countSize) * access * 1/(h/A)^{alpha} * (eviction * hits)/(access * 
eviction-cost = (eviction * hits) / (countSize * (hits/access)^{alpha})
```

Removing the common factor of 'eviction' which is the same in all caches lead us to evicting from the cache with the minimum value of:

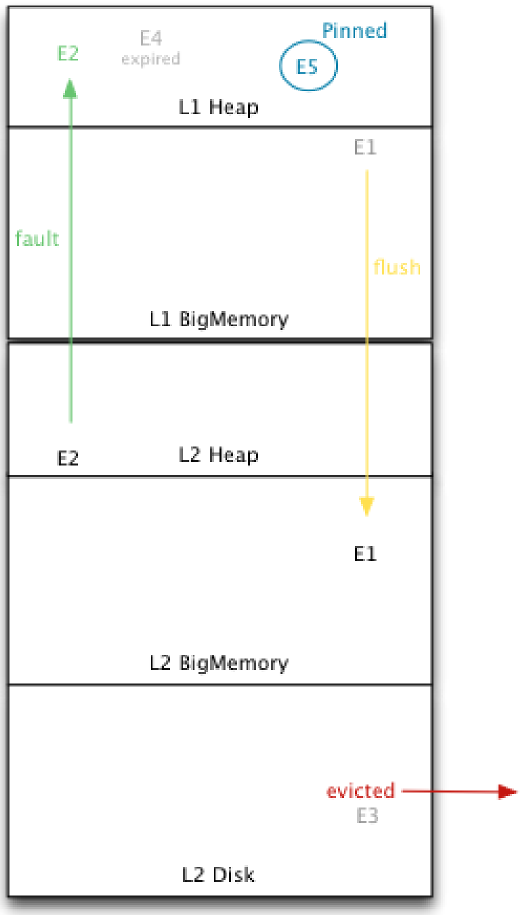
```
eviction-cost = (hits / countSize) / (hits/access)^{alpha}
```

When a cache has a zero hit-rate (it is in a pure loading phase) we deviate from this algorithm and allow the cache to occupy 1/n'th of the pool space where 'n' is the number of caches using the pool. Once the cache starts to be accessed we re-adjust to match the actual usage pattern of that cache.

Pinning, Expiration, and Eviction

Introduction

The architecture of an Ehcache node can include a number of tiers that store data. One of the most important aspects of managing cached data involves managing the life of that data in those tiers.



Distributed Ehcache Client (L1)
With Terracotta Server Array (L2)

Use the figure at right with the definitions below to understand the life of data in the tier of Ehcache nodes backed by the Terracotta Cluster Array.

- Flush – To move a cache entry to a lower tier. Flushing is used to free up resources while still keeping data in the cluster. Entry E1 is shown to be flushed from the L1 off-heap store to the Terracotta Server Array.
- Fault – To copy a cache entry from a lower tier to a higher tier. Faulting occurs when data is required at a higher tier but is not resident there. The entry is not deleted from the lower tiers after being faulted. Entry E2 is shown to be faulted from the Terracotta Server Array to L1 heap.
- Eviction – To remove a cache entry from the cluster. The entry is deleted; it can only be reloaded from a source outside the cluster. Entries are evicted to free up resources. Entry E3, which exists only on the L2 disk, is shown to be evicted from the cluster.

- Expiration – A status based on Time To Live and Time To Idle settings. To maintain cache performance, expired entries may not be immediately flushed or evicted. Entry E4 is shown to be expired but still in the L1 heap.
- Pinning – To force data to remain in certain tiers. Pinning can be set on individual entries or an entire cache, and must be used with caution to avoid exhausting a resource such as heap. E5 is shown pinned to L1 heap.

These definitions apply similarly in standalone Ehcache.

The sections below explore in more detail the aspects of managing data life in Ehcache and the Terracotta Server Array, including the pinning features of Automatic Resource Control (ARC).

Setting Expiration

Cache entries expire based on parameters with configurable values. When eviction occurs, expired elements are the first to be removed. Having an effective expiration configuration is critical to optimizing use of resources such as heap and maintaining cache performance.

To add expiration, edit the values of the following `<cache>` attributes and tune these values based on results of performance tests:

- `timeToIdleSeconds` – The maximum number of seconds an element can exist in the cache without being accessed. The element expires at this limit and will no longer be returned from the cache. The default value is 0, which means no TTI eviction takes place (infinite lifetime).
- `timeToLiveSeconds` – The maximum number of seconds an element can exist in the cache regardless of use. The element expires at this limit and will no longer be returned from the cache. The default value is 0, which means no TTL eviction takes place (infinite lifetime).
- `maxElementsOnDisk` – The maximum sum total number of elements (cache entries) allowed for a distributed cache in all Terracotta clients. If this target is exceeded, eviction occurs to bring the count within the allowed target. The default value is 0, which means no eviction takes place (infinite size is allowed). Note that this value reflects storage allocated on the Terracotta Server Array. **A setting of 0 means that no eviction of the cache's entries takes place on Terracotta Server Array, and consequently can cause the servers to run out of disk space.**
- `eternal` – If the cacheâ s `eternal` flag is set, it overrides any finite TTI/TTL values that have been set.

See How Configuration Affects Element Eviction for more information on how configuration can impact eviction. See Distributed Cache Configuration for definitions of other available configuration properties.

Pinning Data

Data that should remain in the cache regardless of resource constraints can be pinned. You can pin individual entries, or an entire cache.

Pinning Individual Cache Entries

Some APIs like OpenJPA and Hibernate require pinning of specific Elements. Specific entries can be programmatically pinned to the containing cache:

```
cache.setPinned(key, true);
```

The entry can be unpinned by the same method:

```
cache.setPinned(key, false);
```

To unpin all of a cache's pinned entries:

```
cache.unpinAll();
```

To check if an entry is pinned:

```
cache.isPinned(key); // Returns a boolean: true if the key is pinned.
```

Pinning a cache entry guarantees its storage in local memory (heap or off-heap).

Note that pinning is a status applied to a cache entry's key. The entry's value may be null and any operations on value have no effect on the pinning status.

Pinning a Cache

Entire caches can be pinned using the `pinning` element in cache configuration. This element has a required attribute (`store`) to specify which data tiers the cache should be pinned to:

```
<pinning store="localMemory" />
```

The `store` attribute can have one of the following values:

- localHeap – Cache data is pinned to the local heap. Unexpired entries can never be flushed to a lower tier or be evicted.
- localMemory – Cache data is pinned to the local heap or local off-heap. Unexpired entries can never be flushed to a lower tier or be evicted.
- inCache – Cache data is pinned in the cache, which can be in any tier cache data is stored. The tier is chosen based on performance-enhancing efficiency algorithms. This option cannot be used with distributed caches that have a nonzero `maxElementsOnDisk` setting.

For example, the following cache is configured to pin its entries:

```
<cache name="Cache1" ... >
     <pinning store="inCache" />
</cache>
```

The following distributed cache is configured to pin its entries to heap or off-heap only:

```
<cache name="Cache2" ... >
     <pinning store="localMemory" />
    <terracotta/>
</cache>
```

If only a limited set of a cache's entries should be pinned, it may be more efficient to pin those individual elements rather than the entire cache.

Scope of Pinning

Pinning setting is in the local Ehcache client (L1) memory. It is never distributed in the cluster.

Pinning achieved programmatically will not be persisted — after a restart the pinned entries are no longer pinned. Pinning is also lost when an L1 rejoins a cluster. Cache pinning in configuration is reinstated with the configuration file.

How Configuration Affects Element Flushing and Eviction

The following example shows a cache with certain expiration settings:

```
<cache name="myCache"
      maxElementsOnDisk="10000" eternal="false" timeToIdleSeconds="3600"
      timeToLiveSeconds="0" memoryStoreEvictionPolicy="LFU">
 <!-- Adding the element <terracotta /> turns on Terracotta clustering for the cache myCache. -->
 <terracotta />
</cache>
```

Note the following about the myCache configuration:

- If a client accesses an entry in myCache that has been idle for more than an hour (`timeToIdleSeconds`), that element is evicted. The entry is also evicted from the Terracotta Server Array.
- If an entry expires but is not accessed, and no resource constraints force eviction, then the expired entry remains in place.
- Entries in myCache can live forever if accessed at least once per 60 minutes (`timeToLiveSeconds`). However, unexpired entries may still be flushed based on other limitations (see How to Size Caches).
- Cluster-wide, myCache can store a maximum of 10000 entries (`maxElementsOnDisk`). This is the effective maximum number of entries myCache is allowed on the Terracotta Server Array. Note, however, that this value may be exceeded as it is overridden by pinning and other client-side cache-size settings.

Pinning Overrides Cache Sizing

Pinning takes priority over configured cache sizes. For example, in the following distributed cache the pinning configuration overrides the `maxEntriesOnHeap` setting:

```
<cache name="myCache"
     maxEntriesOnHeap="10000"
     maxBytesLocalOffHeap="8g"
     ... >
    <pinning store="localHeap" />
    <terracotta/>
</cache>
```

While expired cache entries (even ones that have been pinned) can always be flushed and eventually evicted from the cluster, most non-expired elements can as well if resource limitations are reached. However, pinned elements, whether pinned individually or resident in a pinned cache, cannot be evicted if they haven't expired. In addition, if a distributed cache is pinned to a specific data tier, its unexpired elements cannot be flushed from that tier.

Pinning Overrides Cache Sizing

Unexpired pinned entries also cannot be evicted from the Terracotta Server Array. While the `maxElementsOnDisk` setting is intended to limit a cache's size in the cluster, it is overridden by pinning because the Terracotta Server Array cannot evict data that is still resident on any client. Persistence takes priority over enforcing resource limits.
CAUTION: Pinning Could Cause Failures Potentially, pinned caches could grow to an unlimited size. Caches should never be pinned unless they are designed to hold a limited amount of data (such as reference data) or their usage and expiration characteristics are understood well enough to conclude that they cannot cause errors.

Nonstop (Non-Blocking) Cache

Introduction

A nonstop cache allows certain cache operations to proceed on clients that have become disconnected from the cluster or if a cache operation cannot complete by the nonstop timeout value. This is useful in meeting SLA requirements, responding to node failures, building a more robust High Availability cluster, and more.

One way clients go into nonstop mode is when they receive a "cluster offline" event. Note that a nonstop cache can go into nonstop mode even if the node is not disconnected, such as when a cache operation is unable to complete within the timeout allotted by the nonstop configuration.

Nonstop can be used in conjunction with rejoin.

Use cases include:

- Setting timeouts on cache operations.

  For example, say you use the cache rather than a mainframe. The SLA calls for 3 seconds. There is a temporary network interruption which delays the response to a cache request. With the timeout you can return after 3 seconds. The lookup is then done against the mainframe. This could also be useful for write-through, writes to disk, or synchronous writes.
- Automatically responding to cluster topology events to take a pre-configured action.
- Allowing Availability over Consistency within the CAP theorem when a network partition occurs.
- Providing graceful degradation to user applications when Distributed Cache becomes unavailable.

Configuring Nonstop Cache

Nonstop is configured in a `<cache>` block under the `<terracotta>` subelement. In the following example, myCache has nonstop configuration:

```
<cache name="myCache" maxEntriesLocalHeap="10000" eternal="false"
       overflowToDisk="false">
 <terracotta>
   <nonstop immediateTimeout="false" timeoutMillis="30000">
     <timeoutBehavior type="noop" />
   </nonstop>
 </terracotta>
</cache>
```

Nonstop is enabled by default or if `<nonstop>` appears in a cacheâ— s `<terracotta>` block.

Nonstop Timeouts and Behaviors

Nonstop caches can be configured with the following attributes:

- `enabled` – Enables ("true" DEFAULT) or disables ("false") the ability of a cache to execute certain actions after a Terracotta client disconnects. This attribute is optional for enabling nonstop.
- `immediateTimeout` – Enables ("true") or disables ("false" DEFAULT) an immediate timeout response if the Terracotta client detects a network interruption (the node is disconnected from the

cluster). If enabled, this parameter overrides `timeoutMillis`, so that the option set in `timeoutBehavior` is in effect immediately.

- `timeoutMillis` – Specifies the number of milliseconds an application waits for any cache operation to return before timing out. The default value is 30000 (thirty seconds). The behavior after the timeout occurs is determined by `timeoutBehavior`.

`<nonstop>` has one self-closing subelement, <timeoutBehavior>. This subelement determines the response after a timeout occurs (`timeoutMillis` expires or an immediate timeout occurs). The response can be set by the <timeoutBehavior> attribute `type`. This attribute can have one of the values listed in the following table:

ValueBehavior `exception` (DEFAULT) Throw `NonStopCacheException`. See When is NonStopCacheException Thrown? for more information on this exception. `noop` Return null for gets. Ignore all other cache operations. Hibernate users may want to use this option to allow their application to continue with an alternative data source. `localReads` For caches with Terracotta clustering, allow inconsistent reads of cache data. Ignore all other cache operations. For caches without Terracotta clustering, throw an exception.

Tuning Nonstop Timeouts and Behaviors

You can tune the default timeout values and behaviors of nonstop caches to fit your environment.

Network Interruptions

For example, in an environment with regular network interruptions, consider disabling `immediateTimeout` and increasing `timeoutMillis` to prevent timeouts for most of the interruptions.

For a cluster that experiences regular but short network interruptions, and in which caches clustered with Terracotta carry read-mostly data or there is tolerance of potentially stale data, you may want to set `timeoutBehavior` to `localReads`.

Slow Cache Operations

In an environment where cache operations can be slow to return and data is required to always be in sync, increase `timeoutMillis` to prevent frequent timeouts. Set `timeoutBehavior` to `noop` to force the application to get data from another source or `exception` if the application should stop.

For example, a `cache.acquireWriteLockOnKey(key)` operation may exceed the nonstop timeout while waiting for a lock. This would trigger nonstop mode only because the lock couldn't be acquired in time. Using `cache.tryWriteLockOnKey(key, timeout)`, with the method's timeout set to less than the nonstop timeout, avoids this problem.

Bulk Loading

If a nonstop cache is bulk-loaded using the Bulk-Load API, a multiplier is applied to the configured nonstop timeout whenever the method `net.sf.ehcache.Ehcache.setNodeBulkLoadEnabled(boolean)` is used. The default value of the multiplier is 10. You can tune the multiplier using the `bulkOpsTimeoutMultiplyFactor` system property:

```
-DbulkOpsTimeoutMultiplyFactor=10
```

This multiplier also affects the methods `net.sf.ehcache.Ehcache.removeAll()`, `net.sf.ehcache.Ehcache.removeAll(boolean)`, and `net.sf.ehcache.Ehcache.setNodeCoherent(boolean)` (DEPRECATED).

Nonstop Exceptions

Typically, application code may access the cache frequently and at various points. Therefore, with a nonstop cache, where your application could encounter NonStopCacheExceptions is difficult to predict. The following sections provide guidance on when to expect NonStopCacheExceptions and how to handle them.

When is NonStopCacheException Thrown?

NonStopCacheException is usually thrown when it is the configured behavior for a nonstop cache in a client that disconnects from the cluster. In the following example, the exception would be thrown 30 seconds after the disconnection (or the "cluster offline" event is received):

```
<nonstop immediateTimeout="false" timeoutMillis="30000">
<timeoutBehavior type="exception" />
</nonstop>
```

However, under certain circumstances the NonStopCache exception can be thrown even if a nonstop cacheâ— s timeout behavior is *not* set to throw the exception. This can happen when the cache goes into nonstop mode during an attempt to acquire or release a lock. These lock operations are associated with certain lock APIs and special cache types such as Explicit Locking, BlockingCache, SelfPopulatingCache, and UpdatingSelfPopulatingCache.

A NonStopCacheException can also be thrown if the cache must fault in an element to satisfy a `get()` operation. If the Terracotta Server Array cannot respond within the configured nonstop timeout, the exception is thrown.

A related exception, InvalidLockAfterRejoinException, can be thrown during or after client rejoin (see Using Rejoin to Automatically Reconnect Terracotta Clients). This exception occurs when an unlock operation takes place on a lock obtained *before* the rejoin attempt completed.
TIP: Use try-finally Blocks To ensure that locks are released properly, application code using Ehcache lock APIs should encapsulate lock-unlock operations with try-finally blocks:

```
myLock.acquireLock();
try {
  // Do some work.
} finally {
  myLock.unlock();
}
```

Handling Nonstop Exceptions

Your application can handle nonstop exceptions in the same way it handles other exceptions. For nonstop caches, an unhandled-exceptions handler could, for example, refer to a separate thread any cleanup needed to manage the problem effectively.

Another way to handle nonstop exceptions is by using a dedicated Ehcache decorator that manages the exception outside of the application framework. The following is an example of how the decorator might operate:

```
try { cache.put(element); }

catch(NonStopCacheException e) {

  handler.handleException(cache, element, e);
}
```

UnlockedReadsView

Introduction

`UnlockedReadsView` is a decorated cache which provides an eventually consistent view of a strongly consistent cache. Views of data are taken without regard to that data's consistency. Writes are not affected by `UnlockedReadsView`. You can have both the unlocked view and a strongly consistent cache at the same time.

The UnlockedReadsView is placed in the CacheManager under its own name so that it can be separately referenced. The purpose of this is to allow business logic faster access to data. It is akin to the READ_UNCOMMITTED database isolation level. Normally, a read lock must first be obtained to read data backed with Terracotta. If there is an outstanding write lock, the read lock queues up. This is done so that the *happens before* guarantee can be made. However, if the business logic is happy to read stale data even if a write lock has been acquired in preparation for changing it, then much higher speeds can be obtained.

Creating an UnlockedReadsView

Programmatically

```
Cache cache = cacheManager.getCache("existingUndecoratedCache");
UnlockedReadsView unlockedReadsView = new UnlockedReadsView(cache, newName);
cacheManager.addDecoratedCache(unlockedReadsView);  //adds a decorated Ehcache
```

If the UnlockedReadsView has the same name as the cache it is decorating, `CacheManager.replaceCacheWithDecoratedCache(Ehcache ehcache, Ehcache decoratedCache)` should be used, instead of using `CacheManager.addDecoratedCache(Ehcache decoratedCache)` as shown above.

If added to the CacheManager, it can be accessed like following:

```
Ehcache unlockedReadsView = cacheManager.getEhcache(newName);
```

NOTE: Right now, `UnlockedReadsView` only accepts `net.sf.ehcache.Cache` instances in the constructor, meaning it can be used to decorate only `net.sf.ehcache.Cache` instances. One disadvantage is that it cannot be used to decorate other already decorated `net.sf.ehcache.Ehcache` instances like `NonStopCache`.

By Configuration

It can be configured in ehcache.xml using the "cacheDecoratorFactory" element. You can specify a factory to create decorated caches and `net.sf.ehcache.constructs.unlockedreadsview.UnlockedReadsViewDecoratorFactory` is available in the unlockedreadsview module itself.

```
<cache name="sample/DistributedCache3"
      maxEntriesLocalHeap="10"
      eternal="false"
      timeToIdleSeconds="100"
      timeToLiveSeconds="100"
      overflowToDisk="true">
   <cacheDecoratorFactory
      class="net.sf.ehcache.constructs.unlockedreadsview.UnlockedReadsViewDecoratorFactory"
      properties="name=unlockedReadsViewOne" />
</cache>
```

It is mandatory to specify the properties for the UnlockedReadsViewDecoratorFactory with "name" property. That property is used as the name of the UnlockedReadsView that will be created.

Download

File

The file is available for download here.

Maven

The UnlockedReadsView is in the ehcache-unlockedreadsview module in the Maven central repo. Add this snippet to your dependencies:

```
<dependency>
  <groupId>net.sf.ehcache</groupId>
  <artifactId>ehcache-unlockedreadsview</artifactId>
</dependency>
```

FAQ

Why is this a CacheDecorator?

This API is emerging. It is production quality and supported, but it is a new API and may evolve over time. As a decorator in its own module, it can evolve separately from ehcache-core.

Why do I see stale values in certain Ehcache nodes for up to 5 minutes?

UnlockedReadsView uses unlocked reads of the Terracotta Server Array combined with a local TTL which, in versions up to Ehcache 2.4, are hardcoded to 300 seconds (5 minutes). If you are already holding a copy in a local node, you will not see an updated value for 5 minutes. As of Ehcache 2.4.1, you also have the option of simply configuring the whole cache as `consitency="eventual"`, which sends changed data to the node as soon as possible. However the whole cache is eventually consistent - you cannot use that with a strongly consistent cache. We plan to make this TTL configurable in a future release.

Distributed Ehcache Configuration Guide

Introduction

The Ehcache configuration file (`ehcache.xml` by default) contains the configuration for one instance of a CacheManager (the Ehcache class managing a set of defined caches). This configuration file must be in your application's classpath to be found. When using a WAR file, `ehcache.xml` should be copied to `WEB-INF/classes`.

Note the following about `ehcache.xml` in a Terracotta cluster:

- The copy on disk is loaded into memory from the first Terracotta client (also called application server or node) to join the cluster.
- Once loaded, the configuration is persisted in memory by the Terracotta servers in the cluster and survives client restarts.
- In-memory configuration can be edited in the Terracotta Developer Console. Changes take effect immediately but are *not* written to the original on-disk copy of `ehcache.xml`.
- The in-memory cache configuration is removed with server restarts if the servers are in non-persistent mode, which is the default. The original (on-disk) `ehcache.xml` is loaded.
- The in-memory cache configuration survives server restarts if the servers are in persistent mode (default is non-persistent). If you are using the Terracotta servers with persistence of shared data, and you want the cluster to load the original (on-disk) `ehcache.xml`, the servers' database must be wiped by removing the data files from the servers' `server-data` directory. This directory is specified in the Terracotta configuration file in effect (`tc-config.xml` by default). Wiping the database causes *all persisted shared data to be lost.*

A minimal distributed-cache configuration should have the following configured:

- A CacheManager
- A Clustering element in individual cache configurations
- A source for Terracotta client configuration

CacheManager Configuration

CacheManager configuration elements and attributes are fully described in the `ehcache.xml` reference file available in the kit.

Via ehcache.xml

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.

TIP: Naming the CacheManager If you employ multiple Ehcache configuration files, use the `name` attribute in the <ehcache> element to identify specific CacheManagers in the cluster. The Terracotta Developer Console provides a menu listing these names, allowing you to choose the CacheManager you want to view.

- 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. The "autodetect" value detects the presence of Terracotta clustering and registers 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">
  ```

Programmatic Configuration

CacheManagers can be configured programmatically with a fluent API. The example below creates a CacheManager with a Terracotta configuration specified in an URL, and creates a defaultCache and a cache named "example".

```
Configuration configuration = new Configuration()
.terracotta(new TerracottaClientConfiguration().url("localhost:9510"))
.defaultCache(new CacheConfiguration("defaultCache", 100))
.cache(new CacheConfiguration("example", 100)
.timeToIdleSeconds(5)
.timeToLiveSeconds(120)
.terracotta(new TerracottaConfiguration()));
CacheManager manager = new CacheManager(configuration);
```

Terracotta Clustering Configuration Elements

Certain elements in the Ehcache configuration control the clustering of caches with Terracotta.

terracotta

The `<terracotta>` element is an optional sub-element of `<cache>`. It can be set differently for each `<cache>` defined in `ehcache.xml`.

`<terracotta>` has one subelement, `<nonstop>` (see Non-Blocking Disconnected (Nonstop) Cache for more information).

The following `<terracotta>` attribute allows you to control the type of data consistency for the distributed cache:

- consistency – Uses no cache-level locks for better performance ("eventual" DEFAULT) or uses cache-level locks for immediate cache consistency ("strong"). When set to "eventual", allows reads without locks, which means the cache may temporarily return stale data in exchange for substantially improved performance. When set to "strong", guarantees that after any update is completed no local read can return a stale value, but at a potentially high cost to performance. If using strong consistency with BigMemory, a large number of locks may need to be stored in client and server heaps. In this case, be sure to test the cluster with the expected data set to detect situations where OutOfMemory errors are likely to occur.

Once set, this consistency mode cannot be changed except by reconfiguring the cache using a configuration file and reloading the file. *This setting cannot be changed programmatically.* See Understanding Performance and Cache Consistency for more information.

Except for special cases, the following `<terracotta>` attributes should operate with their default values:

- clustered – Enables ("true" DEFAULT) or disables ("false") Terracotta clustering of a specific cache. Clustering is enabled if this attribute is not specified. Disabling clustering also disables the effects of all of the other attributes.
- localCacheEnabled – Enables ("true" DEFAULT) or disables ("false") local caching of distributed cache data, forcing all of that cached data to reside on the Terracotta Server Array. Disabling local caching may improve performance for distributed caches in write-heavy use cases.
- synchronousWrites – Enables ("true") or disables ("false" DEFAULT) synchronous writes from Terracotta clients (application servers) to Terracotta servers. Asynchronous writes (synchronousWrites="false") maximize performance by allowing clients to proceed without waiting for a "transaction received" acknowledgement from the server. This acknowledgement is unnecessary in most use cases. Synchronous writes (synchronousWrites="true") provide extreme data safety at a very high performance cost by requiring that a client receive server acknowledgement of a transaction before that client can proceed. *Enabling synchronous writes has a significant detrimental effect on cluster performance.* If the cacheâ— s consistency mode is eventual (consistency="eventual"), or while it is set to bulk load using the bulk-load API, only asynchronous writes can occur (synchronousWrites="true" is ignored).
- storageStrategy – Sets the strategy for storing the cacheâ— s key set. Use "DCV2" (DEFAULT) to store the cacheâ— s key set on the Terracotta server array. DCV2 can be used only with serializable caches (the valueMode attribute must be set to "serialization"), whether using the standard installation or DSO. DCV2 takes advantage of performance optimization built into the Terracotta libraries. Use "classic" to store all keys on every Terracotta client, but note that the performance optimization techniques built into the Terracotta libraries *will not be in effect*. Identity caches (valueMode="identity") must use the classic mode. For more information on using storageStrategy, see Offloading Large Caches.
- concurrency – Sets the number of segments for the map backing the underlying server store managed by the Terracotta Server Array. If concurrency is not explicitly set (or set to "0"), the system selects an optimized value. See Tuning Concurrency for more information on how to tune this value for DCV2.
- valueMode – Sets the type of cache to `serialization` (DEFAULT, the standard Ehcache "copy" cache) or `identity` (Terracotta object identity). **Identity mode is not available with the standard (express) installation**. **Identity mode can be used only with a Terracotta DSO (custom) installation** (see Standard Versus DSO Installations).

TIP: Comparing Serialization and Identity Modes In serialization mode, getting an element from the cache gets a copy of that element. Changes made to that copy do not affect any other copies of the same element or the value in the cache. Putting the element in the cache overwrites the existing value. This type of cache provides high performance with small, read-only data sets. Large data sets with high traffic, or caches with very large elements, can suffer performance degradation because this type of cache serializes clustered objects. This type of cache cannot guarantee a consistent view of object values in read-write data sets if the *consistency* attribute is disabled. Objects clustered in this mode *must be* serializable. Note that `getKeys()` methods return serialized versions of the keys.

In identity mode, getting an element from the cache gets a reference to that element. Changes made to the referenced element updates the element on every node on which it exists (or a reference to it exists) as well as updating the value in the cache. Putting the element in the cache does not overwrite the existing value. This mode guarantees data consistency. It can be used only with a custom Terracotta Distributed Cache installation. Objects clustered in this mode must be portable and must be locked when accessed. If you require identity mode, you must use DSO (see Terracotta DSO Installation).

- `copyOnRead` – DEPRECATED. Use the copyOnRead `<cache>` attribute. Enables ("true") or disables ("false" DEFAULT) "copy cache" mode. If disabled, cache values are not deserialized on every read. For example, repeated get() calls return a reference to the same object (references are ==). When enabled, cache values are deserialized (copied) on every read and the materialized values are *not* re-used between get() calls; each get() call returns a unique reference. When enabled, allows Ehcache to behave as a component of OSGI, allows a cache to be shared by callers with different classloaders, and prevents local drift if keys/values are mutated locally without being put back into the cache. **Enabling copyOnRead is relevant only for caches with valueMode set to serialization**.
- `coherentReads` – DEPRECATED. This attribute is superseded by the attribute *consistency*. Disallows ("true" DEFAULT) or allows ("false") "dirty" reads in the cluster. If set to "true", reads must be consistent on any node and returned data is guaranteed to be consistent. If set to false, local unlocked reads are allowed and returned data may be stale. Allowing dirty reads may boost the clusterâ— s performance by reducing the overhead associated with locking. Read-only applications, applications where stale data is acceptable, and certain read-mostly applications may be suited to allowing dirty reads.

The following attributes are used with Enterprise Ehcache for Hibernate:

- localKeyCache – Enables ("true") or disables ("false" DEFAULT) a local key cache. Enterprise Ehcache for Hibernate can cache a "hotset" of keys on clients to add locality-of-reference, a feature suitable for read-only cases. Note that the set of keys must be small enough for available memory.
- localKeyCacheSize – Defines the size of the local key cache in number (positive integer) of elements. In effect if localKeyCache is enabled. The default value, 300000, should be tuned to meet application requirements and environmental limitations.
- orphanEviction – Enables ("true" DEFAULT) or disables ("false") orphan eviction. *Orphans* are cache elements that are not resident in any Hibernate second-level cache but still present on the cluster's Terracotta server instances.
- orphanEvictionPeriod – The number of local eviction cycles (that occur on Hibernate) that must be completed before an orphan eviction can take place. The default number of cycles is 4. Raise this value for less aggressive orphan eviction that can reduce faulting on the Terracotta server, or raise it if garbage on the Terracotta server is a concern.

Default Behavior

By default, adding `<terracotta/>` to a `<cache>` block is equivalent to adding the following:

```
<cache name="sampleTerracottaCache"
    maxEntriesLocalHeap="1000"
    eternal="false"
    timeToIdleSeconds="3600"
    timeToLiveSeconds="1800"
    overflowToDisk="false">
  <terracotta clustered="true" valueMode="serialization" consistency="eventual" storageStrategy="
</cache>
```

terracottaConfig

The `<terracottaConfig>` element enables the distributed-cache client to identify a source of Terracotta configuration. It also allows a client to rejoin a cluster after disconnecting from that cluster and being timed out by a Terracotta server. For more information on the rejoin feature, see Using Rejoin to Automatically Reconnect Terracotta Clients.

Note that the `<terracottaConfig>` element can *not* be used with a DSO installation (refer to Standard Versus DSO Installations).

The client must load the configuration from a file or a Terracotta server. The value of the `url` attribute should contain a path to the file, a system property, or the address and DSO port (9510 by default) of a server.
TIP: Terracotta Clients and Servers In a Terracotta cluster, the application server is also known as the client.

For more information on client configuration, see the *Clients Configuration Section* in the Terracotta Configuration Reference.

Adding an URL Attribute

Add the `url` attribute to the `<terracottaConfig>` element as follows:

```
<terracottaConfig url="<source>" />
```

where `<source>` must be one of the following:

- A path (for example, `url="/path/to/tc-config.xml"`)
- An URL (for example, `url="http://www.mydomain.com/path/to/tc-config.xml`)
- A system property (for example, `url="${terracotta.config.location}"`), where the system property is defined like this:

```
System.setProperty("terracotta.config.location","10.x.x.x:9510"");
```

- A Terracotta host address in the form `<host>:<dso-port>` (for example, `url="host1:9510"`). Note the following about using server addresses in the form `<host>:<dso-port>`:
    - ♦ The default DSO port is 9510.
    - ♦ In a multi-server cluster, you can specify a comma-delimited list (for example, `url="host1:9510,host2:9510,host3:9510"`).
    - ♦ If the Terracotta configuration source changes at a later time, it must be updated in configuration.

Embedding Terracotta Configuration

You can embed the contents of a Terracotta configuration file in `ehcache.xml` as follows:

```
<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>
```

Note that not all elements are supported. For example, the <dso> section of a Terracotta configuration file is ignored in an Ehcache configuration file.

Controlling Cache Size

Certain Ehcache cache configuration attributes affect caches clustered with Terracotta.

See How Configuration Affects Element Eviction for more information on how configuration affects eviction.

To learn about eviction and controlling the size of the cache, see the Ehcache documentation on data life and sizing caches.

Setting Cache Eviction

Cache eviction removes elements from the cache based on parameters with configurable values. Having an optimal eviction configuration is critical to maintaining cache performance.

To learn about eviction and controlling the size of the cache, see the Ehcache documentation on data life and sizing caches.

Ensure that the edited `ehcache.xml` is in your application's classpath. If you are using a WAR file, `ehcache.xml` should be in `WEB-INF/classes`.

See How Configuration Affects Element Eviction for more information on how configuration can impact eviction. See Terracotta Clustering Configuration Elements for definitions of other available configuration properties.

Cache-Configuration File Properties

See Terracotta Clustering Configuration Elements for more information.

Cache Events Configuration

The `<cache>` subelement <cacheEventListenerFactory>, which registers listeners for cache events such as puts and updates, has a notification scope controlled by the attribute `listenFor`. This attribute can have one of the following values:

- local – Listen for events on the local node. No remote events are detected.
- remote – Listen for events on other nodes. No local events are detected.
- all – (DEFAULT) Listen for events on both the local node and on remote nodes.

In order for cache events to be detected by remote nodes in a Terracotta cluster, event listeners must have a scope that includes remote events. For example, the following configuration allows listeners of type MyCacheListener to detect both local and remote events:

```
<cache name="myCache" ... >
 <!-- Not defining the listenFor attribute for <cacheEventListenerFactory> is by default equivale
 <cacheEventListenerFactory class="net.sf.ehcache.event.TerracottaCacheEventReplicationFactory" /
 <terracotta />
</cache>
```

Cache Events Configuration

You must use `net.sf.ehcache.event.TerracottaCacheEventReplicationFactory` as the factory class to enable cluster-wide cache-event broadcasts in a Terracotta cluster.

See Cache Events in a Terracotta Cluster for more information on cache events in a Terracotta cluster.

Copy On Read

The `copyOnRead` setting is most easily explained by first examining what it does when not enabled and exploring the potential problems that can arise. For a cache in which `copyOnRead` is NOT enabled, the following reference comparison will always be true:

```
Object obj1 = c.get("key").getValue();
// Assume no other thread changes the cache mapping between these get() operations ....
Object obj2 = c.get("key").getValue();
if (obj1 == obj2) {
 System.err.println("Same value objects!");
}
```

The fact that the same object reference is returned across multiple `get()` operations implies that the cache is storing a direct reference to cache value. This default behavior (copyOnRead=false) is usually desired, although there are at least two scenarios in which it is problematic:

(1) Caches shared between classloaders

and

(2) Mutable value objects

Imagine two web applications that both have access to the same Cache instance (this implies that the core ehcache classes are in a common classloader). Imagine further that the classes for value types in the cache are duplicated in the web application (so they are not present in the common loader). In this scenario you would get ClassCastExceptions when one web application accessed a value previously read by the other application.

One obvious solution to this problem is to move the value types to the common loader, but another is to enable `copyOnRead`. When copyOnRead is in effect, the object references are unique with every `get()`. Having unique object references means that the thread context loader of the caller will be used to materialize the cache values on each `get()`. This feature has utility in OSGi environments as well where a common cache service might be shared between bundles.

Another subtle issue concerns mutable value objects in a distributed cache. Consider this simple code with a Cache containing a mutable value type (Foo):

```
class Foo {
 int field;
}
Foo foo = (Foo) c.get("key").getValue();
foo.field++;
// foo instance is never re-put() to the cache
// ...
```

If the Foo instance is never put back into the Cache your local cache is no longer consistent with the cluster (it is locally modified only). Enabling `copyOnRead` eliminates this possibility since the only way to affect cache values is to call mutator methods on the Cache.

It is worth noting that there is a performance penalty to copyOnRead since values are deserialized on every `get()`.

Configuring Robust Distributed Caches

Making caches robust is typically a combination of Ehcache configuration and Terracotta configuration and architecture. For more information, see the following documentation:

- Nonstop caches – Configure caches to take a specified action after an Ehcache node appears to be disconnected from the cluster.
- Rejoin the cluster – Allow Ehcache nodes to rejoin the cluster as new clients after being disconnected from the cluster.
- High Availability in a Terracotta cluster – Configure nodes to ride out network interruptions and long Java GC cycles, connect to a backup Terracotta server, and more.
- Architecture – Design a cluster that provides failover.

Incompatible Configuration

For any clustered cache, you must delete, disable, or edit configuration elements in `ehcache.xml` that are incompatible when clustering with Terracotta. Clustered caches have a `<terracotta/>' or <terracotta clustered="true"> element.

The following Ehcache configuration attributes or elements should be deleted or disabled:

- DiskStore-related attributes `overflowToDisk` and `diskPersistent`. The Terracotta server automatically provides a disk store.
- Replication-related attributes such as `replicateAsynchronously` and `replicatePuts`.
- The attribute `MemoryStoreEvictionPolicy` must be set to either LFU or LRU. Setting `MemoryStoreEvictionPolicy` to FIFO causes the error `IllegalArgumentException`.

Exporting Configuration from the Developer Console

To create or edit a cache configuration in a live cluster, see Editing Cache Configuration.

To persist custom cache configuration values, create a cache configuration file by exporting customized configuration from the Terracotta Developer Console or create a file that conforms to the required format. This file must take the place of any configuration file used when the cluster was last started.

Default Settings for Terracotta Distributed Ehcache

Introduction

A number of properties control the way the Terracotta Server Array and Ehcache clients perform in a Terracotta cluster. Some of these settings are found in the Terracotta configuration file (`tc-config.xml`), while others are found in the Ehcache configuration file (`ehcache.xml`). A few must be set programmatically.

The following sections detail the most important of these properties and shows their default values. To confirm the latest default values for your version of Terracotta software, see the XSD included with your Terracotta kit.

Terracotta Server Array

A Terracotta cluster is composed of clients and servers. Terracotta properties often use a shorthand notation where a client is referred to as "l1" and a server as "l2".

These properties are set at the top of `tc-config.xml` using a configuration block similar to the following:

```
<tc-properties>
   <property name="l2.nha.tcgroupcomm.reconnect.enabled" value="true" />
<!-- More properties here. -->
</tc-properties>
```

See the Terracotta Server Arrays documentation for more information on the Terracotta Server Array.

Reconnection and Logging Properties

The following reconnection properties are shown with default values. These properties can be set to custom values using Terracotta configuration properties (`<tc-properties>`/`<property>` elements in `tc-config.xml`).

PropertyDefault ValueNotes l2.nha.tcgroupcomm.reconnect.enabledtrue

Enables server-to-server reconnections.

l2.nha.tcgroupcomm.reconnect.timeout5000ms

l2-l2 reconnection timeout.

l2.l1reconnect.enabledtrue

Enables an l1 to reconnect to servers.

l2.l1reconnect.timeout.millis5000ms

The reconnection time out, after which an l1 disconnects.

l1.max.connect.retries-1

The number of allowed reconnection tries from an l1 to an l2. Affects both initial and subsequent reconnection attempts. -1 allows infinite attempts.

tc.config.getFromSource.timeout30000ms

Timeout for getting configuration from a source. For example, this controls how long a client can try to access configuration from a server. If the client fails to do so, it will fail to connect to the cluster.

logging.maxBackups20

Upper limit for number of backups of Terracotta log files.

logging.maxLogFileSize512MB

Maximum size of Terracotta log files before rolling logging starts.

HealthChecker Tolerances

The following properties control disconnection tolerances between Terracotta servers (l2 l2), Terracotta servers and Terracotta clients (l2 -> l1), and Terracotta clients and Terracotta servers (l1 -> l2).

l2l2 GC tolerance : 40 secs, cable pull/network down tolerance : 10secs

```
l2.healthcheck.l2.ping.enabled = true
l2.healthcheck.l2.ping.idletime = 5000
l2.healthcheck.l2.ping.interval = 1000
l2.healthcheck.l2.ping.probes = 3
l2.healthcheck.l2.socketConnect = true
l2.healthcheck.l2.socketConnectTimeout = 5
l2.healthcheck.l2.socketConnectCount = 10
```

l2->l1 GC tolerance : 40 secs, cable pull/network down tolerance : 10secs

```
l2.healthcheck.l1.ping.enabled = true
l2.healthcheck.l1.ping.idletime = 5000
l2.healthcheck.l1.ping.interval = 1000
l2.healthcheck.l1.ping.probes = 3
l2.healthcheck.l1.socketConnect = true
l2.healthcheck.l1.socketConnectTimeout = 5
l2.healthcheck.l1.socketConnectCount = 10
```

l1->l1 GC tolerance : 50 secs, cable pull/network down tolerance : 10secs

```
l1.healthcheck.l2.ping.enabled = true
l1.healthcheck.l2.ping.idletime = 5000
l1.healthcheck.l2.ping.interval = 1000
l1.healthcheck.l2.ping.probes = 3
l1.healthcheck.l2.socketConnect = true
l1.healthcheck.l2.socketConnectTimeout = 5
l1.healthcheck.l2.socketConnectCount = 13
```

Ehcache

Ehcache configuration properties typically address the behavior, size, and functionality of caches. Others affect certain types of cache-related bulk operations.

Properties are set in `ehcache.xml` except as noted.

General Cache Settings

The following default cache settings affect cached data. For more information on these settings, see the Ehcache documentation.

PropertyDefault ValueNotesvalue modeSERIALIZATIONconsistencyEVENTUALXAfalseorphan evictiontruelocal key cachefalsesynchronous writefalsememory store eviction policyLRUttl00 means never expire.tti00 means never expire.transactional modeoffdisk persistentfalsemaxElementsOnDisk00 means infinite; this is the cache size on the Terracotta Server Array.maxBytesLocalHeap0maxBytesLocalOffHeap0maxEntriesLocalHeap00 means infinite.

NonStop Cache

The following default settings affect the behavior of the cache when while the client is disconnected from the cluster. For more information on these settings, see the nonstop-cache documentation.

PropertyDefault ValueNotesenablefalsetimeout behaviorexceptiontimeout30000msnet.sf.ehcache.nonstop.bulkOpsTimeoutMultiplyFactor10 This value is a timeout multiplication factor affecting bulk operations such as `removeAll()` and `getAll()`. Since the default nonstop timeout is 30 seconds, it sets a timeout of 300 seconds for those operations. The default can be changed programmatically:

```
cache.getTerracottaConfiguration() .getNonstopConfiguration()
.setBulkOpsTimeoutMultiplyFactor(10)
```

Bulk Operations

The following properties are shown with default values. These properties can be set to custom values using Terracotta configuration properties.

Increasing batch sizes may improve throughput, but could raise latency due to the load on resources from processing larger blocks of data.
PropertyDefault ValueNotesehcache.bulkOps.maxKBSize1MBBatch size for bulk operations such as putAll and removeAll.

ehcache.getAll.batchSize1000The number of elements per batch in a getAll operation.

ehcache.incoherent.putsBatchByteSize5MBFor bulk-loading mode. The minimum size of a batch in a bulk-load operation. Increasing batch sizes may improve throughput, but could raise latency due to the load on resources from processing larger blocks of data.

ehcache.incoherent.putsBatchTimeInMillis600 ms For bulk-loading mode. The maximum time the bulk-load operation takes to batch puts before flushing to the Terracotta Server Array.

BigMemory Overview

BigMemory gives Java applications instant, effortless access to a large memory footprint with **in-memory data management** that lets you store large amounts of data closer to your application, improving memory utilization and application performance with both standalone and distributed caching. BigMemory's in-process, off-heap cache is not subject to Java garbage collection, is 100x faster than DiskStore, and allows you to create very large caches. In fact, the size of the off-heap cache is limited only by address space and the amount of RAM on your hardware. In performance tests, weâ— ve achieved fast, predictable response times with terabyte caches on a single machine.

Rather than stack lots of 1-4 GB JVMs on a single machine in an effort to minimize the GC problem, with BigMemory you can increase application density, running a smaller number of larger-memory JVMs. This simpler deployment model eases application scale out and provides a more sustainable, efficient solution as your dataset inevitably grows.

The following sections provide a documentation Table of Contents and additional information sources for BigMemory.

BigMemory Table of Contents

Topic	Description
BigMemory Configuration	Introduction to BigMemory, how to configure Ehcache with BigMemory, performance comparisons, FAQs, and more.
Further Performance Analysis	Further performance results for off-heap store for a range of scenarios.
Pooling Resources Versus Sizing Individual Caches	Additional information for configuring Ehcache to use local off-heap memory.
Storage Options	Discussion of BigMemory in the context of storage options for Ehcache.
Terracotta Clustering Configuration Elements	The role of BigMemory in data consistency for the distributed cache.

BigMemory Resources

Additional information and downloads:

- About BigMemory
- Tutorial of Ehcache with BigMemory
- Using BigMemory in a Terracotta Server Array

Automatic Resource Control Overview

Automatic Resource Control (ARC) is an intelligent approach to caching with fine-grained controls for tuning cache performance. ARC offers a wealth of benefits, including:

- Sizing limitations on in-memory caches to avoid OutOfMemory errors
- Pooled (CacheManager-level) sizing â—   no requirement to size caches individually
- Differentiated tier-based sizing for flexibility
- Sizing by bytes, entries, or percentages for more flexibility
- Keeping hot or eternal data where it can substantially boost performance

The following sections provide a documentation Table of Contents and additional information sources for ARC.

ARC Table of Contents

Topic	Description
Dynamic Sizing of Memory	Tuning Ehcache often involves sizing cached data appropriately. Ehcache provides a number of ways to size the different data tiers using simple cache-configuration sizing attributes. This page explains simplified tuning of cache size by configuring dynamic allocation of memory and automatic load balancing.
Pinning Caches and Entries	The architecture of an Ehcache node can include a number of tiers that store data. One of the most important aspects of managing cached data involves managing the life of the data in each tier. This page covers managing data life in Ehcache and the Terracotta Server Array, including the pinning features of Automatic Resource Control (ARC).

Additional Information about ARC

The following page provides background information:

- About Automatic Resource Control

APIs Overview

The following sections provide a documentation Table of Contents and additional information sources for the Ehcache APIs.

APIs Table of Contents

Topic	Description
Cache Search	The Ehcache Search API allows you to execute arbitrarily complex queries against either a standalone cache or a Terracotta clustered cache with pre-built indexes. Searchable attributes may be extracted from both keys and values. Keys, values, or summary values (Aggregators) can all be returned.
Bulk Loading	Ehcache has a bulk loading mode that dramatically speeds up bulk loading into caches using the Terracotta Server Array. The bulk-load API should be used for temporarily suspending the Terracotta's normal consistency guarantees to allow for special bulk-load operations, such as cache warming and periodic batch loading.
Transactions (JTA)	Transactional modes are a powerful extension of Ehcache allowing you to perform atomic operations on your caches and potentially other data stores, to keep your cache in sync with your database. This page covers all of the background and configuration information for the transactional modes. Additional information about JTA can be found in the Using Caches section of the Code Samples page.
Explicit Locking	With explicit locking (using Read and Write locks), it is possible to get more control over Ehcache's locking behaviour to allow business logic to apply an atomic change with guaranteed ordering across one or more keys in one or more caches. This API can be used as a custom alternative to XA Transactions or Local transactions.
CacheWriter	Write-through and write-behind are available with the Ehcache CacheWriter API for handling how writes to the cache are subsequently propagated to the SOR. This page covers all of the background and configuration information for the CacheWriter API. An additional discussion about Ehcache Write-Behind may be found in Recipes.
Blocking and Self-populating Caches	With BlockingCache, which can scale up to very busy systems, 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. With SelfPopulatingCache, or pull-through cache, you can specify keys to populate entries. This page introduces these APIs. Additional information may be found in Cache Decorators and in Recipes.
Terracotta Cluster Events	The Terracotta Distributed Ehcache cluster events API provides access to Terracotta cluster events and cluster topology. This event-notification mechanism reports events related to the nodes in the Terracotta cluster, not cache events.
Unlocked Reads View	With this API, you can have both the unlocked view and a strongly consistent cache at the same time. UnlocksReadView provides an eventually consistent view of a strongly consistent cache. Views of data are taken without regard to that data's consistency, and writes are not affected by UnlockedReadsView. This page covers creating an UnlockedReadsView and provides a download link and an FAQ.
Cache Decorators	A cache decorator allows extended functionality to be added to an existing cache dynamically, and can be combined with other decorators on a per-use basis. It is generally required that a decorated cache, once constructed, is made available to other execution threads. The simplest way of doing this is to substitute the original cache for the decorated one.

CacheManager Event Listeners	CacheManager event listeners allow implementers to register callback methods that will be executed when a CacheManager event occurs. The events include adding a cache and removing a cache.
Cache Event Listeners	Cache listeners allow implementers to register callback methods that will be executed when a cache event occurs. The events include Element puts, updates, removes, and expires. Elements can also be put or removed from a cache without notifying listeners. In clustered environments, event propagation can be configured to be propagated only locally, only remotely, or both.
Cache Exception Handlers	A CacheExceptionHandler can be configured to intercept Exceptions and not Errors. Ehcache can be configured with ExceptionHandling so that CacheManager.getEhcache() does not simply return the underlying undecorated cache.
Cache Extensions	CacheExtensions are a general purpose mechanism, tied into the cache lifecycle, which allow generic extensions to a cache. The CacheExtension perform operations such as registering a CacheEventListener or even a CacheManagerEventListener, all from within a CacheExtension, creating more opportunities for customisation.
Cache Eviction Algorithms	A cache eviction algorithm is a way of deciding which element to evict when the cache is full. LRU, LFU, and FIFO are provided, or you can plug in your own algorithm.
Class Loading	Ehcache allows plugins for events and distribution. This page demonstrates how to load and create plug-ins, and it covers loading ehcache.xml resources and classloading with Terracotta clustering.

Additional Information about APIs

The following pages provide additional information about Search, Cluster Events, Bulk-load, and other APIs:

- Enterprise Ehcache API Guide
- CacheManager Code Examples

Ehcache Search API

Introduction

The Ehcache Search API allows you to execute arbitrarily complex queries against either a standalone cache or a Terracotta clustered cache with pre-built indexes. This allows development of alternative indexes on values so that data can be looked up based on multiple criteria instead of just keys.

Searchable attributes may be extracted from both keys and values. Keys, values, or summary values (Aggregators) can all be returned. Here is a simple example: Search for 32-year-old males and return the cache values.

```
Results results = cache.createQuery().includeValues()
  .addCriteria(age.eq(32).and(gender.eq("male"))).execute();
```

What is Searchable?

Searches can be performed against Element keys and values.

Element keys and values are made searchable by extracting attributes with supported search types out of the keys and values. It is the attributes themselves which are searchable.

How to Make a Cache Searchable

By Configuration

Caches are made searchable by adding a `<searchable/>` tag to the ehcache.xml.

```
<cache name="cache2" maxEntriesLocalHeap="10000" eternal="true" overflowToDisk="false">
<searchable/>
</cache>
```

This configuration will scan keys and vales and if they are of supported search types, add them as attributes called "key" and "value" respectively. If you do not want automatic indexing of keys and values you can disable it with:

```
<cache name="cache3" ...>
<searchable keys="false" values="false">
   ...
</searchable>
</cache>
```

You might want to do this if you have a mix of types for your keys or values. The automatic indexing will throw an exception if types are mixed. Often keys or values will not be directly searchable and instead you will need to extract searchable attributes out of them. The following example shows this more typical case. Attribute Extractors are explained in more detail in the following section.

```
<cache name="cache3" maxEntriesLocalHeap="10000" eternal="true" overflowToDisk="false">
<searchable>
   <searchAttribute name="age" class="net.sf.ehcache.search.TestAttributeExtractor"/>
   <searchAttribute name="gender" expression="value.getGender()"/>
```

```
</searchable>
</cache>
```

Programmatically

The following example shows how to programmatically create the cache configuration, with search attributes.

```
Configuration cacheManagerConfig = new Configuration();
CacheConfiguration cacheConfig = new CacheConfiguration("myCache", 0).eternal(true);
Searchable searchable = new Searchable();
cacheConfig.addSearchable(searchable);
// Create attributes to use in queries.
searchable.addSearchAttribute(new SearchAttribute().name("age"));
// Use an expression for accessing values.
searchable.addSearchAttribute(new SearchAttribute()
    .name("first_name")
    .expression("value.getFirstName()"));
searchable.addSearchAttribute(new SearchAttribute().name("last_name").expression("value.getLastNa
     searchable.addSearchAttribute(new SearchAttribute().name("zip_code").expression("value.getZi
cacheManager = new CacheManager(cacheManagerConfig);
cacheManager.addCache(new Cache(cacheConfig));
Ehcache myCache = cacheManager.getEhcache("myCache");
// Now create the attributes and queries, then execute.
...
```

To learn more about the Ehcache Search API, see the `net.sf.ehcache.search*` packages in this Javadoc.

Attribute Extractors

Attributes are extracted from keys or values. This is done during search or, if using Distributed Ehcache, on `put()` into the cache using `AttributeExtractors`. Extracted attributes must be one of the following supported types:

- Boolean
- Byte
- Character
- Double
- Float
- Integer
- Long
- Short
- String
- java.util.Date
- java.sql.Date
- Enum

If an attribute cannot be extracted due to not being found or being the wrong type, an AttributeExtractorException is thrown on search execution or, if using Distributed Ehcache, on `put()`.

Well-known Attributes

The parts of an Element that are well-known attributes can be referenced by some predefined, well-known names. If a keys and/or value is of a supported search type, they are added automatically as attributes with the names "key" amd "value". These well-known attributes have convenience constant attributes made available on the `Query` class. So, for example, the attribute for "key" may be referenced in a query by `Query.KEY`. For even greater readability it is recommended to statically import so that in this example you would just use `KEY`.

Well-known Attribute Name	Attribute Constant
key	Query.KEY
value	Query.VALUE

ReflectionAttributeExtractor

The ReflectionAttributeExtractor is a built-in search attribute extractor which uses JavaBean conventions and also understands a simple form of expression. Where a JavaBean property is available and it is of a searchable type, it can be simply declared using:

```
<cache>
  <searchable>
    <searchAttribute name="age"/>
  </searchable>
</cache>
```

Finally, when things get more complicated, we have an expression language using method/value dotted expression chains. The expression chain must start with one of either "key", "value", or "element". From the starting object a chain of either method calls or field names follows. Method calls and field names can be freely mixed in the chain. Some more examples:

```
<cache>
  <searchable>
    <searchAttribute name="age" expression="value.person.getAge()"/>
  </searchable>
</cache>
<cache>
  <searchable>
     <searchAttribute name="name" expression="element.toString()"/>
  </searchable>
</cache>
```

The method and field name portions of the expression are case sensitive.

Custom AttributeExtractor

In more complex situations you can create your own attribute extractor by implementing the AttributeExtractor interface. Providing your extractor class is shown in the following example:

```
<cache name="cache2" maxEntriesLocalHeap="0" eternal="true" overflowToDisk="false">
  <searchable>
     <searchAttribute name="age" class="net.sf.ehcache.search.TestAttributeExtractor"/>
  </searchable>
</cache>
```

Custom AttributeExtractor

If you need to pass state to your custom extractor you may do so with properties as shown in the following example:

```
<cache>
  <searchable>
    <searchAttribute name="age"
    class="net.sf.ehcache.search.TestAttributeExtractor"
    properties="foo=this,bar=that,etc=12" />
  </searchable>
</cache>
```

If properties are provided, then the attribute extractor implementation must have a public constructor that accepts a single `java.util.Properties` instance.

Query API

Ehcache Search introduces a fluent Object Oriented query API, following DSL principles, which should feel familiar and natural to Java programmers. Here is a simple example:

```
Query query = cache.createQuery().addCriteria(age.eq(35)).includeKeys().end();
Results results = query.execute();
```

Using Attributes in Queries

If declared and available, the well-known attributes are referenced by their names or the convenience attributes are used directly as shown in this example:

```
Results results = cache.createQuery().addCriteria(Query.KEY.eq(35)).execute();
Results results = cache.createQuery().addCriteria(Query.VALUE.lt(10)).execute();
```

Other attributes are referenced by the names given them in the configuration. For example:

```
Attribute<Integer> age = cache.getSearchAttribute("age");
Attribute<String> gender = cache.getSearchAttribute("gender");
Attribute<String> name = cache.getSearchAttribute("name");
```

Expressions

The Query to be searched for is built up using Expressions. Expressions include logical operators such as <and> and <or>. It also includes comparison operators such as <ge> (>=), <between>, and <like>. `addCriteria(...)` is used to add a clause to a query. Adding a further clause automatically <and>s the clauses.

```
query = cache.createQuery().includeKeys().addCriteria(age.le(65)).add(gender.eq("male")).end();
```

Both logical and comparison operators implement the `Criteria` interface. To add a criteria with a different logical operator, explicitly nest it within a new logical operator Criteria Object. For example, to check for age = 35 or gender = female, do the following:

```
query.addCriteria(new Or(age.eq(35),
             gender.eq(Gender.FEMALE))
            );
```

More complex compound expressions can be further created with extra nesting. See the Expression JavaDoc for a complete list.

List of Operators

Operators are available as methods on attributes, so they are used by adding a ".". For example, "lt" means "less than" and is used as `age.lt(10)`, which is a shorthand way of saying `new LessThan(10)`. The full listing of operator shorthand is shown below.

Shorthand	Criteria Class	Description
and	And	The Boolean AND logical operator
between	Between	A comparison operator meaning between two values
eq	EqualTo	A comparison operator meaning Java "equals to" condition
gt	GreaterThan	A comparison operator meaning greater than.
ge	GreaterThanOrEqual	A comparison operator meaning greater than or equal to.
in	InCollection	A comparison operator meaning in the collection given as an argument
lt	LessThan	A comparison operator meaning less than.
le	LessThanOrEqual	A comparison operator meaning less than or equal to
ilike	ILike	A regular expression matcher. '?' and "*" may be used. Note that placing a wildcard in front of the expression will cause a table scan. ILike is always case insensitive.
not	Not	The Boolean NOT logical operator
ne	NotEqualTo	A comparison operator meaning not the Java "equals to" condition
or	Or	The Boolean OR logical operator

Making Queries Immutable

By default, a query can be executed and then modified and re-executed. If `end` is called, the query is made immutable.

Search Results

Queries return a `Results` object which contains a list of objects of class `Result`. Each `Element` in the cache found with a query will be represented as a `Result` object. So if a query finds 350 elements there will be 350 `Result` objects. An exception to this would be if no keys or attributes are included but aggregators are -- in this case, there will be exactly one `Result` present.

A Result object can contain:

- the Element key - when `includeKeys()` is added to the query,
- the Element value - when `includeValues()` is added to the query,
- predefined attribute(s) extracted from an Element value - when `includeAttribute(...)` is added to the query. To access an attribute from Result, use `getAttribute(Attribute<T> attribute`.
- aggregator results

  Aggregator results are summaries computed for the search. They are available through `Result.getAggregatorResults` which returns a list of `Aggregators` in the same order in which they were used in the `Query`.

Aggregators

Aggregators are added with `query.includeAggregator(\<attribute\>.\<aggregator\>).` For example, to find the sum of the age attribute:

```
query.includeAggregator(age.sum());
```

See the Aggregators JavaDoc for a complete list.

Ordering Results

Query results may be ordered in ascending or descending order by adding an `addOrderBy` clause to the query, which takes as parameters the attribute to order by and the ordering direction. For example, to order the results by ages in ascending order

```
query.addOrderBy(age, Direction.ASCENDING);
```

Limiting the Size of Results

Either all results can be returned using `results.all()` to get them all in one chunk, or page results can be returned with ranges using `results.range(int start, int count).`

By default a query will return an unlimited number of results. For example the following query will return all keys in the cache.

```
Query query = cache.createQuery();
query.includeKeys();
query.execute();
```

If too many results are returned, it could cause an OutOfMemoryError The `maxResults` clause is used to limit the size of the results. For example, to limit the above query to the first 100 elements found:

```
Query query = cache.createQuery();
query.includeKeys();
query.maxResults(100);
query.execute();
```

When you are done with the results, call `discard()` to free up resources. In the distributed implementation with Terracotta, resources may be used to hold results for paging or return.

Interrogating Results

To determine what was returned by a query, use one of the interrogation methods on `Results`:

- `hasKeys()`
- `hasValues()`
- `hasAttributes()`
- `hasAggregators()`

Sample Application

We have created a simple standalone sample application with few dependencies for you to easily get started with Ehcache Search. You can also check out the source:

```
git clone git://github.com/sharrissf/Ehcache-Search-Sample.git
```

The Ehcache Test Sources page has further examples on how to use each Ehcache Search feature.

Scripting Environments

Ehcache Search is readily amenable to scripting. The following example shows how to use it with BeanShell:

```
Interpreter i = new Interpreter();
//Auto discover the search attributes and add them to the interpreter's context
Map<String, SearchAttribute> attributes = cache.getCacheConfiguration().getSearchAttributes();
for (Map.Entry<String, SearchAttribute> entry : attributes.entrySet()) {
i.set(entry.getKey(), cache.getSearchAttribute(entry.getKey()));
LOG.info("Setting attribute " + entry.getKey());
}
//Define the query and results. Add things which would be set in the GUI i.e.
//includeKeys and add to context
Query query = cache.createQuery().includeKeys();
Results results = null;
i.set("query", query);
i.set("results", results);
//This comes from the freeform text field
String userDefinedQuery = "age.eq(35)";
//Add the stuff on that we need
String fullQueryString = "results = query.addCriteria(" + userDefinedQuery + ").execute()";
i.eval(fullQueryString);
results = (Results) i.get("results");
assertTrue(2 == results.size());
for (Result result : results.all()) {
LOG.info("" + result.getKey());
}
```

Concurrency Considerations

Unlike cache operations which has selectable concurrency control and/or transactions, the Search API does not. This may change in a future release, however our survey of prospective users showed that concurrency control in search indexes was not sought after. The indexes are eventually consistent with the caches.

Index Updating

Indexes will be updated asynchronously, so their state will lag slightly behind the state of the cache. The only exception is when the updating thread then performs a search.

For caches with concurrency control, an index will not reflect the new state of the cache until:

- The change has been applied to the cluster.
- For a cache with transactions, when `commit` has been called.

Query Results

There are several ways unexpected results could present:

- A search returns an Element reference which no longer exists.
- Search criteria select an Element, but the Element has been updated and a new Search would no longer match the Element.
- Aggregators, such as sum(), might disagree with the same calculation done by redoing the calculation yourself by re-accessing the cache for each key and repeating the calculation.
- `includeValues` returns values. Under the covers the index contains a server value reference. The reference gets returned with the search and Terracotta supplies the matching value. Because the cache is always updated before the search index it is possible that a value reference may refer to a value that has been removed from the cache. If this happens the value will be null but the key and attributes which were supplied by the now stale cache index will be non-null. Because values in Ehcache are also allowed to be null, you cannot tell whether your value is null because it has been removed from the cache since the index was last updated or because it is a null value.

Recommendations

Because the state of the cache can change between search executions it is recommended to add all of the Aggregators you want for a query at once so that the returned aggregators are consistent. Use null guards when accessing a cache with a key returned from a search.

Implementations

Standalone Ehcache

The standalone Ehcache implementation does not use indexes. It uses fast iteration of the cache instead, relying on the very fast access to do the equivalent of a table scan for each query. Each element in the cache is only visited once. Attributes are not extracted ahead of time. They are done during query execution.

Performance

Search operations perform in O(n) time. Checkout this Maven-based performance test showing standalone cache performance. This test shows search performance of an average of representative queries at 10ms per 10,000 entries. So, a typical query would take 1 second for a 1,000,000 entry cache. Accordingly, standalone implementation is suitable for development and testing.

For production it is recommended to only standalone search for caches that are less than 1 million elements. Performance of different `Criteria` vary. For example, here are some queries and their execute times on a 200,000 element cache. (Note that these results are all faster than the times given above because they execute a single Criteria).

```
final Query intQuery = cache.createQuery();
  intQuery.includeKeys();
  intQuery.addCriteria(age.eq(35));
  intQuery.end();
Execute Time: 62ms
final Query stringQuery = cache.createQuery();
  stringQuery.includeKeys();
  stringQuery.addCriteria(state.eq("CA"));
  stringQuery.end();
Execute Time: 125ms
final Query iLikeQuery = cache.createQuery();
  iLikeQuery.includeKeys();
  iLikeQuery.addCriteria(name.ilike("H*"));
  iLikeQuery.end();
Execute Time: 180ms
```

Ehcache Backed by the Terracotta Server Array

This implementation uses indexes which are maintained on each Terracotta server. In Ehcache EX the index is on a single active server. In Ehcache FX the cache is sharded across the number of active nodes in the cluster. The index for each shard is maintained on that shard's server. Searches are performed using the Scatter-Gather pattern. The query executes on each node and the results are then aggregated back in the Ehcache that initiated the search.

Performance

Search operations perform in O(log n / number of shards) time. Performance is excellent and can be improved simply by adding more servers to the FX array.

Network Effects

Search results are returned over the network. The data returned could potentially be very large, so techniques to limit return size are recommended such as:

- Limiting the results with `maxResults` or using the paging API `Results.range(int start, int length).`
- Only including the data you need. Specifically only use `includeKeys()` and/or `includeAttribute()` if those values are actually required for your application logic.
- Using a built-in `Aggregator` function when you only need a summary statistic. `includeValues` rates a special mention. Once a query requiring values is executed we push the values from the server to the Ehcache CacheManager which requested it in batches for network efficiency. This is done ahead as soon as possible reducing the risk that `Result.getValue()` might have to wait for data over the network.
- Turning off key and value indexing if you are not going to search against them as they will just chew up space on the server.

  You do this as follows:

  ```
  <cache name="cache3" ...>
    <searchable keys="false" values="false">
    ...
    </searchable>
  </cache>
  ```

Bulk Loading in Ehcache

Introduction

Ehcache has a bulk loading mode that dramatically speeds up bulk loading into caches using the Terracotta Server Array. Bulk loading is designed to be used for:

- cache warming - where caches need to be filled before bringing an application online
- periodic batch loading - say an overnight batch process that uploads data

API

With bulk loading, the API for putting data into the cache stays the same. Just use `cache.put(...)` `cache.load(...)` or `cache.loadAll(...)`. What changes is that there is a special mode that suspends Terracotta's normal consistency guarantees and provides optimised flushing to the Terracotta Server Array (the L2 cache).
NOTE: The Bulk-Load API and the Configured Consistency Mode The initial consistency mode of a cache is set by configuration and cannot be changed programmatically (see the `<terracotta>` element's `consistency` attribute). The bulk-load API should be used for temporarily suspending the configured consistency mode to allow for bulk-load operations.

The following are the bulk-load API methods that are available in `org.terracotta.modules.ehcache.Cache`.

- `public boolean isClusterBulkLoadEnabled()`

  Returns true if a cache is in bulk-load mode (is not consistent) throughout the cluster. Returns false if the cache is not in bulk-load mode (is consistent) anywhere in the cluster.
- `public boolean isNodeBulkLoadEnabled()`

  Returns true if a cache is in bulk-load mode (is not consistent) on the current node. Returns false if the cache is not in bulk-load mode (is consistent) on the current node.
- `public void setNodeBulkLoadEnabled(boolean)`

  Sets a cacheâ— s consistency mode to the configured mode (false) or to bulk load (true) on the local node. There is no operation if the cache is already in the mode specified by `setNodeBulkLoadEnabled()`. When using this method on a nonstop cache , a multiple of the nonstop cacheâ— s timeout value applies. The bulk-load operation must complete within that timeout multiple to prevent the configured nonstop behavior from taking effect. For more information on tuning nonstop timeouts, see Tuning Nonstop Timeouts and Behaviors.
- `public void waitUntilBulkLoadComplete()`

  Waits until a cache is consistent before returning. Changes are automatically batched and the cache is updated throughout the cluster. Returns immediately if a cache is consistent throughout the cluster.

Note the following about using bulk-load mode:

- Consistency cannot be guaranteed because `isClusterBulkLoadEnabled()` can return false in one node just before another node calls `setNodeBulkLoadEnabled(true)` on the same cache.

Understanding exactly how your application uses the bulk-load API is crucial to effectively managing the integrity of cached data.
- If a cache is not consistent, any ObjectNotFound exceptions that may occur are logged.
- `get()` methods that fail with ObjectNotFound return null.
- Eviction is independent of consistency mode. Any configured or manually executed eviction proceeds unaffected by a cacheâ— s consistency mode.

The following example code shows how a clustered application with Enterprise Ehcache can use the bulk-load API to optimize a bulk-load operation:

```
import net.sf.ehcache.Cache;
public class MyBulkLoader {
CacheManager cacheManager = new CacheManager();  // Assumes local ehcache.xml.
  Cache cache = cacheManager.getEhcache(\"myCache\"); // myCache defined in ehcache.xml.
  cache.setNodeBulkLoadEnabled(true); // myCache is now in bulk mode.
// Load data into myCache.
cache.setNodeBulkLoadEnabled(false); // Done, now set myCache back to its configured consistency
}
```

NOTE: Potentional Error With Non-Singleton CacheManagerEhcache 2.5 and higher does not allow multiple CacheManagers with the same name to exist in the same JVM. `CacheManager()` constructors creating non-Singleton CacheManagers can violate this rule, causing an error. If your code may create multiple CacheManagers of the same name in the same JVM, avoid this error by using the static `CacheManager.create()` methods, which always return the named (or default unnamed) CacheManager if it already exists in that JVM. If the named (or default unnamed) CacheManager does not exist, the `CacheManager.create()` methods create it.

On another node, application code that intends to touch myCache can run or wait, based on whether myCache is consistent or not:

```
...
if (!cache.isClusterBulkLoadEnabled()) {
// Do some work.
}
else {
 cache.waitUntilBulkLoadComplete()
 // Do the work when waitUntilBulkLoadComplete() returns.
}
...
```

Waiting may not be necessary if the code can handle potentially stale data:

```
...
if (!cache.isClusterBulkLoadEnabled()) {
// Do some work.
}
else {
// Do some work knowing that data in myCache may be stale.
}
...
```

The following methods have been deprecated: `setNodeCoherent(boolean mode)`, `isNodeCoherent()`, `isClusterCoherent()` and `waitUntilClusterCoherent()`.

Speed Improvement

The speed performance improvement is an order of magnitude faster. ehcacheperf (Spring Pet Clinic) now has a bulk load test which shows the performance improvement for using a Terracotta cluster.

FAQ

Are there any alternatives to putting the cache into bulk-load mode?

Bulk-loading Cache methods putAll(), getAll(), and removeAll() provide high-performance and eventual consistency. These can also be used with strong consistency. If you can use them, it's unnecessary to use bulk-load mode. See the API documentation for details.

Why does the bulk loading mode only apply to Terracotta clusters?

Ehcache, both standalone and replicated is already very fast and nothing needed to be added.

How does bulk load with RMI distributed caching work?

The core updates are very fast. RMI updates are batched by default once per second, so bulk loading will be efficiently replicated.

Performance Tips

When to use Multiple Put Threads

It is not necessary to create multiple threads when calling `cache.put`. Only a marginal performance improvement will result, because the call is already so fast. It is only necessary if the source is slow. By reading from the source in multiple threads a speed up could result. An example is a database, where multiple reading threads will often be better.

Bulk Loading on Multiple Nodes

The implementation scales very well when the load is split up against multiple Ehcache CacheManagers on multiple machines. You add extra nodes for bulk loading to get up to 93 times performance.

Why not run in bulk load mode all the time

Terracotta clustering provides consistency, scaling and durability. Some applications will require consistency, or not for some caches, such as reference data. It is possible to run a cache permanently in inconsistent mode.

Download

The bulk loading feature is in the ehcache-core module but only provides a performance improvement to Terracotta clusters (as bulk loading to Ehcache standalone is very fast already) Download here. For a full distribution enabling connection to the Terracotta Server array download here.

Further Information

Saravanan who was the lead on this feature has blogged about it here.

Transactions in Ehcache

Introduction

Transactions are supported in versions of Ehcache 2.0 and higher. The 2.3.x or lower releases only support XA. However since ehcache 2.4 support for both Global Transactions with `xa_strict` and `xa` modes, and Local Transactions with `local` mode has been added.

All or nothing

If a cache is enabled for transactions, all operations on it must happen within a transaction context otherwise a `TransactionException` will be thrown.

Change Visibility

The isolation level offered in Ehcache is `READ_COMMITTED`. Ehcache can work as an XAResource in which case, full two-phase commit is supported. Specifically:

- All mutating changes to the cache are transactional including `put`, `remove`, `putWithWriter`, `removeWithWriter` and `removeAll`.
- Mutating changes are not visible to other transactions in the local JVM or across the cluster until `COMMIT` has been called.
- Until then, read such as by `cache.get(...)` by other transactions will return the old copy. Reads do not block.

When to use transactional modes

Transactional modes are a powerful extension of Ehcache allowing you to perform atomic operations on your caches and potentially other data stores, eg: to keep your cache in sync with your database.

- `local` When you want your changes across multiple caches to be performed atomically. Use this mode when you need to update your caches atomically, ie: have all your changes be committed or rolled back using a straight simple API. This mode is most useful when a cache contains data calculated out of other cached data.
- `xa` Use this mode when you cache data from other data stores (eg: DBMS, JMS) and want to do it in an atomic way under the control of the JTA API but don't want to pay the price of full two-phase commit. In this mode, your cached data can get out of sync with the other resources participating in the transactions in case of a crash so only use it if you can afford to live with stale data for a brief period of time.
- `xa_strict` Same as xa but use it only if you need strict XA disaster recovery guarantees. In this mode, the cached data can never get out of sync with the other resources participating in the transactions, even in case of a crash but you pay a high price in performance to get that extra safety. This is the only mode that can work with nonstop caches (beginning with Ehcache 2.4.1).

Requirements

The objects you are going to store in your transactional cache must:

- implement `java.io.Serializable` This is required to store cached objects when the cache is clustered with Terracotta but it's also required by the copy on read / copy on write mechanism used to implement isolation.
- override `equals` and `hashcode` Those must be overridden as the transactional stores rely on element value comparison, see: `ElementValueComparator` and the `elementValueComparator` configuration setting.

Configuration

Transactions are enabled on a cache by cache basis with the `transactionalMode` cache attribute. The allowed values are:

- `xa_strict`
- `xa`
- `local`
- `off`

The default value is off. Enabling a cache for `xa_strict` transactions is shown in the following example:

```
<cache name="xaCache"
   maxEntriesLocalHeap="500"
   eternal="false"
   timeToIdleSeconds="300"
   timeToLiveSeconds="600"
   overflowToDisk="false"
   diskPersistent="false"
   diskExpiryThreadIntervalSeconds="1"
   transactionalMode="xa_strict">
 </cache>
```

Transactional Caches with Terracotta Clustering

For Terracotta clustered caches, `transactionalMode` can only be used where `terracotta consistency="strong"`. Because caches can be dynamically changed to bulk-load mode, any attempt to perform a transaction when this is the case will throw a `CacheException`. Note that transactions do not work with Terracotta's `identity` mode. An attempt to initialise a transactional cache when this mode is set will result in a `CacheException` being thrown. The default mode is `serialization` mode. Also note that all transactional modes are currently sensitive to the ABA problem.

Transactional Caches with Spring

Note the following when using Spring:

- If you access the cache from an @Transactional Spring-annotated method, then begin/commit/rollback statements are not required in application code as they are emitted by Spring.
- Both Spring and Ehcache need to access the transaction manager internally, and therefore you must inject your chosen transaction manager into Spring's PlatformTransactionManager as well as use an appropriate lookup strategy for Ehcache.
- The Ehcache default lookup strategy may not be able to detect your chosen transaction manager. For example, it cannot detect the WebSphere transaction manager (see Transactions Managers).
- Configuring a `<tx:method>` with read-only=true could be problematic with certain transaction

managers such as WebSphere.

Global Transactions

Global Transactions are supported by Ehcache. Ehcache can act as an {XAResouce} to participate in JTA ("Java Transaction API") transactions under the control of a Transaction Manager. This is typically provided by your application server, however you may also use a third party transaction manager such as Bitronix. To use Global Transactions, select either `xa_strict` or `xa` mode. The differences are explained in the sections below.

Implementation

Global transactions support is implemented at the Store level, through XATransactionStore and JtaLocalTransactionStore. The former actually decorates the underlying MemoryStore implementation, augmenting it with transaction isolation and two-phase commit support through an <XAResouce> implementation. The latter decorates a LocalTransactionStore-decorated cache to make it controllable by the standard JTA API instead of the proprietary TransactionController API. During its initialization, the Cache will lookup the TransactionManager using the provided TransactionManagerLookup implementation. Then, using the `TransactionManagerLookup.register(XAResouce)`, the newly created XAResource will be registered. The store is automatically configured to copy every Element read from the cache or written to it. Cache is copy-on-read and copy-on-write.

Failure Recovery

As specified by the JTA specification, only <prepared> transaction data is recoverable. Prepared data is persisted onto the cluster and locks on the memory are held. This basically means that non-clustered caches cannot persist transactions data, so recovery errors after a crash may be reported by the transaction manager.

Recovery

At any time after something went wrong, an `XAResource` may be asked to recover. Data that has been prepared may either be committed or rolled back during recovery. In accordance with XA, data that has not yet been `prepared` is discarded. The recovery guarantee differs depending on the xa mode.

xa Mode

With `xa`, the cache doesn't get registered as an {XAResource} with the transaction manager but merely can follow the flow of a JTA transaction by registering a JTA {Synchronization}. The cache can end up inconsistent with the other resources if there is a JVM crash in the mutating node. In this mode, some inconsistency may occur between a cache and other XA resources (such as databases) after a crash. However, the cache's data remains consistent because the transaction is still fully atomic on the cache itself.

xa_strict Mode

If `xa_strict` is used the cache will always respond to the TransactionManager's recover calls with the list of prepared XIDs of failed transactions. Those transaction branches can then be committed or rolled back by the transaction manager. This is the standard XA mechanism in strict compliance with the JTA specification.

Sample Apps

We have three sample applications showing how to use XA with a variety of technologies.

XA Sample App

This sample application uses JBoss application server. It shows an example using User managed transactions. While we expect most people will use JTA from within Spring or EJB where the container rather than managing it themselves, it clearly shows what is going on. The following snippet from our SimpleTX servlet shows a complete transaction.

```
Ehcache cache = cacheManager.getEhcache("xaCache");
UserTransaction ut = getUserTransaction();
printLine(servletResponse, "Hello...");
try {
   ut.begin();
   int index = serviceWithinTx(servletResponse, cache);
   printLine(servletResponse, "Bye #" + index);
   ut.commit();
} catch(Exception e) {
   printLine(servletResponse,
       "Caught a " + e.getClass() + "! Rolling Tx back");
   if(!printStackTrace) {
       PrintWriter s = servletResponse.getWriter();
       e.printStackTrace(s);
       s.flush();
   }
   rollbackTransaction(ut);
}
```

The source code for the demo can be checked out from the Terracotta Forge. A README.txt explains how to get the JTA Sample app going.

XA Banking Application

The Idea of this application is to show a real world scenario. A Web app reads <account transfer> messages from a queue and tries to execute these account transfers. With JTA turned on, failures are rolled back so that the cached account balance is always the same as the true balance summed from the database. This app is a Spring-based Java web app running in a Jetty container. It has (embedded) the following components:

- A JMS Server (ActiveMQ)
- 2 databases (embedded Derby XA instances)
- 2 caches (JTA Ehcache)

All XA Resources are managed by Atomikos TransactionManager. Transaction demarcation is done using Spring AOP's `@Transactional` annotation. You can run it with: `mvn clean jetty:run`. Then point your browser at: http://localhost:9080. To see what happens without XA transactions: `mvn clean jetty:run -Dxa=no`

The source code for the demo can be checked out from the Terracotta Forge. A README.txt explains how to get the JTA Sample app going.

Examinator

Examinator is our complete application that shows many aspects of caching in one web based Exam application, all using the Terracotta Server Array. Check out from the Terracotta Forge.

Transaction Managers

Automatically Detected Transaction Managers

Ehcache automatically detects and uses the following transaction managers in the following order:

- GenericJNDI (e.g. Glassfish, JBoss, JTOM and any others that register themselves in JNDI at the standard location of java:/TransactionManager
- Weblogic (since 2.4.0)
- Bitronix
- Atomikos

No configuration is required; they work out of the box. The first found is used.

Configuring a Transaction Manager

If your Transaction Manager is not in the above list or you wish to change the priority, provide your own lookup class based on an implementation of `net.sf.ehcache.transaction.manager.TransactionManagerLookup` and specify it in place of the `DefaultTransactionManagerLookup` in `ehcache.xml`:

```
<transactionManagerLookup
  class= "com.mycompany.transaction.manager.MyTransactionManagerLookupClass"
  properties="" propertySeparator=":"/>
```

Another option is to provide a different location for the JNDI lookup by passing the jndiName property to the DefaultTransactionManagerLookup. The example below provides the proper location for the TransactionManager in GlassFish v3:

```
<transactionManagerLookup
  class="net.sf.ehcache.transaction.manager.DefaultTransactionManagerLookup"
  properties="jndiName=java:appserver/TransactionManager" propertySeparator=";"/>
```

Local Transactions

Local Transactions allow single phase commit across multiple cache operations, across one or more caches, and in the same CacheManager, whether distributed with Terracotta or standalone. This lets you apply multiple changes to a CacheManager all in your own transaction. If you also want to apply changes to other resources such as a database then you need to open a transaction to them and manually handle commit and rollback to ensure consistency. Local transactions are not controlled by a Transaction Manager. Instead there is an explicit API where a reference is obtained to a `TransactionController` for the CacheManager using `cacheManager.getTransactionController()` and the steps in the transaction are called explicitly. The steps in a local transaction are:

- `transactionController.begin()` - This marks the beginning of the local transaction on the current thread. The changes are not visible to other threads or to other transactions.
- `transactionController.commit()` - Commits work done in the current transaction on the calling thread.
- `transactionController.rollback()` - Rolls back work done in the current transaction on the calling thread. The changes done since begin are not applied to the cache. These steps should be placed in a try-catch block which catches `TransactionException`. If any exceptions are thrown, rollback() should be called. Local Transactions has it's own exceptions that can be thrown, which are all subclasses of `CacheException`. They are:
- `TransactionException` - a general exception
- `TransactionInterruptedException` - if Thread.interrupt() got called while the cache was processing a transaction.
- `TransactionTimeoutException` - if a cache operation or commit is called after the transaction timeout has elapsed.

Introduction Video

Ludovic Orban the primary author of Local Transactions presents an introductory video on Local Transactions.

Configuration

Local transactions are configured as follows:

```
<cache name="sampleCache"
   ...
   transactionalMode="local"
 </cache>
```

Isolation Level

As with the other transaction modes, the isolation level is READ_COMMITTED.

Transaction Timeouts

If a transaction cannot complete within the timeout period, then a `TransactionTimeoutException` will be thrown. To return the cache to a consistent state you need to call `transactionController.rollback()`. Because `TransactionController` is at the level of the CacheManager, a default timeout can be set which applies to all transactions across all caches in a CacheManager. If not set, it is 15 seconds. To change the defaultTimeout:

```
transactionController.setDefaultTransactionTimeout(int defaultTransactionTimeoutSeconds)
```

The countdown starts straight after `begin()` is called. You might have another local transaction on a JDBC connection and you may be making multiple changes. If you think it could take longer than 15 seconds for an individual transaction, you can override the default when you begin the transaction with:

```
transactionController.begin(int transactionTimeoutSeconds) {
```

Sample Code

This example shows a transaction which performs multiple operations across two caches.

```
CacheManager cacheManager = CacheManager.getInstance();
try {
   cacheManager.getTransactionController().begin();
    cache1.put(new Element(1, "one"));
   cache2.put(new Element(2, "two"));
   cache1.remove(4);
    cacheManager.getTransactionController().commit();
} catch (CacheException e) {
   cacheManager.getTransactionController().rollback()
}
```

What can go wrong

JVM crash between begin and commit

On restart none of the changes applied after begin are there. On restart, nothing needs to be done. Under the covers in the case of a Terracotta cluster, the Element's new value is there but not applied. It's will be lazily removed on next access.

Performance

Managing Contention

If two transactions, either standalone or across the cluster, attempt to perform a cache operation on the same element then the following rules apply:

- The first transaction gets access
- The following transactions will block on the cache operation until either the first transaction completes or the transaction timeout occurs.

Under the covers, when an element is involved in a transaction, it is replaced with a new element with a marker that is locked, along with the transaction ID. The normal cluster semantics are used. Because transactions only work with consistency=strong caches, the first transaction will be the thread that manages to atomically place a soft lock on the Element. (Up to Terracotta 3.4 this was done with write locks. After that it is done with the CAS based putIfAbsent and replace methods).

What granularity of locking is used?

Ehcache standalone up to 2.3 used page level locking, where each segment in the `CompoundStore` is locked. From 2.4, it is one with soft locks stored in the Element itself and is on a key basis. Terracotta clustered caches are locked on the key level.

Performance Comparisons

Any transactional cache adds an overhead which is significant for writes and nearly negligible for reads. Within the modes the relative time take to perform writes, where off = 1, is:

- off - no overhead
- xa_strict - 20 times slower
- xa - 3 times slower
- local - 3 times slower The relative read performance is:
- off - no overhead
- xa_strict - 20 times slower
- xa - 30% slower
- local - 30% slower

Accordingly, xa_strict should only be used where it's full guarantees are required, othewise one of the other modes should be used.

FAQ

Why do some threads regularly time out and throw an excption?

In transactional caches, write locks are in force whenever an element is updated, deleted, or added. With concurrent access, these locks cause some threads to block and appear to deadlock. Eventually the deadlocked threads time out (and throw an exception) to avoid being stuck in a deadlock condition.

Is IBM Websphere Transaction Manager supported?

Mostly. xa_strict is not supported due to each version of Websphere essentially being a custom implementation i.e. no stable interface to implement against. However, `xa`, which uses TransactionManager callbacks and `local` are supported.

When using Spring, make sure your configuration is set up correctly with respect to the PlatformTransactionManager and the Websphere TM.

To confirm that Ehcache will succeed, try to manually register a `com.ibm.websphere.jtaextensions.SynchronizationCallback` in the `com.ibm.websphere.jtaextensions.ExtendedJTATransaction`. Simply get `java:comp/websphere/ExtendedJTATransaction` from JNDI, cast that to `com.ibm.websphere.jtaextensions.ExtendedJTATransaction` and call the `registerSynchronizationCallbackForCurrentTran` method. If you succeed, then Ehcache should too.

How do transactions interact with Write-behind and Write-through caches?

If your transactional enabled cache is being used with a writer, write operations will be queued until transaction commit time. Solely a Write-through approach would have its potential XAResource participate in the same transaction. Write-behind, while supported, should probably not be used with an XA transactional Cache, as the operations would never be part of the same transaction. Your writer would also be responsible for obtaining a new transaction... Using Write-through with a non XA resource would also work, but there is no guarantee the transaction will succeed after the write operations have been executed successfully. On the other hand, any thrown exception during these write operations would cause the transaction to be rolled back by having UserTransaction.commit() throw a RollbackException.

Are Hibernate Transactions supported?

Ehcache is a "transactional" cache for Hibernate purposes. The `net.sf.ehcache.hibernate.EhCacheRegionFactory` has support for Hibernate entities configured with <cache usage="transactional"/>.

How do I make WebLogic 10 work with Ehcache JTA?

WebLogic uses an optimization that is not supported by our implementation. By default WebLogic 10 will spawn threads to start the Transaction on each XAResource in parallel. As we need transaction work to be performed on the same Thread, you will have to turn this optimization off by setting `parallel-xa-enabled` option to `false` in your domain configuration :

... 300 false 30 ...

How do I make Atomikos work with Ehcache JTA's `xa` mode?

Atomikos has a bug which makes the `xa` mode's normal transaction termination mechanism unreliable, There is an alternative termination mechanism built in that transaction mode that is automatically enabled when `net.sf.ehcache.transaction.xa.alternativeTerminationMode` is set to true or when Atomikos is detected as the controlling transaction manager. This alternative termination mode has strict requirement on the way threads are used by the transaction manager and Atomikos's default settings won't work. You have to configure this property to make it work:

```
com.atomikos.icatch.threaded_2pc=false
```

Explicit Locking

Introduction

This package contains an implementation of an Ehcache which provides for explicit locking, using Read and Write locks. It is possible to get more control over Ehcache's locking behaviour to allow business logic to apply an atomic change with guaranteed ordering across one or more keys in one or more caches. It can therefore be used as a custom alternative to `XA` Transactions or `Local` transactions.

With that power comes a caution. It is possible to create deadlocks in your own business logic using this API.

Note that prior to Ehcache 2.4, this API was implemented as a CacheDecorator and was available in the `ehcache-explicitlocking` module. It is now built into the `core` module.

The API

The following methods are available on `Cache` and `Ehcache`.

```
/**
* Acquires the proper read lock for a given cache key
*
* @param key - The key that retrieves a value that you want to protect via locking
*/
public void acquireReadLockOnKey(Object key) {
   this.acquireLockOnKey(key, LockType.READ);
}
/**
* Acquires the proper write lock for a given cache key
*
* @param key - The key that retrieves a value that you want to protect via locking
*/
public void acquireWriteLockOnKey(Object key) {
   this.acquireLockOnKey(key, LockType.WRITE);
}
/**
* Try to get a read lock on a given key. If can't get it in timeout millis then
* return a boolean telling that it didn't get the lock
*
* @param key - The key that retrieves a value that you want to protect via locking
* @param timeout - millis until giveup on getting the lock
* @return whether the lock was awarded
* @throws InterruptedException
*/
public boolean tryReadLockOnKey(Object key, long timeout) throws InterruptedException {
   Sync s = getLockForKey(key);
   return s.tryLock(LockType.READ, timeout);
}
/**
* Try to get a write lock on a given key. If can't get it in timeout millis then
* return a boolean telling that it didn't get the lock
*
* @param key - The key that retrieves a value that you want to protect via locking
* @param timeout - millis until giveup on getting the lock
* @return whether the lock was awarded
* @throws InterruptedException
*/
```

```
public boolean tryWriteLockOnKey(Object key, long timeout) throws InterruptedException {
   Sync s = getLockForKey(key);
   return s.tryLock(LockType.WRITE, timeout);
}
/**
* Release a held read lock for the passed in key
*
* @param key - The key that retrieves a value that you want to protect via locking
*/
public void releaseReadLockOnKey(Object key) {
   releaseLockOnKey(key, LockType.READ);
}
/**
* Release a held write lock for the passed in key
*
* @param key - The key that retrieves a value that you want to protect via locking
*/
public void releaseWriteLockOnKey(Object key) {
   releaseLockOnKey(key, LockType.WRITE);
}
/**
* Returns true if a read lock for the key is held by the current thread
*
* @param key
* @return true if a read lock for the key is held by the current thread
*/
boolean isReadLockedByCurrentThread(Object key);
/**
* Returns true if a write lock for the key is held by the current thread
*
* Only Terracotta clustered cache instances currently support querying a thread's read lock hold
*
* @param key
* @return true if a write lock for the key is held by the current thread
*/
boolean isWriteLockedByCurrentThread(Object key);
```

Example

Here is a brief example:

```
String key = "123";
Foo val = new Foo();
cache.acquireWriteLockOnKey(key);
try {
    cache.put(new Element(key, val));
} finally {
    cache.releaseWriteLockOnKey(key);
}
...sometime later
String key = "123";
cache.acquireWriteLockOnKey(key);
try {
    Object cachedVal = cache.get(key).getValue();
    cachedVal.setSomething("abc");
    cache.put(new Element(key, cachedVal));
 } finally {
cache.releaseWriteLockOnKey(key);
 }
```

Supported Topologies

Except as noted in the Javadoc (see above), explicit locking is supported in Ehcache standalone and also in Distributed Ehcache. It is not supported in Replicated Ehcache.

How it works

A READ lock does not prevent other READers from also acquiring a READ lock and reading. A READ lock cannot be obtained if there is an outstanding WRITE lock - it will queue. A WRITE lock cannot be obtained while there are outstanding READ locks - it will queue. In each case the lock should be released after use to avoid locking problems. The lock release should be in a `finally` block. If before each read you acquire a READ lock and then before each write you acquire a WRITE lock, then an isolation level akin to READ_COMMITTED is achieved.

Write-through and Write-behind Caching with the CacheWriter

Introduction

Write-through caching is a caching pattern where writes to the cache cause writes to an underlying resource. The cache acts as a facade to the underlying resource. With this pattern, it often makes sense to read through the cache too. Write-behind caching uses the same client API; however, the write happens asynchronously. Ehcache-2.0 introduced write-through and write-behind caching. While file systems or a web-service clients can underlie the facade of a write-through cache, the most common underlying resource is a database. To simplify the discussion, we will use the database as the example resource.

Potential Benefits of Write-Behind

The major benefit of write-behind is database offload. This can be achieved in a number of ways:

- time shifting - moving writes to a specific time or time interval. For example, writes could be batched up and written overnight, or at 5 minutes past the hour, to avoid periods of peak contention.
- rate limiting - spreading writes out to flatten peaks. Say a Point of Sale network has an end-of-day procedure where data gets written up to a central server. All POS nodes in the same time zone will write all at once. A very large peak will occur. Using rate limiting, writes could be limited to 100 TPS, and the queue of writes are whittled down over several hours
- conflation - consolidate writes to create fewer transactions. For example, a value in a database row is updated by 5 writes, incrementing it from 10 to 20 to 31 to 40 to 45. Using conflation, the 5 transactions are replaced by one to update the value from 10 to 45.

These benefits must be weighed against the limitations and constraints imposed.

Limitations & Constraints of Write-Behind

Transaction Boundaries

If the cache participates in a JTA transaction (ehcache-2.0 and higher), which means it is an XAResource, then the cache can be made consistent with the database. A write to the database, and a commit or rollback, happens with the transaction boundary. In write-behind, the write to the resource happens after the write to the cache. The transaction boundary is the write to the outstanding queue, not the write behind. In write-through mode, commit can get called and both the cache and the underlying resource can get committed at once. Because the database is being written to outside of the transaction, there is always a risk that a failure on the eventual write will occur. While this can be mitigated with retry counts and delays, compensating actions may be required.

Time delay

The obvious implication of asynchronous writes is that there is a delay between when the cache is updated and when the database is updated. This introduces an inconsistency between the cache and the database, where the cache holds the correct value and the database will be eventually consistent with the cache. The data passed into the CacheWriter methods is a snapshot of the cache entry at the time of the write to operation. A read against the database will result in incorrect data being loaded.

Applications Tolerant of Inconsistency

The application must be tolerant of inconsistent data. The following examples illustrate this requirement:

- The database is logging transactions and only appends are done.
- Reading is done by a part of the application that does not write, so there is no way that data can be corrupted. The application is tolerant of delays. For example, a news application where the reader displays the articles that are written.

Note if other applications are writing to the database, then a cache can often be inconsistent with the database.

Node time synchronisation

Ideally node times should be synchronised. The write-behind queue is generally written to the underlying resource in timestamp order, based on the timestamp of the cache operation, although there is no guaranteed ordering. The ordering will be more consistent if all nodes are using the same time. This can easily be achieved by configuring your system clock to synchronise with a time authority using Network Time Protocol.

No ordering guarantees

The items on the write-behind queue are generally in order, but this isn't guaranteed. In certain situations and more particularly in clustered usage, the items can be processed out of order. Additionally, when batching is used, write and delete collections are aggregated separately and can be processed inside the CacheWriter in a different order than the order that was used by the queue. Your application must be tolerant of item reordering or you need to compensate for this in your implementation of the CacheWriter. Possible examples are:

- Working with versioning in the cache elements.

  You may have to explicitly version elements. Auto-versioning is off by default and is effective only for unclustered MemoryStore caches. Distributed caches or caches that use off-heap or disk stores cannot use auto-versioning. To enable auto-versioning, set the system property `net.sf.ehcache.element.version.auto` (it is false by default). Note that if this property is turned on for one of the ineligible caches, auto-versioning will silently fail.
- Verifications with the underlying resource to check if the scheduled write-behind operation is still relevant.

Using a combined Read-Through and Write-Behind Cache

For applications that are not tolerant of inconsistency, the simplest solution is for the application to always read through the same cache that it writes through. Provided all database writes are through the cache, consistency is guaranteed. And in the distributed caching scenario, using Terracotta clustering extends the same guarantee to the cluster. If using transactions, the cache is the XAResource, and a commit is a commit to the cache. The cache effectively becomes the System Of Record ("SOR"). Terracotta clustering provides HA and durability and can easily act as the SOR. The database then becomes a backup to the SOR. The following aspects of read-through with write-behind should be considered:

Lazy Loading

The entire data set does not need to be loaded into the cache on startup. A read-through cache uses a `CacheLoader` that loads data into the cache on demand. In this way the cache can be populated lazily.

Caching of a Partial Dataset

If the entire dataset cannot fit in the cache, then some reads will miss the cache and fall through to the `CacheLoader` which will in turn hit the database. If a write has occurred but has not yet hit the database due to write-behind, then the database will be inconsistent. The simplest solution is to ensure that the entire dataset is in the cache. This then places some implications on cache configuration in the areas of expiry and eviction.

Eviction

Eviction or flushing of elements, occurs when the maximum elements for the cache have been exceeded. Be sure to size the cache appropriately to avoid eviction or flushing. See How to Size Caches for more information.

Expiry

Even if all of the dataset can fit in the cache, it could be evicted if Elements expire. Accordingly, both `timeToLive` and `timeToIdle` should be set to eternal ("0") to prevent this from happening.

Introductory Video

Alex Snaps the primary author of Write Behind presents an introductory video on Write Behind.

Sample Application

We have created a sample web application for a raffle which fully demonstrates how to use write behind. You can also checkout the Ehcache Raffle application, that demonstrates Cache Writers and Cache Loaders from github.com.

Ehcache Versions

Both Ehcache standalone (DX) and with Terracotta Server Array (Ehcache EX and FX) are supported.

Ehcache DX (Standalone Ehcache)

The write-behind queue is stored locally in memory. It supports all configuration options, but any data in the queue will be lost on JVM shutdown.

Ehcache EX and FX

Durable HA write-behind Queue

EX and FX when used with the Terracotta Server Array will store the queue on the Terracotta Server Array and can thus be configured for durability and HA. The data is still kept in the originating node for performance.

Configuration

There are many configuration options. See the `CacheWriterConfiguration` for properties that may be set and their effect. Below is an example of how to configure the cache writer in XML:

```
<cache name="writeThroughCache1" ... >
<cacheWriter writeMode="write_behind" maxWriteDelay="8" rateLimitPerSecond="5"
        writeCoalescing="true" writeBatching="true" writeBatchSize="20"
        retryAttempts="2" retryAttemptDelaySeconds="2">
   <cacheWriterFactory class="com.company.MyCacheWriterFactory"
                   properties="just.some.property=test; another.property=test2" propertySeparator
</cacheWriter>
</cache>
```

Further examples:

```
<cache name="writeThroughCache2" ... >
 <cacheWriter/>
</cache>
<cache name="writeThroughCache3" ... >
 <cacheWriter writeMode="write_through" notifyListenersOnException="true" maxWriteDelay="30"
      rateLimitPerSecond="10" writeCoalescing="true" writeBatching="true" writeBatchSize="8"
      retryAttempts="20" retryAttemptDelaySeconds="60"/>
</cache>
<cache name="writeThroughCache4" ... >
 <cacheWriter writeMode="write_through" notifyListenersOnException="false" maxWriteDelay="0"
      rateLimitPerSecond="0" writeCoalescing="false" writeBatching="false" writeBatchSize="1"
      retryAttempts="0" retryAttemptDelaySeconds="0">
 <cacheWriterFactory class="net.sf.ehcache.writer.WriteThroughTestCacheWriterFactory"/>
 </cacheWriter>
</cache>
<cache name="writeBehindCache5" ... >
 <cacheWriter writeMode="write-behind" notifyListenersOnException="true" maxWriteDelay="8" rateLi
      writeCoalescing="true" writeBatching="false" writeBatchSize="20"
      retryAttempts="2" retryAttemptDelaySeconds="2">
 <cacheWriterFactory class="net.sf.ehcache.writer.WriteThroughTestCacheWriterFactory"
                 properties="just.some.property=test; another.property=test2" propertySeparator="
 </cacheWriter>
</cache>
```

This configuration can also be achieved through the `Cache` constructor in Java:

```
Cache cache = new Cache(
new CacheConfiguration("cacheName", 10)
.cacheWriter(new CacheWriterConfiguration()
.writeMode(CacheWriterConfiguration.WriteMode.WRITE_BEHIND)
.maxWriteDelay(8)
.rateLimitPerSecond(5)
.writeCoalescing(true)
.writeBatching(true)
.writeBatchSize(20)
.retryAttempts(2)
.retryAttemptDelaySeconds(2)
.cacheWriterFactory(new CacheWriterConfiguration.CacheWriterFactoryConfiguration()
   .className("com.company.MyCacheWriterFactory")
   .properties("just.some.property=test; another.property=test2")
   .propertySeparator(";"))));
```

Configuration

Instead of relying on a `CacheWriterFactoryConfiguration` to create a `CacheWriter`, it's also possible to explicitly register a `CacheWriter` instance from within Java code. This allows you to refer to local resources like database connections or file handles.

```
Cache cache = manager.getCache("cacheName");
MyCacheWriter writer = new MyCacheWriter(jdbcConnection);
cache.registerCacheWriter(writer);
```

Configuration Attributes

The CacheWriterFactory supports the following attributes:

All modes

- write-mode [write-through | write-behind] - Whether to run in write-behind or write-through mode. The default is write-through.

write-through mode only

- notifyListenersOnException - Whether to notify listeners when an exception occurs on a store operation. Defaults to false. If using cache replication, set this attribute to "true" to ensure that changes to the underlying store are replicated.

write-behind mode only

- writeBehindMaxQueueSize - The maximum number of elements allowed per queue, or per bucket (if the queue has multiple buckets). "0" means unbounded (default). When an attempt to add an element is made, the queue size (or bucket size) is checked, and if full then the operation is blocked until the size drops by one. Note that elements or a batch currently being processed (and coalesced elements) are not included in the size value. Programmatically, this attribute can be set with `net.sf.ehcache.config.CacheWriterConfiguration.setWriteBehindMaxQueueSize(`
- writeBehindConcurrency - The number of thread-bucket pairs on the node for the given cache (default is 1). Each thread uses the settings configured for write-behind. For example, if rateLimitPerSecond is set to 100, each thread-bucket pair will perform up to 100 operations per second. In this case, setting writeBehindConcurrency="4" means that up to 400 operations per second will occur on the node for the given cache. Programmatically, this attribute can be set with `net.sf.ehcache.config.CacheWriterConfiguration.setWriteBehindConcurrency()`
- maxWriteDelaySeconds - The maximum number of seconds to wait before writing behind. Defaults to 0. If set to a value greater than 0, it permits operations to build up in the queue to enable effective coalescing and batching optimisations.
- rateLimitPerSecond - The maximum number of store operations to allow per second.
- writeCoalescing - Whether to use write coalescing. Defaults to false. When set to true, if multiple operations on the same key are present in the write-behind queue, then only the latest write is done (the others are redundant). This can dramatically reduce load on the underlying resource.
- writeBatching - Whether to batch write operations. Defaults to false. If set to true, storeAll and deleteAll will be called rather than store and delete being called for each key. Resources such as databases can perform more efficiently if updates are batched to reduce load.
- writeBatchSize - The number of operations to include in each batch. Defaults to 1. If there are less entries in the write-behind queue than the batch size, the queue length size is used. Note that batching is split across operations. For example, if the batch size is 10 and there were 5 puts and 5 deletes, the CacheWriter is invoked. It does not wait for 10 puts or 10 deletes.

- retryAttempts - The number of times to attempt writing from the queue. Defaults to 1.
- retryAttemptDelaySeconds - The number of seconds to wait before retrying.

API

CacheLoaders are exposed for API use through the `cache.getWithLoader(...)` method. CacheWriters are exposed with `cache.putWithWriter(...)` and `cache.removeWithWriter(...)` methods. For example, following is the method signature for `cache.putWithWriter(...)`.

```
/**
* Put an element in the cache writing through a CacheWriter. If no CacheWriter has been
* set for the cache, then this method has the same effect as cache.put().
*
* Resets the access statistics on the element, which would be the case if it has previously
* been gotten from a cache, and is now being put back.
*
* Also notifies the CacheEventListener, if the writer operation succeeds, that:
*
* - the element was put, but only if the Element was actually put.
* - if the element exists in the cache, that an update has occurred, even if the element
* would be expired if it was requested
*
*
* @param element An object. If Serializable it can fully participate in replication and the
* DiskStore.
* @throws IllegalStateException    if the cache is not {@link net.sf.ehcache.Status#STATUS_ALIVE}
* @throws IllegalArgumentException if the element is null
* @throws CacheException
*/
void putWithWriter(Element element) throws IllegalArgumentException, IllegalStateException,
CacheException;
```

See the Cache JavaDoc for the complete API.

SPI

The Ehcache write-through SPI is the `CacheWriter` interface. Implementers perform writes to the underlying resource in their implementation.

```
/**
* A CacheWriter is an interface used for write-through and write-behind caching to a
* underlying resource.
* <p/>
* If configured for a cache, CacheWriter's methods will be called on a cache operation.
* A cache put will cause a CacheWriter write
* and a cache remove will cause a writer delete.
* <p>
* Implementers should create an implementation which handles storing and deleting to an
* underlying resource.
* </p>
* <h4>Write-Through</h4>
* In write-through mode, the cache operation will occur and the writer operation will occur
* before CacheEventListeners are notified. If
* the write operation fails an exception will be thrown. This can result in a cache which
* is inconsistent with the underlying resource.
* To avoid this, the cache and the underlying resource should be configured to participate
* in a transaction. In the event of a failure
* a rollback can return all components to a consistent state.
* <p/>
* <h4>Write-Behind</h4>
* In write-behind mode, writes are written to a write-behind queue. They are written by a
* separate execution thread in a configurable
* way. When used with Terracotta Server Array, the queue is highly available. In addition
* any node in the cluster may perform the
* write-behind operations.
* <p/>
* <h4>Creation and Configuration</h4>
* CacheWriters can be created using the CacheWriterFactory.
* <p/>
* The manner upon which a CacheWriter is actually called is determined by the
* {@link net.sf.ehcache.config.CacheWriterConfiguration} that is set up for a cache
* using the CacheWriter.
* <p/>
* See the CacheWriter chapter in the documentation for more information on how to use writers.
*
* @author Greg Luck
* @author Geert Bevin
* @version $Id: $
*/
public interface CacheWriter {
/**
* Creates a clone of this writer. 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 CacheWriter clone(Ehcache cache) throws CloneNotSupportedException;
 /**
* Notifies writer 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.
*/
void dispose() throws CacheException;
/**
* Write the specified value under the specified key to the underlying store.
* This method is intended to support both key/value creation and value update for a
* specific key.
*
* @param element the element to be written
*/
void write(Element element) throws CacheException;
/**
* Write the specified Elements to the underlying store. This method is intended to
* support both insert and update.
* If this operation fails (by throwing an exception) after a partial success,
* the convention is that entries which have been written successfully are to be removed
* from the specified mapEntries, indicating that the write operation for the entries left
* in the map has failed or has not been attempted.
*
* @param elements the Elements to be written
*/
void writeAll(Collection<Element> elements) throws CacheException;
 /**
* Delete the cache entry from the store
*
* @param entry the cache entry that is used for the delete operation
*/
void delete(CacheEntry entry) throws CacheException;
 /**
* Remove data and keys from the underlying store for the given collection of keys, if
* present. If this operation fails * (by throwing an exception) after a partial success,
* the convention is that keys which have been erased successfully are to be removed from
* the specified keys, indicating that the erase operation for the keys left in the collection
* has failed or has not been attempted.
*
* @param entries the entries that have been removed from the cache
*/
void deleteAll(Collection<CacheEntry> entries) throws CacheException;
  /**
 * This method will be called whenever an Element couldn't be handled by the writer and all of
 * the {@link net.sf.ehcache.config.CacheWriterConfiguration#getRetryAttempts() retryAttempts} ha
 * <p>When batching is enabled, all of the elements in the failing batch will be passed to this m
 * <p>Try to not throw RuntimeExceptions from this method. Should an Exception occur, it will be
 * the element will still be lost.
 * @param element the Element that triggered the failure, or one of the elements in the batch tha
 * @param operationType the operation we tried to execute
 * @param e the RuntimeException thrown by the Writer when the last retry attempt was being execu
 */
void throwAway(Element element, SingleOperationType operationType, RuntimeException e);

}
```

FAQ

Is there a way to monitor the write-behind queue size?

Use the method `net.sf.ehcache.statistics.LiveCacheStatistics#getWriterQueueLength()`. This method returns the number of elements on the local queue (in all local buckets) that are waiting to be processed, or -1 if no write-behind queue exists. Note that elements or a batch currently being processed (and coalesced elements) are not included in the returned value.

What happens if an exception occurs when the writer is called?

Once all retry attempts have been executed, on exception the element (or all elements of that batch) will be passed to the `net.sf.ehcache.writer.CacheWriter#throwAway` method. The user can then act one last time on the element that failed to write. A reference to the last thrown RuntimeException, and the type of operation that failed to execute for the element, are received. Any Exception thrown from that method will simply be logged and ignored. The element will be lost forever. It is important that implementers are careful about proper Exception handling in that last method.

What happens if an exception occurs when the writer is called?

A handy pattern is to use an eternal cache (potentially using a writer, so it is persistent) to store failed operations and their element. Users can monitor that cache and manually intervene on those errors at a later point.

BlockingCache and SelfPopulatingCache

Introduction

The `net.sf.ehcache.constructs` package contains some applied caching classes which use the core classes to solve everyday caching problems. Two of these are BlockingCache and SelfPopulatingCache.

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. 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.

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.

Terracotta Cluster Events

Introduction

Beginning with Ehcache 2.0, the Terracotta Distributed Ehcache cluster events API provides access to Terracotta cluster events and cluster topology. This event-notification mechanism reports events related to the nodes in the Terracotta cluster, not cache events.

Cluster Topology

The interface `net.sf.ehcache.cluster.CacheCluster` provides methods for obtaining topology information for a Terracotta cluster. The following methods are available:

- `String getScheme()`

  Returns a scheme name for the cluster information. Currently `TERRACOTTA` is the only scheme supported.
- `Collection<ClusterNode> getNodes()`

  Returns information on all the nodes in the cluster, including ID, hostname, and IP address.
- `boolean addTopologyListener(ClusterTopologyListener listener)`

  Adds a cluster-events listener. Returns true if the listener is already active.
- `boolean removeTopologyListener(ClusterTopologyListener)`

  Removes a cluster-events listener. Returns true if the listener is already inactive.

The interface `net.sf.ehcache.cluster.ClusterNode` provides methods for obtaining information on specific cluster nodes.

```
public interface ClusterNode {
/**
* Get a unique (per cluster) identifier for this node.
*
* @return Unique per cluster identifier
*/
String getId();
/**
* Get the host name of the node
*
* @return Host name of node
*/
String getHostname();
/**
* Get the IP address of the node
*
* @return IP address of node
*/
String getIp();
}
```

Listening For Cluster Events

The interface `net.sf.ehcache.cluster.ClusterTopologyListener` provides methods for detecting the following cluster events:

```
public interface ClusterTopologyListener {
/**
* A node has joined the cluster
*
* @param node The joining node
*/
void nodeJoined(ClusterNode node);
/**
* A node has left the cluster
*
* @param node The departing node
*/
void nodeLeft(ClusterNode node);
/**
* This node has established contact with the cluster and can execute clustered operations.
*
* @param node The current node
*/
void clusterOnline(ClusterNode node);
/**
* This node has lost contact (possibly temporarily) with the cluster and cannot execute
* clustered operations
*
* @param node The current node
*/
void clusterOffline(ClusterNode node);
}
/**
* This node lost contact and rejoined the cluster again.
*
```

* This event is only fired in the node which rejoined and not to all the connected nodes * @param oldNode The old node which got disconnected * @param newNode The new node after rejoin */ void clusterRejoined(ClusterNode oldNode, ClusterNode newNode);

Example Code

This example prints out the cluster nodes and then registers a `ClusterTopologyListener` which prints out events as they happen.

```
CacheManager mgr = ...
CacheCluster cluster = mgr.getCluster("TERRACOTTA");
  // Get current nodes
Collection<ClusterNode> nodes = cluster.getNodes();
for(ClusterNode node : nodes) {
  System.out.println(node.getId() + " " + node.getHostname() + " " + node.getIp());
}
  // Register listener
cluster.addTopologyListener(new ClusterTopologyListener() {
  public void nodeJoined(ClusterNode node) { System.out.println(node + " joined"); }
  public void nodeLeft(ClusterNode node) { System.out.println(node + " left"); }
  public void clusterOnline(ClusterNode node) { System.out.println(node + " enabled"); }
  public void clusterOffline(ClusterNode node) { System.out.println(node + " disabled"); }
```

Example Code

```
  public void clusterRejoined(ClusterNode node, ClusterNode newNode) {
    System.out.println(node + " rejoined the cluster as " + newNode);
  }
});
```

Uses for Cluster Events

From Ehcache 2.4.1/Terracotta 3.5 these events are used for operation of NonStopCache

If Ehcache got disconnected from the Terracotta Server Array say due to a network issue, then in Ehcache 2.0 each cache operation will block indefinitely. In other words it is configured for fail-fast to protect the ACIDity of the cluster. However this approach will also cause processing of requests to the cache to stop likely causing the outage to cascade.

In some cases graceful degradation may be more appropriate. When the `clusterOffline` events fire you could call `Cache.setDisabled()`, which will cause puts and gets to bypass the cache. Your application would then degrade to operating without a cache, but might be able to do useful work. You could also take the whole application off-line. When connectivity is restored you could then reverse the action, taking the cache back online or the application back on line as the case may be.

Cache Decorators

Introduction

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.

Decorated caches are accessed from the CacheManager using `CacheManager.getEhcache(String name)`. Note that, for backward compatibility, `CacheManager.getCache(String name)` has been retained. However only `CacheManager.getEhcache(String name)` returns the decorated cache.

Creating a Decorator

Programmatically

Cache decorators are created as follows:

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

The class must implement Ehcache.

By Configuration

Cache decorators can be configured directly in ehcache.xml. The decorators will be created and added to the CacheManager.

It accepts the name of a concrete class that extends net.sf.ehcache.constructs.CacheDecoratorFactory

The properties will be parsed according to the delimiter (default is comma ',') and passed to the concrete factory's `createDecoratedEhcache(Ehcache cache, Properties properties)` method along with the reference to the owning cache.

It is configured as per the following example:

```
   <cacheDecoratorFactory
        class="com.company.SomethingCacheDecoratorFactory"
        properties="property1=36 ..." />
```

Note that from version 2.2, decorators can be configured against the `defaultCache`. This is very useful for frameworks like Hibernate that add caches based on the `defaultCache`.

Adding decorated caches to the CacheManager

Having created a decorator programmatically it is generally useful to put it in a place where multiple threads may access it. Note that decorators created via configuration in ehcache.xml have already been added to the

`CacheManager`.

Using `CacheManager.replaceCacheWithDecoratedCache()`

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 overwridden 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.

Using `CacheManager.addDecoratedCache()`

Sometimes you want to add a decorated cache but retain access to the underlying cache.

Using CacheManager.addDecoratedCache()

The way to do this is to create a decorated cache and then call `cache.setName(new_name)` and then add it to `CacheManager` with `CacheManager.addDecoratedCache()`.

```
/**
* Adds a decorated {@link Ehcache} to the CacheManager. This method neither creates
* the memory/disk store nor initializes the cache. It only adds the cache reference
* to the map of caches held by this cacheManager.
*
```

* It is generally required that a decorated cache, once constructed, is made available * to other execution threads. The simplest way of doing this is to either add it to * the cacheManager with a different name or substitute the original cache with the * decorated one. *

* This method adds the decorated cache assuming it has a different name. If another * cache (decorated or not) with the same name already exists, it will throw * {@link ObjectExistsException}. For replacing existing * cache with another decorated cache having same name, please use * {@link #replaceCacheWithDecoratedCache(Ehcache, Ehcache)} *

* Note that any overridden Ehcache methods by the decorator will take on new * behaviours without casting. Casting is only required for new methods that the * decorator introduces. For more information see the well known Gang of Four * Decorator pattern. * * @param decoratedCache * @throws ObjectExistsException * if another cache with the same name already exists. */ public void addDecoratedCache(Ehcache decoratedCache) throws ObjectExistsException {

Built-in Decorators

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`.

SelfPopulatingCache

A selfpopulating decorator for 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.

Caches with Exception Handling

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

CacheManager Event Listeners

Introduction

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.

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 CacheManagerEventListener, which is notified when Caches are added or removed from the CacheManager. The attributes of a 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.

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 4369 2011-07-15 19:59:14Z ilevy $
* @see "http://ehcache.org/documentation/cachemanager_event_listeners.html"
*/
```

Implementing a CacheManagerEventListenerFactoryand CacheManagerEventListener

```
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 4369 2011-07-15 19:59:14Z ilevy $
* @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 ClassLoader returned by `Thread.currentThread().getContextClassLoader()` to load classes.

Cache Event Listeners

Introduction

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. In clustered environments event propagation can be configured to be propagated only locally, only remotely, or both. The default is both, to be backwardly compatible.

Configuration

Cache event listeners are configured per cache. Each cache can have multiple listeners. Each listener is configured by adding a cacheEventListenerFactory element as follows:

```
<cache ...>
  <cacheEventListenerFactory class="" properties="" listenFor=""/>
  ...
</cache>
```

The entry specifies a CacheEventListenerFactory which is used to create a CacheEventListener, which then receives notifications. The attributes of a CacheEventListenerFactory are:

- class - a fully qualified factory class name
- properties - an optional comma separated properties having meaning only to the factory.
- listenFor - describes which events will be delivered in a clustered environment, defaults to 'all'.

  These are the possible values:

  - all - the default is to deliver all local and remote events
  - local - deliver only events originating in the current node
  - remote - deliver only events originating in other nodes

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.

Implementing a CacheEventListenerFactory and CacheEventListener

Implementing a CacheEventListenerFactory andCacheEventListener

A 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 following example demonstrates how to create an abstract CacheEventListenerFactory:

```
/**
* 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 4369 2011-07-15 19:59:14Z ilevy $
*/
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 following example demonstrates how to create a concrete implementation of the CacheEventListener interface:

```
/**
* 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 4369 2011-07-15 19:59:14Z ilevy $
* @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;
}
```

Two other methods are also available:

- `void notifyElementEvicted(Ehcache cache, Element element)`

  Called immediately after an element is evicted from the cache. Eviction, which happens when a cache entry is deleted from a store, should not be confused with removal, which is a result of calling `Cache.removeElement(Element)`.
- `void notifyRemoveAll(Ehcache cache)`

  Called during `Ehcache.removeAll()` to indicate that all elements have been removed from the cache in a bulk operation. The usual `notifyElementRemoved(net.sf.ehcache.Ehcache, net.sf.ehcache.Element)` is not called. Only one notification is emitted because performance considerations do not allow for serially processing notifications where potentially millions of elements have been bulk deleted.

The implementations need to be placed in the classpath accessible to Ehcache. See the chapter on Classloading for details on how the loading of these classes will be done.

Adding a Listener Programmatically

To add a listener programmatically, follow this example:

```
cache.getCacheEventNotificationService().registerListener(myListener);
```

Cache Exception Handlers

Introduction

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.

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>
```

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 4369 2011-07-15 19:59:14Z ilevy $
*/
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 4369 2011-07-15 19:59:14Z ilevy $
*/
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.

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);
```

Programmatic Configuration

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

Cache Extensions

Introduction

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 `createCacheCacheExtension()` 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.

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>
```

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 CacheExtensions. Implementers should
* provide their own * concrete factory extending this factory. It can then be configured
* in ehcache.xml.
*
* @author Greg Luck
* @version $Id: cache_extensions.apt 4369 2011-07-15 19:59:14Z ilevy $
*/
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);
}
```

Implementing a CacheExtensionFactory andCacheExtension

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.
*
* 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
* createCacheCacheExtension() 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.
*
* @author Greg Luck
* @version $Id: cache_extensions.apt 4369 2011-07-15 19:59:14Z ilevy $
*/
public interface CacheExtension {
/**
* Notifies providers to initialise themselves.
*
* 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.
*
* 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.
*
* 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.

Programmatic Configuration

Cache Extensions may also be programmatically added to a Cache as shown.

```
TestCacheExtension testCacheExtension = new TestCacheExtension(cache, ...);
testCacheExtension.init();
cache.registerCacheExtension(testCacheExtension);
```

Cache Eviction Algorithms

Introduction

A cache eviction algorithm is a way of deciding which element to evict when the cache is full. In Ehcache, the `MemoryStore` may be limited in size (see How to Size Caches for more information). When the store gets full, elements are evicted. The eviction algorithms in Ehcache determine which elements are 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 (is *flushed* to disk); otherwise it will be removed. The `DiskStore` size by default is unbounded. But a maximum size can be set (see Sizing Caches for more information). If the `DiskStore` is full, then adding an element will cause one to be evicted unless it is unbounded. The `DiskStore` eviction algorithm is not configurable. It uses LFU.

The local `DiskStore` is not used in distributed cache, which relies on the Terracotta Server Array for storage.

Provided 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 phenomenon and is the default in most caches.

A variation of LRU is the default eviction algorithm in Ehcache.

Altogether Ehcache provides three eviction algorithms to choose from for the `MemoryStore`.

Least Recently Used (LRU)

This is the default and is a variation on Least Frequently Used.

The oldest element is the Less Recently Used (LRU) element. 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.

This algorithm takes a random sample of the Elements and evicts the smallest. Using the sample size of 15 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 99% of the time.

If probabilistic eviction does not suit your application, a true Least Recently Used deterministic algorithm is available by setting `java -Dnet.sf.ehcache.use.classic.lru=true`.

Least Frequently Used (LFU)

Least 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 Least 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 15 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 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.

It takes a random sample of the Elements and evicts the smallest. Using the sample size of 15 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 99% of the time.

Plugging in your own Eviction Algorithm

Ehcache 1.6 and higher allows you to plugin in your own eviction algorithm. You can utilise any Element metadata which makes possible some very interesting approaches. For example, evict an Element if it has been hit more than 10 times.

```
/**
* Sets the eviction policy strategy. The Cache will use a policy at startup.
* There are three policies which can be configured: LRU, LFU and FIFO. However
* many other policies are possible. That the policy has access to the whole element
* enables policies based on the key, value, metadata, statistics, or a combination
* of any of the above.
*
* It is safe to change the policy of a store at any time. The new policy takes
* effect immediately.
*
* @param policy the new policy
*/
public void setMemoryStoreEvictionPolicy(Policy policy) {
  memoryStore.setEvictionPolicy(policy);
}
```

A Policy must implement the following interface:

```
public interface Policy {
  /**
   * @return the name of the Policy. Inbuilt examples are LRU, LFU and FIFO.
   */
  String getName();
  /**
   * Finds the best eviction candidate based on the sampled elements. What
   * distinguishes this approach from the classic data structures approach is
   * that an Element contains metadata (e.g. usage statistics) which can be used
   * for making policy decisions, while generic data structures do not. It is
   * expected that implementations will take advantage of that metadata.
   *
   * @param sampledElements this should be a random subset of the population
   * @param justAdded we probably never want to select the element just added.
   * It is provided so that it can be ignored if selected. May be null.
   * @return the selected Element
   */
  Element selectedBasedOnPolicy(Element[] sampledElements, Element justAdded);
  /**
   * Compares the desirableness for eviction of two elements
   *
   * @param element1 the element to compare against
   * @param element2 the element to compare
   * @return true if the second element is preferable for eviction to the first
   * element under ths policy
   */
  boolean compare(Element element1, Element element2);
}
```

DiskStore Eviction Algorithms

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

Class loading and Class Loaders

Introduction

Class loading, within the plethora of environments that 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`.

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 ClassLoader 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 ClassLoader that all classes in ehcache, and
 * extensions, should use for classloading. This is used if the context class loader
 * does not work.
 * @return the ClassLoaderUtil.class.getClassLoader();
 */
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.

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.

Classloading with Terracotta clustering

If Terracotta clustering is being used with valueMode="serialization" then keys and values will be moved across the cluster in byte[] and deserialized on other nodes.

The classloaders used (in order) to instantiate those classes will be:

- Thread.currentThread().getContextClassLoader()
- The classloader that defined the CacheManager initially

Operations Overview

The following sections provide a documentation Table of Contents and additional information sources about Ehcache operations.

Operations Table of Contents

Topic	Description
Tuning GC	Detecting Garbage Collection problems, Garbage Collection tuning, and distributed caching Garbage Collection tuning.
Ehcache Monitor	The Ehcache Monitor is an add-on tool for Ehcache which provides enterprise-class monitoring and management capabilities for use in both development and production. It is intended to help understand and tune cache usage, detect errors, and provide an easy-to-use access point to integrate with production management systems. It also provides administrative functionality, such as the ability to forcefully remove items from caches.
JMX Management	As an alternative to the Ehcache Monitor, JMX creates a standard way of instrumenting classes and making them available to a management and monitoring infrastructure.
Logging	Ehcache uses the the slf4j logging facade, so you can plug in your own logging framework. This page also provides recommended logging levels.
Shutting Down Ehcache	If you are using persistent disk stores, or distributed caching, care should be taken when shutting down Ehcache. This page covers the ServletContextListener, the shutdown hook, and dirty shutdown.
RMI Cache Remote Debugger	The Remote Debugger 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.

Additional Information about Operations

The following page provides additional information about the Ehcache Monitor:

- Terracotta Console for Enterprise Ehcache
- Cache Size and Statistics Code Samples
- CacheManager Shutdown Code Sample

Tuning Garbage Collection

Introduction

Applications that use Ehcache can be expected to use large heaps. Some Ehcache applications have heap sizes greater than 6GB. Ehcache works well at this scale. However, large heaps or long held objects, which is what a cache is composed of, can place strain on the default Garbage Collector. Now with Ehcache 2.3 and higher, this problem can be solved with BigMemory in-memory data management, which provides an additional store outside of the heap.

Note: The following documentation relates to Sun JDK 1.5.

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 every 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 amount of time the system was paused. A jump of more than a few seconds will not be acceptable in most application contexts.

Garbage Collection Tuning

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

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

Note that it is better to use `-XX:+DisableExplicitGC`, instead of calling `System.gc()`. It also helps to use the low pause collector `-XX:+UseConcMarkSweepGC`.

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 RMI has is similar to a user application calling `System.gc()` each minute. The setting above disables explicit GC calls, 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.

Ehcache Monitor

Introduction

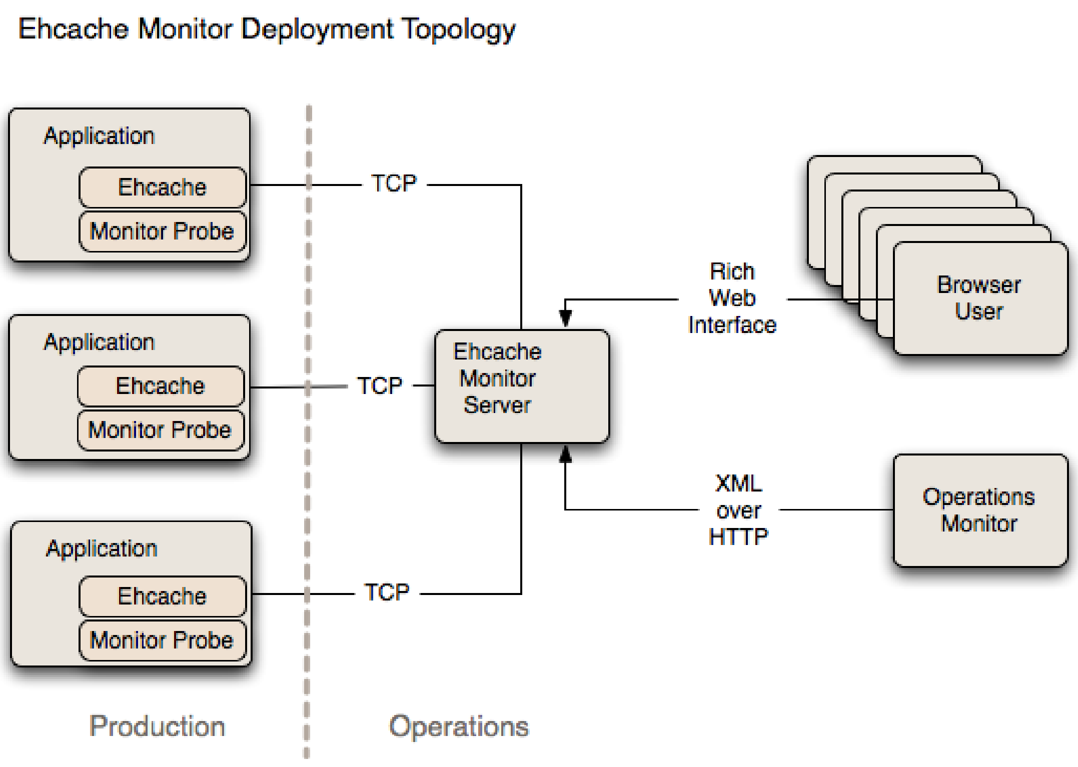
The Ehcache Monitor is an add-on tool for Ehcache provides enterprise-class monitoring and management capabilities for use in both development and production. It is intended to help understand and tune cache usage, detect errors, and provide an easy to use access point to integrate with production management systems. It also provides administrative functionality such as the ability to forcefully remove items from caches.

Simply install the Monitor on an Operations server, add the Monitor Probe jar to your app, add a few lines of config in ehcache.xml and your done. The package contains a probe and a server. The probe installs with your existing Ehcache cache instance, and communicates to a central server. The server aggregates data from multiple probes. It can be accessed via a simple web UI, as well as a scriptable API. In this way, it is easy to integrate with common third party systems management tools (such as Hyperic, Nagios etc). The probe is designed to be compatible with all versions of Ehcache from 1.5 and requires JDK 1.5 or 1.6.

Installation And Configuration

First download and extract the Ehcache Monitor package. The package consists of a lib directory with the probe and monitor server jars, a bin directory with startup and shutdown scripts for the monitor server and an `etc` directory with an example monitor server configuration file and a Jetty Server configuration file.

Recommended Deployment Topology



It is recommended that you install the Monitor on an Operations server separate to production. The Monitor acts as an aggregation point for access by end users and for scripted connection from Operations tools for data feeds and set up of alerts.

Probe

To include the probe in your Ehcache application, you need to perform two steps:

1. Add the ehcache-probe-.jar to your application classpath (or war file). Do this in the same way you added the core ehcache jar to your application. If you are Maven based, the probe module is in the Terracotta public repository for easy integration.

   ```
   <repository>
     <id>terracotta-releases</id>
     <url>http://www.terracotta.org/download/reflector/releases</url>
   </repository>
   <dependency>
     <groupId>org.terracotta</groupId>
     <artifactId>ehcache-probe</artifactId>
     <version>[version]</version>
   </dependency>
   ```
2. Configure Ehcache to communicate with the probe by specifying the class name of the probe, the address (or hostname), the port that the monitor will be running on and whether to do memory measurement. This is done by adding the following to ehcache.xml:

   ```
   <cacheManagerPeerListenerFactory
   class="org.terracotta.ehcachedx.monitor.probe.ProbePeerListenerFactory"
   properties="monitorAddress=localhost, monitorPort=9889, memoryMeasurement=true" />
   ```
3. Include required SLF4J logging jars.

   Ehcache 1.7.1 and above require SLF4J. Earlier versions used commons logging. The probe, like all new Ehcache modules, uses SLF4J, which is becoming a new standard in open source projects.

   If you are using Ehcache 1.5 to 1.7.0, you will need to add slf4j-api and one concrete logger. If you are using Ehcache 1.7.1 and above you should not need to do anything because you will already be using slf4j-api and one concrete logger.

   More information on SLF4J is available from http://www.slf4j.org.
4. Ensure that statistics capture in each cache is turned on for the probe to gather statistics. Statistics were turned off by default from Ehcache 2.1 onwards.

   ```
   <cache name="sampleCache2"
     maxEntriesLocalHeap="1000"
     eternal="true"
     overflowToDisk="false"
     memoryStoreEvictionPolicy="FIFO"
     statistics="true"
     />
   ```

Starting the Monitor

Copy the monitor package to a monitoring server.

Starting the Monitor

To start the monitor, run the startup script provided in the bin directory: startup.sh on Unix and startup.bat on Microsoft Windows.

NOTE: If errors occur at startup, remove the line `-j "$PRGDIR/etc/jetty.xml" \` (or `-j %PRGDIR%\etc\jetty.xml ^`) from the startup script.

The monitor port selected in this script should match the port specified in ehcache.xml. The monitor can be configured, including interface, port and simple security settings, in `etc/ehcache-monitor.conf`. Note that for the commercial version, the location of your license file must be specified in `ehcache-monitor.conf`.

For example:

```
license_file=/Users/karthik/Documents/workspace/lib/license/terracotta-license.key
```

The monitor connection timeout can also be configured. If the monitor is frequently timing out while attempting to connect to a node (due to long GC cycles, for example), then the default timeout value may not be suitable for your environment. You can set the monitor timeout using the system property `ehcachedx.connection.timeout.seconds`. For example, `-Dehcachedx.connection.timeout.seconds=60` sets the timeout to 60 seconds.

Securing the Monitor

The Monitor can be secured in a variety of ways. The simplest method involves simply editing `ehcache-monitor.conf` to specify a single user name and password. This method has the obvious drawbacks that (1) it provides only a single login identity, and (2) the credentials are stored in clear-text.

A more comprehensive security solution can be achieved by configuring the Jetty Server with one ore more `UserRealm`s as described by Jetty and JAAS. Simply edit `etc/jetty.xml` to use the appropriate `UserRealm` implementation for your needs. To configure the Monitor to authenticate against an existing LDAP server, first ensure that you have defined and properly registered a `LoginConfig`, such as the following example:

```
MyExistingLDAPLoginConfig {
  com.sun.security.auth.module.LdapLoginModule REQUIRED
  java.naming.security.authentication="simple"
  userProvider="ldap://ldap-host:389"
  authIdentity="uid={USERNAME},ou=People,dc=myorg,dc=org"
  useSSL=false
  bindDn="cn=Manager"
  bindCredential="secretBindCredential"
  bindAuthenticationType="simple"
  debug=true;
};
```

Note: the `LdapLoginModule` is new with JDK 1.6.

JAAS supports many different types of login modules and it is up to the reader to provide a valid, working JAAS environment. For more information regarding JAAS refer to JAAS Reference Guide. For information on how to register your LoginConfig refer to `$JAVA_HOME/jre/lib/security/java.security`.

Next, edit `etc/jetty.xml`:

```
<?xml version="1.0"?>
<!DOCTYPE Configure PUBLIC "-//Mort Bay Consulting//DTD Configure//EN"
 "http://jetty.mortbay.org/configure.dtd">
<Configure id="Server" class="org.terracotta.ehcachedx.org.mortbay.jetty.Server">
  <Set name="UserRealms">
    <Array type="org.terracotta.ehcachedx.org.mortbay.jetty.security.UserRealm">
      <Item>
        <New class="org.terracotta.ehcachedx.org.mortbay.jetty.plus.jaas.JAASUserRealm">
          <Set name="Name">MyArbitraryLDAPRealmName</Set>
          <Set name="LoginModuleName">MyExistingLDAPLoginConfig</Set>
        </New>
      </Item>
    </Array>
  </Set>
</Configure>
```

The `LoginModuleName` you specify as the second constructor parameter to the `JAASUserRealm` class must exactly match the name of your `LoginModule`. The realm name specified as the first constructor parameter can be an arbitrary value.

Note: the version of Jetty used in the Monitor has been repackaged so be sure to prefix any standard Jetty class names with `org.terracotta.ehcachedx`.

If the Jetty Server is found to have been configured with any security realms, the simple user name and password from `ehcache-monitor.conf` is ignored.

Using the Web GUI

The web-based GUI is available by pointing your browser at http://:/monitor. For a default installation on the local machine, this would be http://localhost:9889/monitor The GUI contains six tabs, described as follows:

Cache Managers

This tab shows aggregate statistics for the cache managers being monitored by probes connected to the monitor server. Double-clicking on any cache manager drills down to the detailed Statistics tab for that manager.

Statistics

This tab shows the statistics being gathered for each cache managed by the selected cache manager. The Settings button permits you to add additional statistics fields to the display. Note: only displayed fields are collected and aggregated by the probe. Adding additional display fields will increase the processing required for probe and the monitor. The selected settings are stored in a preferences cookie in your browser. Double-clicking on any cache drills down to the Contents tab for that cache.

Configuration

This tab shows the key configuration information for each cache managed by the selected cache manager.

Contents

This tab enables you to look inside the cache, search for elements via their keys and remove individual or groups of elements from the cache. The GUI is set to refresh at the same frequency that the probes aggregate their statistic samples which is every 10 seconds by default. The progress bar at the bottom of the screen indicates the time until the next refresh.

Charts

This tab contains various live charts of cache statistics. It gives you a feel for the trending of the each statistic, rather than just the latest value.

Estimated Memory Use Chart

This chart shows the estimated memory use of the Cache. Memory is estimated by sampling. The first 15 puts or updates are measured and then every 100th put or update. Most caches contain objects of similar size. If this is not the case for your cache, then the estimate will not be accurate. Measurements are performed by walking the object graph of sampled elements through reflection. In some cases such as classes not visible to the classloader, the measurement fails and 0 is recorded for cache size. If you see a chart with 0 memory size values but the cache has data in it, then this is the cause. For this release, caches distributed via Terracotta server show as 0.

API

This tab contains a listing of the API methods. Each is a hyperlink, which may be clicked on. Some will display data and some will require additional arguments. If additional arguments are required an error message will be displayed with the details. This tab is meant for iterative testing of the API.

Using the API

The Monitor provides a API over HTTP on the same port as the Web GUI. The list of functions supported by the API can be accessed by pointing your browser at http://:/monitor/list. For a default installation on the local machine, this would be http://localhost:9889/monitor/list. The API returns data as either structured XML or plan text. The default format is txt. For example, the getVersion function returns the software version of the monitor server. It can be called as follows: http://localhost:9889/monitor/getVersion or, to receive the results as XML: http://localhost:9889/monitor/getVersion?format=xml

To query the data collected by the monitor server from scripts that can then be used to pass the data to enterprise system management frameworks, commands such as `curl` or `wget` can be used. For example, on a Linux system, to query the list of probes that a local monitor on the default port is currently aware of, and return the data in XML format, the following command could be used:

```
$ curl http://localhost:9889/monitor/listProbes?format=xml
```

Licensing

Unless otherwise indicated, this module is licensed for usage in development. For details see the license terms in the appropriate LICENSE.txt. To obtain a commercial license for use in production, please contact sales@terracottatech.com

Limitations

History not Implemented

This release has server side history configuration in place, however history is not implemented. It is anticipated it will be implemented in the next release. In the meantime, the charts with their recent history provide trending.

Memory Measurement limitations

Unfortunately in Java, there is no JSR for memory measurement of objects. Implementations, such as the sizeof one we use are subject to fragilities. For example, Java 7 memory measurement is not supported at this time. You will get a `java.lang.NoSuchFieldException: header` exception message if you use memory measurement with Java 7. Memory measurement can optionally be turned off by setting `memoryMeasurement=false` in the probe configuration.

JMX Management and Monitoring

Introduction

As an alternative to the Ehcache Monitor, JMX creates a standard way of instrumenting classes and making them available to a management and monitoring infrastructure.

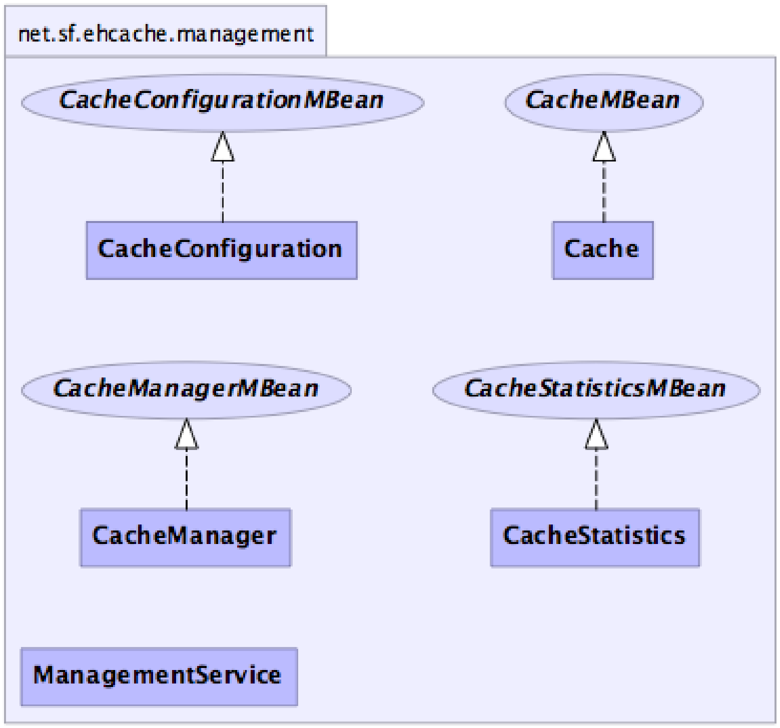
Terracotta Monitoring Products

An extensive monitoring product, available in Enterprise Ehcache, 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-monitor documentation for more information. When using Ehcache 1.7 with Terracotta clustering, the Terracotta Developer Console shows statistics for Ehcache.

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 is illustrated in the follwing image.



generated by yDoc

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.

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

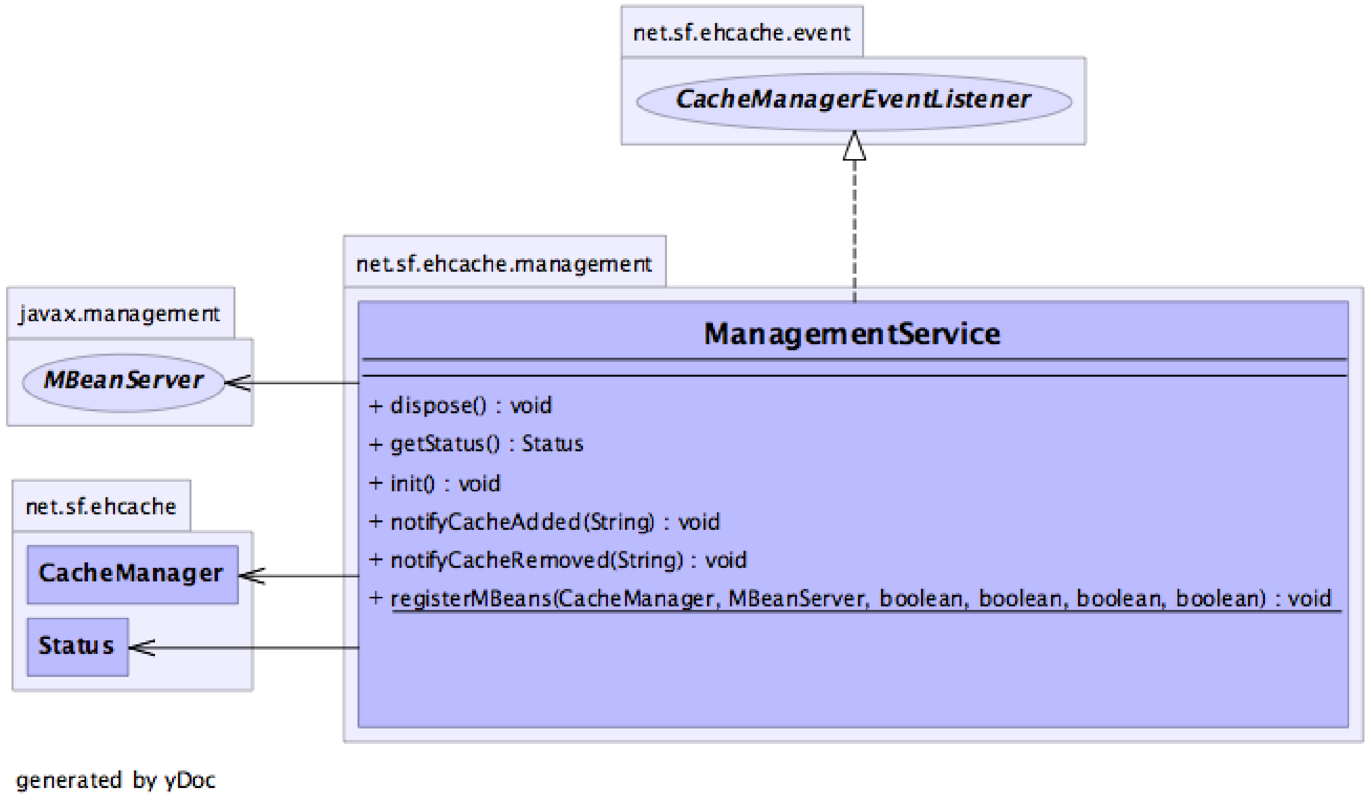
Most attributes use built-in types. To access all attributes, you need to add ehcache.jar to the remote JMX client's classpath e.g. `jconsole -J-Djava.class.path=ehcache.jar`.

`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>"

The Management Service

The `ManagementService` class is the API entry point.


net.sf.ehcache.event
CacheManagerEventListener
net.sf.ehcache.management
ManagementService
+ dispose() : void
+ getStatus() : Status
+ init() : void
+ notifyCacheAdded(String) : void
+ notifyCacheRemoved(String) : void
+ registerMBeans(CacheManager, MBeanServer, boolean, boolean, boolean, boolean) : void
javax.management
MBeanServer
net.sf.ehcache
CacheManager
Status
generated by yDoc

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.
*
```

```
* 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.
*
* 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 {
```

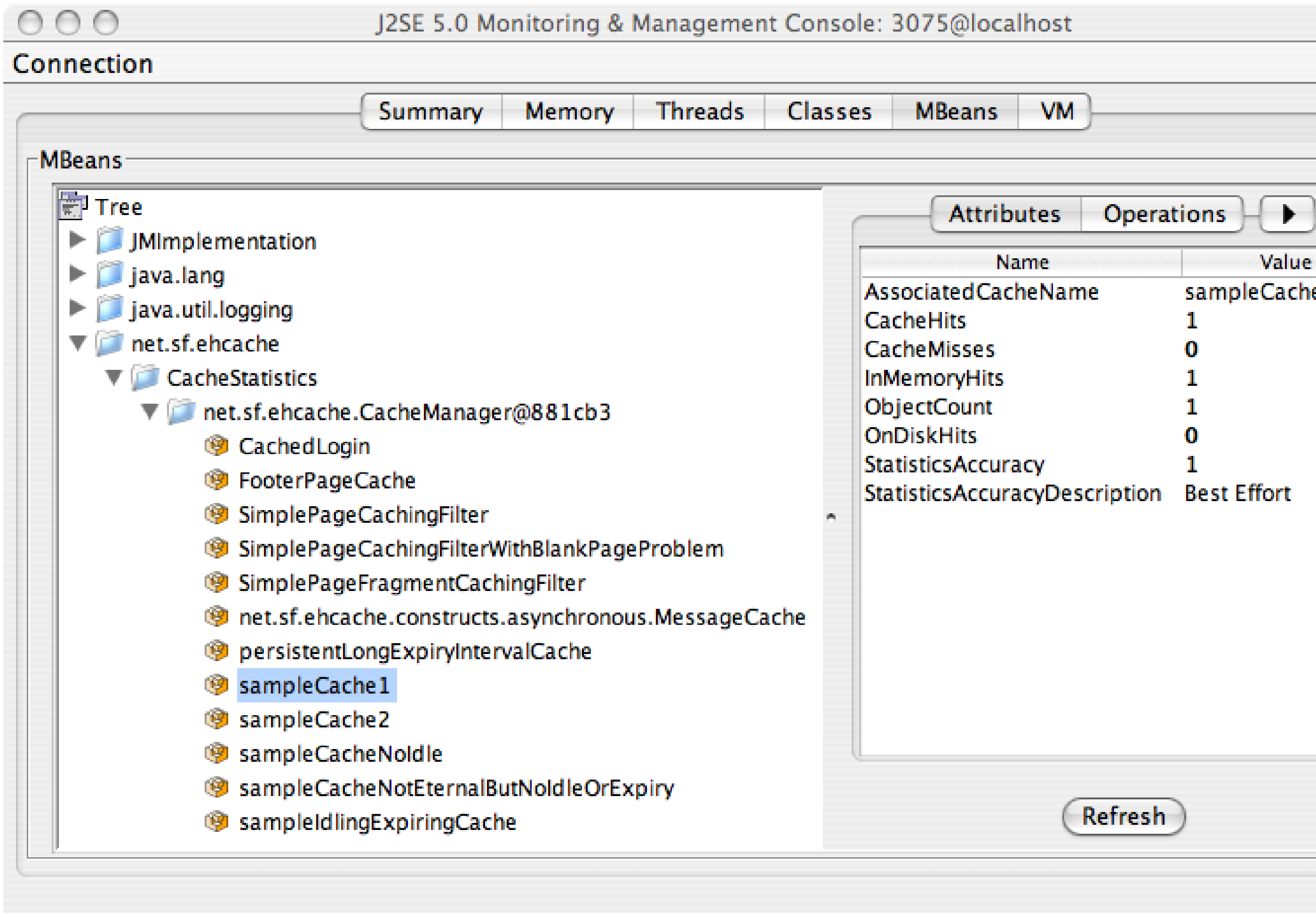
JConsole Example

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.

JConsole Example


J2SE 5.0 Monitoring & Management Console: 3075@localhost
Connection
Summary
Memory
Threads
Classes
MBeans
VM
MBeans
Tree
JMImplementation
java.lang
java.util.logging
net.sf.ehcache
CacheStatistics
net.sf.ehcache.CacheManager@881cb3
CachedLogin
FooterPageCache
SimplePageCachingFilter
SimplePageCachingFilterWithBlankPageProblem
SimplePageFragmentCachingFilter
net.sf.ehcache.constructs.asynchronous.MessageCache
persistentLongExpiryIntervalCache
sampleCache1
sampleCache2
sampleCacheNoIdle
sampleCacheNotEternalButNoIdleOrExpiry
sampleIdlingExpiringCache
Attributes
Operations
Refresh

Name	Value
AssociatedCacheName	sampleCache
CacheHits	1
CacheMisses	0
InMemoryHits	1
ObjectCount	1
OnDiskHits	0
StatisticsAccuracy	1
StatisticsAccuracyDescription	Best Effort

CacheStatistics MBeans in JConsole

Hibernate statistics

If you are running Terracotta clustered caches as hibernate second-level cache provider, it is possible to access the hibernate statistics + ehcache stats etc via jmx. `EhcacheHibernateMBean` is the main interface that exposes all the API's via jmx. It basically extends two interfaces -- `EhcacheStats` and `HibernateStats`. And as the name implies `EhcacheStats` contains methods related with Ehcache and `HibernateStats` related with Hibernate. You can see cache hit/miss/put rates, change config element values dynamically -- like maxElementInMemory, TTI, TTL, enable/disable statistics collection etc and various other things. Please look into the specific interface for more details.

JMX Tutorial

See this online tutorial.

Performance

Collection of cache statistics is not entirely free of overhead. In production systems where monitoring is not required statistics can be disabled. This can be done either programatically by calling `setStatisticsEnabled(false)` on the cache instance, or in configuration by setting the `statistics="false"` attribute of the relevant cache configuration element. From Ehcache 2.1.0 statistics are off by default.

Logging

Introduction

As of 1.7.1, Ehcache uses the the slf4j logging facade, so you can plug in your own logging framework. This page covers Ehcache logging. For more information about slf4j in general, refer to the slf4j site.

SLF4J Logging

With slf4j, users must choose a concrete logging implementation at deploy time. The options include Maven and the download kit.

Concrete Logging Implementation Use in Maven

The maven dependency declarations are reproduced here for convenience. Add *one* of these to your Maven pom.

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jdk14</artifactId>
    <version>1.5.8</version>
</dependency>
 <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.5.8</version>
</dependency>
 <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.5.8</version>
</dependency>
```

Concrete Logging Implemenation Use in the Download Kit

We provide the slf4j-api and slf4j-jdk14 jars in the kit along with the ehcache jars so that, if the app does not already use SLF4J, you have everything you need. Additional concrete logging implementations can be downloaded from http://www.slf4j.org.

Recommended Logging Levels

Ehcache seeks to trade off informing production support developers or important messages and cluttering the log. ERROR messages should not occur in normal production and indicate that action should be taken.

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

Shutting Down Ehcache

Introduction

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.

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>
```

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.

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.

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:

What the shutdown hook does

- 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

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

The shutdown hook runs when:

- A program exists normally. For example, `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.

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.

Remote Network debugging and monitoring for Distributed Caches

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.

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 on the download page.

Limitations

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

Usage

It is invoked as follows:

```
java -jar ehcache-debugger-1.5.0.jar ehcache.xml sampleCache1 path_to_ehcache.xml [cacheToMonitor
```

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.
NOTE: Adding Application Libraries to the Classpath Use the Class-Path attribute inside a manifest file to add any additional libraries to the debugger's classpath. Be sure to leave a blank line at the end of the file. Following is an example of Class-Path attribute inside a manifest file (showing two separate JARs added to the classpath):

```
Class-Path: lib/my.jar lib/myLoggerjar
```

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.

Output

```
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.
```

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 `net.sf.ehcache.distribution` 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.

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.

Replication Overview

The following sections provide a documentation Table of Contents and additional information sources about replication.

Replication Table of Contents

Topic	Description
RMI Replicated Caching	Ehcache provides replicated caching using RMI. To set up RMI replicated caching, you need to configure the CacheManager with a PeerProvider and a CacheManagerPeerListener. Then for each cache that will be replicated, you 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.
JGroups Replicated Caching	JGroups can be used as the underlying mechanism for the replication operations in Ehcache. JGroups offers a very flexible protocol stack, reliable unicast, and multicast message transmission. To set up replicated caching using JGroups, you need to configure a PeerProviderFactory. For each cache that will be replicated, you then need to add a cacheEventListenerFactory to propagate messages.
JMS Replicated Caching	JMS can also be used as the underlying mechanism for replication operations in Ehcache. 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, pushing of data directly to cache nodes from external topic publishers, and a JMSCacheLoader, which sends cache load requests to a queue.

Additional Information about Replication

The following pages provide additional information about replicated caching with Ehcache:

- Replicated Caching FAQ
- Hibernate and Replicated Caching

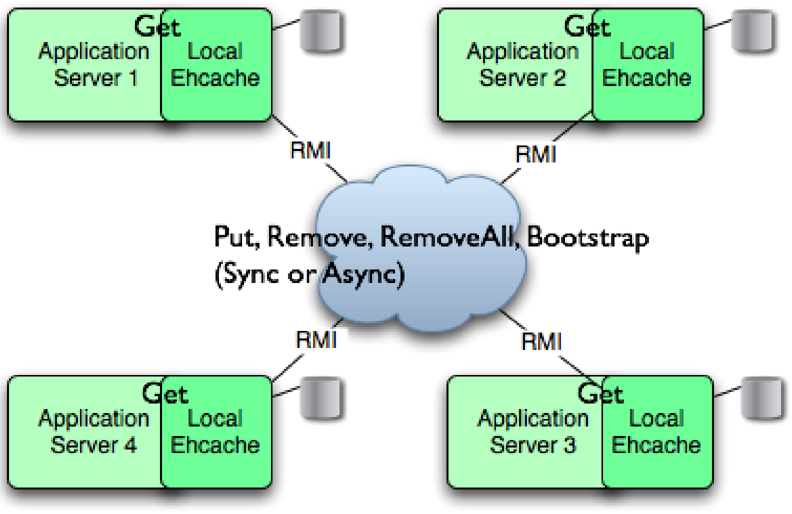
RMI Replicated Caching

Introduction

Since version 1.2, Ehcache has provided replicated 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 replicated caching you need to configure the CacheManager with:

- a PeerProvider
- a CacheManagerPeerListener

The for each cache that will be replicated, 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.

Suitable Element Types

Only Serializable Elements are suitable for replication.

Suitable Element Types

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.

Configuring the Peer Provider

Peer Discovery

Ehcache has the notion of a group of caches acting as a replicated 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.

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"/>
```

Manual Peer Discovery {#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"/>
```

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 RMICacheManagerPeerListener which is configured using RMICacheManagerPeerListenerFactory. It is configured as per the following example:

```
<cacheManagerPeerListenerFactory
```

Configuring the CacheManagerPeerListener

```
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.

Configuring Cache Replicators

Each cache that will be replicated 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"
  maxEntriesLocalHeap="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"
      maxEntriesLocalHeap="10"
      eternal="true"
      overflowToDisk="false"
      memoryStoreEvictionPolicy="LFU">
   <cacheEventListenerFactory
       class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>
</cache>
```

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 cache replication 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.

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/RMICacheReplicatorTest.html

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

Common Problems

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.

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.

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.

Replicated Caching using JGroups

Introduction

JGroups can be used as the underlying mechanism for the replication 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 replicated caching using JGroups you need to configure a PeerProviderFactory of type JGroupsCacheManagerPeerProviderFactory which is done globally for a CacheManager

For each cache that will be replicated, you then need to add a cacheEventListenerFactory of type JGroupsCacheReplicatorFactory to propagate messages.

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.

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.

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 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.

Example configuration using UDP Multicast

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.

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 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.

Example configuration using UDP Multicast

Suppose you have two servers in a cluster. You wish to replicated 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="::"
/>
```

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="::" />
```

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, Protocol considerations.

pbcast.NAKACK (Message reliability), pbcast.STABLE (message garbage collection).

Configuring CacheReplicators

Each cache that will be replicated 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"
  maxEntriesLocalHeap="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

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"
    maxEntriesLocalHeap="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>
```

Common Problems

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

```
<cacheManagerPeerProviderFactory class="net.sf.ehcache.distribution.jgroups.JGroupsCacheManagerPe
<cache name="sampleCacheAsync"
  maxEntriesLocalHeap="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, see the example programs in the JGroups documentation.

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`.

Replicated Caching using JMS

Introduction

As of version 1.6, JMS can be used as the underlying mechanism for the replicated 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.

Ehcache Replication and External Publishers

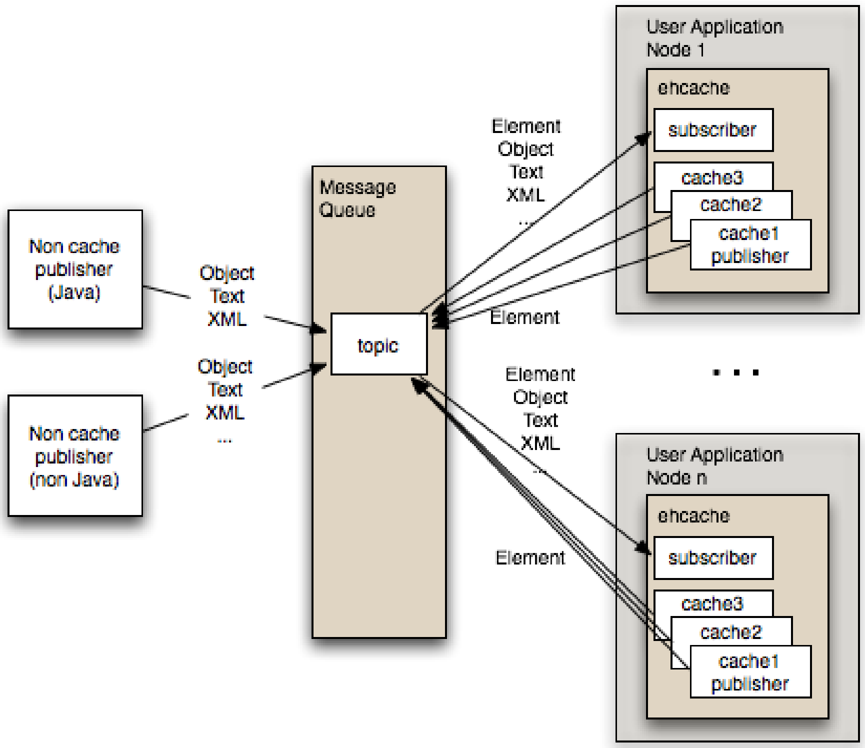
Ehcache replicates using JMS as follows:

- 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 replicated caching using JMS you need to configure a JMSCacheManagerPeerProviderFactory which is done globally for a CacheManager.

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

Ehcache Replication and External Publishers



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}.
```

Configuration

```
===========================


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"
 maxEntriesLocalHeap="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>
```

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.

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.

Example Configuration Using Active MQ

```
<cache name="sampleCacheNorep"
  maxEntriesLocalHeap="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>
```

Example Configuration Using Open MQ

```
<cache name="sampleCacheNorep"
  maxEntriesLocalHeap="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>
```

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.

Tested Message Queues

Sun Open MQ

This open source message queue is tested in integration tests. It works perfectly.

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.

Oracle AQ

Versions up to an including 0.4 do not work, due to Oracle not supporting the unified API (send) for topics.

JBoss Queue

Works as reported by a user.

Known JMS Issues

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.

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.

Modules Overview

The following sections provide a documentation Table of Contents and additional information sources about the Ehcache modules.

Modules Table of Contents

Topic	Description
SOAP and RESTful Cache Server	The Ehcache Cache Server has two APIs: RESTful resource oriented, and SOAP. The Ehcache RESTFul Web Services API exposes the singleton CacheManager. Resources are identified using a URI template. Ehcache's W3C Web Services support the WS-I definition and use the SOAP and WSDL specifications.
Web Caching	Ehcache provides a set of general purpose web caching filters in the ehcache-web module. Using these can make a significant difference to web application performance. With built-in gzipping, storage and network transmission are highly efficient. Cache pages and fragments make excellent candidates for DiskStore storage.

Additional Information about the Modules

The following pages provide additional information about the Ehcache modules:

- Download Information
- JMX Management and Monitoring
- Web Caching Discussion

Cache Server

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 the Oreilly catalog.)

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 Web Application Description Language (WADL) 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.

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.

CacheManager Resource Operations

OPTIONS /{cache}}

Retrieves the WADL for describing the available CacheManager operations.

GET} /

Lists the Caches in the CacheManager.

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.

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\ {#GET} /

Lists the Caches in the CacheManager.

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.

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()`.

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.

{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"?>
```

{RESTful Code Samples}

```
<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

{RESTful 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
```

Creating Massive Caches with Load Balancers and Partitioning

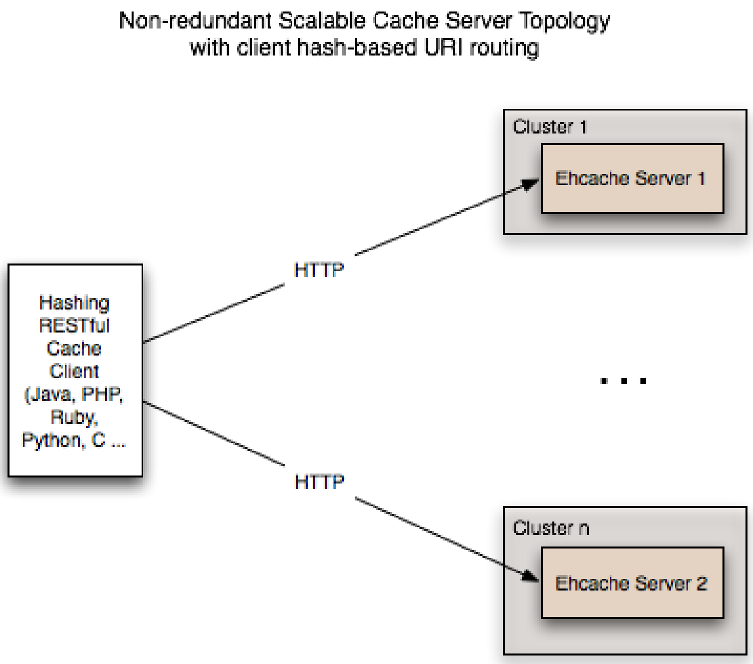
The RESTful Ehcache Server is designed to achieve massive scaling using data partitioning - all from a RESTful interface. The largest Ehcache single instances run at around 20GB in memory. The largest disk stores run at 100Gb each. Add nodes together, with cache data partitioned across them, to get larger 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.

Non-redundant, Scalable with client hash-based routing

Non-redundant, Scalable with client hash-based routing

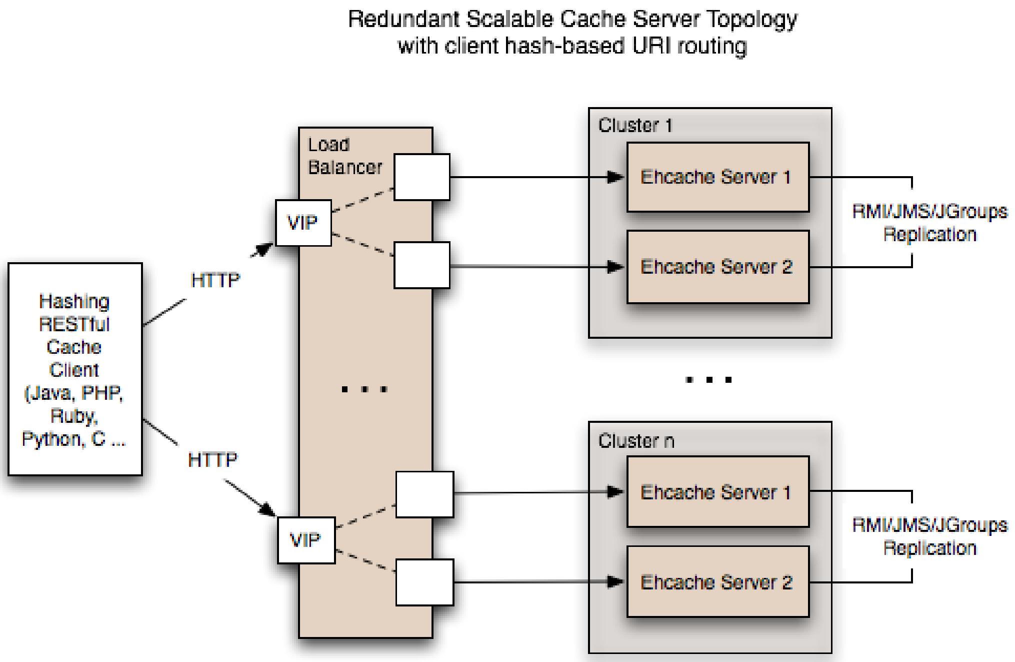

Non-redundant Scalable Cache Server Topology
with client hash-based URI routing
Hashing RESTful Cache Client (Java, PHP, Ruby, Python, C ...
HTTP
HTTP
Cluster 1
Ehcache Server 1
...
Cluster n
Ehcache Server 2

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];
```

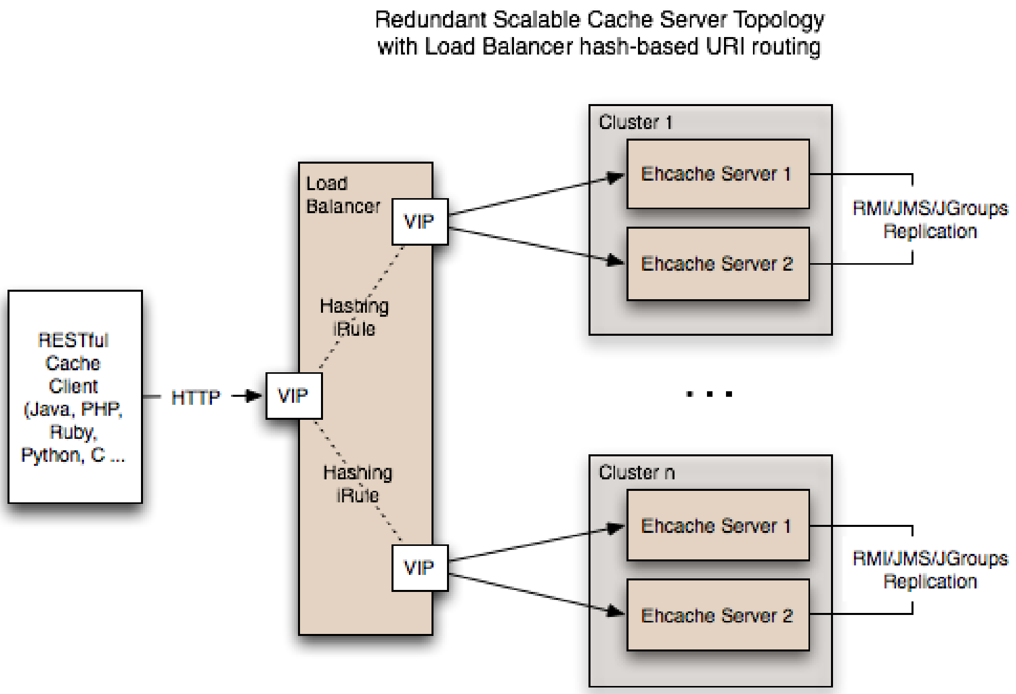
Redundant, Scalable with client hash-based routing

Redundant, Scalable with client hash-based routing


Redundant Scalable Cache Server Topology
with client hash-based URI routing
Load Balancer
VIP
VIP
. . .
Hashing RESTful Cache Client (Java, PHP, Ruby, Python, C ...
HTTP
HTTP
Cluster 1
Ehcache Server 1
Ehcache Server 2
RMI/JMS/JGroups Replication
. . .
Cluster n
Ehcache Server 1
Ehcache Server 2
RMI/JMS/JGroups Replication

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.

Redundant, Scalable with load balancer hash-based routing


Redundant Scalable Cache Server Topology
with Load Balancer hash-based URI routing
Load Balancer
VIP
Hashing iRule
VIP
Hashing iRule
VIP
RESTful Cache Client (Java, PHP, Ruby, Python, C ...
HTTP
Cluster 1
Ehcache Server 1
Ehcache Server 2
RMI/JMS/JGroups Replication
. . .
Cluster n
Ehcache Server 1
Ehcache Server 2
RMI/JMS/JGroups Replication

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:

Redundant, Scalable with load balancer hash-based routing

```
/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 this article for how to implment a URI hashing iRule on F5 load balancers.

W3C (SOAP) Web Services

The W3C 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. An interoperability organisation, WS-I, seeks to achieve interoperability 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, 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 specifications. 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.

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

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. 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:

1. Add configuration in accordance with XWSS to the `server_security_config.xml` file.
2. Create a class which implements the `CallbackHandler` interface and provide its fully qualified path in the `SecurityEnvironmentHandler` element.
3. Use the integration test `EhcacheWebServiceEndpoint` to see how to use the XWSS client side.
4. On the client side, make sure configuration is provided in a file called `client_security_config.xml`, which must be in the root of the classpath.
5. 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();
```

Requirements

Java

Java 5 or 6.

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

Downloading

The server is available as follows:

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.

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>
```

Installation

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.

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

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.

Configuring the Standalone Server

Configuration is by editing the `war/web.xml` file as per the instructions for the WAR packaging.

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	Ehcache Standalone Server Effect
1	HUP	Restarts the server.
2	INT	Interrupts the server.
9	KILL	The process is killed. The server is not given a chance to shutdown.
15	TERM	Stops the server, giving it a chance to shutdown in an orderly way.

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
```

Monitoring

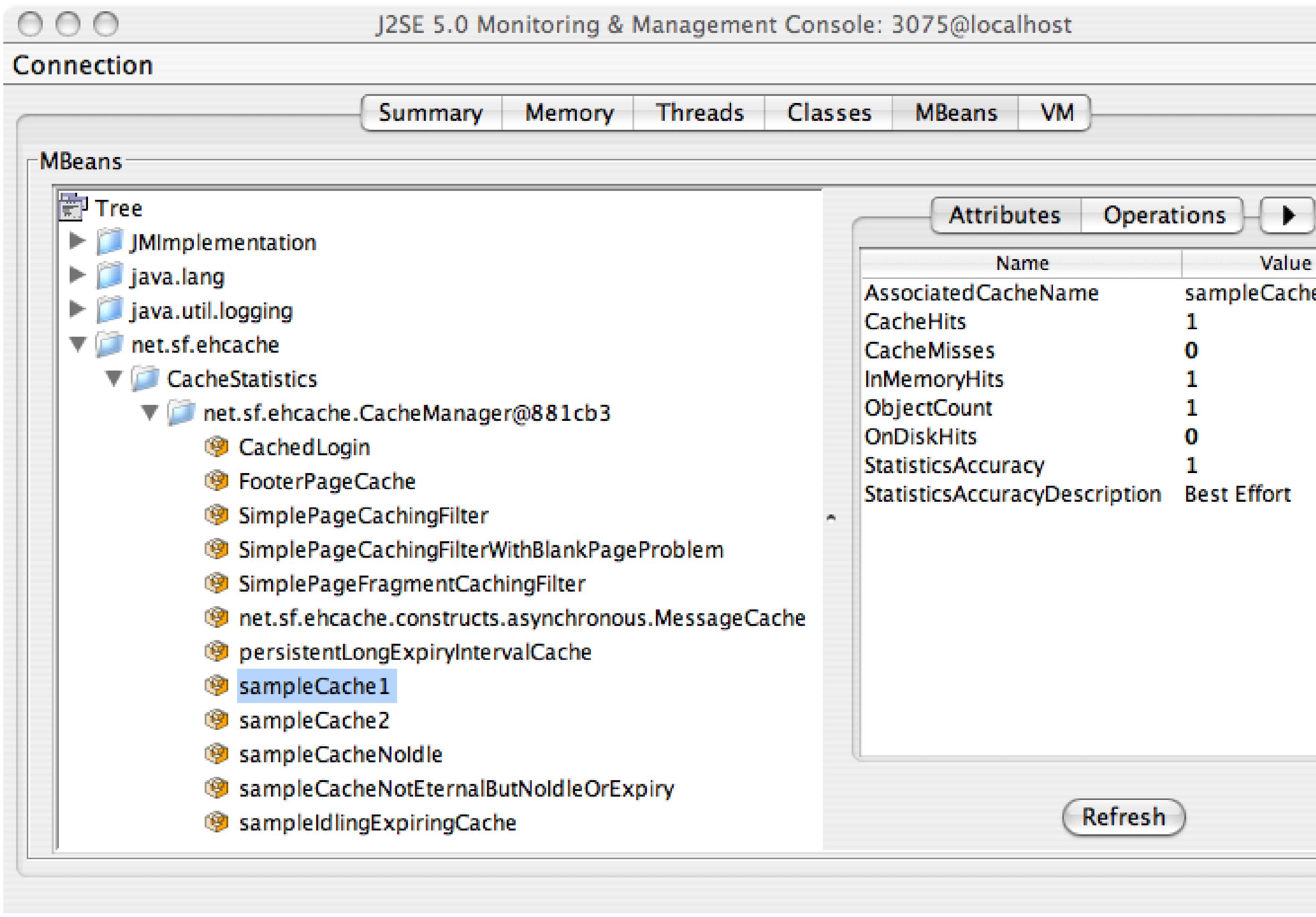
The CacheServer registers Ehcache MBeans with the platform MBeanServer. Remote monitoring of the MBeanServer is the responsibility of the Web Container or Application Server vendor. For example, some instructions for Tomcat are here. See your Web Container documentation for how to do this for your web container.

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 port. In 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.



```
JMImplementation
java.lang
java.util.logging
net.sf.ehcache
  CacheStatistics
    net.sf.ehcache.CacheManager@881cb3
      CachedLogin
      FooterPageCache
      SimplePageCachingFilter
      SimplePageCachingFilterWithBlankPageProblem
      SimplePageFragmentCachingFilter
      net.sf.ehcache.constructs.asynchronous.MessageCache
      persistentLongExpiryIntervalCache
      sampleCache1
      sampleCache2
      sampleCacheNoIdle
      sampleCacheNotEternalButNoIdleOrExpiry
      sampleIdlingExpiringCache
```

Name	Value
AssociatedCacheName	sampleCache
CacheHits	1
CacheMisses	0
InMemoryHits	1
ObjectCount	1
OnDiskHits	0
StatisticsAccuracy	1
StatisticsAccuracyDescription	Best Effort

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.

Download

Download the ehcache-standalone-server from sourceforge.net.

FAQ

Does Cache Server work with WebLogic?

Yes (we have tested 10.3.2), but the SOAP libraries are not compatible. Either comment out the SOAP service from web.xml or do the following:

1. Unzip `ehcache-server.war` to a folder called `ehcache`.
2. Remove the following jars from `WEB-INF/lib`:
   - jaxws-rt-2.1.4.jar
   - metro-webservices-api-1.2.jar
   - metro-webservices-rt-1.2.jar
   - metro-webservices-tools-1.2.jar
3. Deploy the folder to WebLogic.
4. Use the soapUI GUI in WebLogic to add a project from http://:/ehcache/soap/EhcacheWebServiceEndpoint?wsdl

Web Caching

Introduction

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[]`.

SimplePageCachingFilter

This is a simple caching filter suitable for caching compressible 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. For example, text or text/html

For fragments see the SimplePageFragmentCachingFilter.

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.

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 `cacheName`, or by sub-classing this class and overriding the name.

Concurrent 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 `blockingTimeoutMillis`. 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.

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.

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.

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.
- `varyHeader` - set to true to set Vary:Accept-Encoding in the response when doing Gzip. This header is needed to support HTTP proxies however it is off by default.

  ```
  <init-param>
    <param-name>varyHeader</param-name>
    <param-value>true</param-value>
  </init-param>
  ```

Re-entrance

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.

SimplePageFragmentCachingFilter

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

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
version="2.5">

 <filter>
<filter-name>CachePage1CachingFilter</filter-name>
<filter-class>net.sf.ehcache.constructs.web.filter.SimplePageCachingFilter
</filter-class>
<init-param>
 <param-name>suppressStackTrace</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.constr
</filter-class>
<init-param>
 <param-name>suppressStackTrace</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.
</filter-class>
<init-param>
 <param-name>suppressStackTrace</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
   maxEntriesLocalHeap="10"
   eternal="false"
   timeToIdleSeconds="5"
   timeToLiveSeconds="10"
   overflowToDisk="true"
   />
  <!-- Page and Page Fragment Caches -->
<cache name="CachePage1CachingFilter"
  maxEntriesLocalHeap="10"
  eternal="false"
  timeToIdleSeconds="10000"
  timeToLiveSeconds="10000"
  overflowToDisk="true">
</cache>
<cache name="CachedPage2Cache"
  maxEntriesLocalHeap="10"
  eternal="false"
  timeToLiveSeconds="3600"
  overflowToDisk="true">
</cache>
<cache name="SimplePageFragmentCachingFilter"
  maxEntriesLocalHeap="10"
  eternal="false"
  timeToIdleSeconds="10000"
  timeToLiveSeconds="10000"
  overflowToDisk="true">
</cache>
```

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

CachingFilter Exceptions

Additional exception types have been added to the Caching Filter.

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.

ResponseHeadersNotModifiableException

Same as FilterNonReentrantException.

AlreadyGzippedException

This exception is thrown when a gzip is attempted on already gzipped content.

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 unreadable content or a blank page.

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.

Hibernate Overview

Accelerating Hibernate applications typically involves reducing their reliance on the database when fetching data. Terracotta offers powerful in-memory solutions for maximizing the performance of Hibernate applications:

- Ehcache as a plug-in second-level cache for Hibernate – Automatically cache common queries in memory to substantially lower latency.
- BigMemory for an in-memory store – Leverage off-heap physical memory to keep more of the data set close to your application and out of reach of Java garbage collection.
- Automatic Resource Control for intelligent caching – Pin the hot set in memory for high-speed access and employ fine-grained sizing controls to avoid OutOfMemory errors.

The following sections provide a documentation Table of Contents and important information on using Ehcache with Hibernate.

Hibernate Table of Contents

Topic	Description
Hibernate Second-Level Cache	Ehcache easily integrates with the Hibernate Object/Relational persistence and query service. This page should be your first stop for configuration information, performance tips, and FAQs.
JMX Management and Monitoring	JMX monitoring is often used for Hibernate replicated caching. This page contains a section on Hibernate Statistics.
Grails	Includes recipes and code samples for using Ehcache with Hibernate and Grails.

Important Notices - PLEASE READ

Users of Ehcache and/or Terracotta Ehcache for Hibernate prior to Ehcache 2.0 should read Upgrade Notes for Ehcache versions prior to 2.0. These instructions are for Hibernate 3.

For older instructions on how to use Hibernate 2.1, please refer to Guide for Version 1.1.

Additional Information about Hibernate

The following pages provide additional information about using Ehcache with Hibernate:

- General Ehcache FAQ
- Transactions FAQ

Integrations Overview

The following sections provide a documentation Table of Contents and additional information sources about integrating Ehcache.

Integrations Table of Contents

Topics	Description
ColdFusion	ColdFusion ships with Ehcache. This page covers integrating ColdFusion versions 9, 9.0.1, and 8 with Ehcache.
Spring Caching	Ehcache simplifies caching in Spring. This page covers integrating Spring with Ehcache. Additional information may be found in Recipes.
Hibernate Caching	Ehcache easily integrates with the Hibernate Object/Relational persistence and query service. This page provides everything you need to configure Ehcache for Hibernate, and it includes performance tips and FAQs.
JRuby and Rails	ruby-ehcache is a JRuby Ehcache library which makes a commonly used subset of Ehcache's API available to JRuby. All of the strength of Ehcache is there, including BigMemory and the ability to cluster with Terracotta. It can be used directly via its own API, or as a Rails caching provider.
Google App Engine	The ehcache-googleappengine module combines the speed of Ehcache with the scale of Google's memcache. This page provides setup and troubleshooting information for configuring Google App Engine (GAE) caching.
Tomcat	Ehcache is probably used most commonly with Tomcat. This page documents some known issues with Tomcat, as well as recommended practices.
JDBC Caching	Ehcache can easily be combined with your existing JDBC code. Whether you access JDBC directly, or have a DAO/DAL layer, Ehcache can be combined with your existing data access pattern to speed up frequently accessed data to reduce page load times, improve performance, and reduce load from your database. This page discusses how to add caching to a JDBC application with the commonly used DAO/DAL layer patterns.
OpenJPA	Ehcache easily integrates with the OpenJPA persistence framework. This page provides installation and configuration information.
Grails	Grails 1.2RC1 and higher use Ehcache as the default Hibernate second level cache, and earlier versions of Grails ship with the Ehcache library and are very simple to enable. This page shows how to configure Grails to use Ehcache.
Glassfish	The maintainer uses Ehcache in production with Glassfish. This page is a how-to for working with Glassfish.
JSR107 (JCACHE)	JSR107 is currently being drafted and will continue to change until finalised. This support page has links to sites with information about JSR107 and its Ehcache implementation, JCACHE.

Additional Information about Integrating Ehcache

The following page provides additional information about integration:

- Spring Annotations

Using Coldfusion and Ehcache

Introduction

ColdFusion ships with Ehcache. This page covers integrating ColdFusion versions 9, 9.0.1, and 8 with Ehcache.

Which version of Ehcache comes with which version of ColdFusion?

The following versions of ColdFusion ship with Ehcache:

- ColdFusion 9.0.1 includes Ehcache 2.0 out-of-the-box
- ColdFusion 9 includes Ehcache 1.6 out-of-the-box
- ColdFusion 8 caching was not built on Ehcache, but Ehcache can easily be integrated with a CF8 application (see below).

Which version of Ehcache should I use if I want a distributed cache?

Ehcache is designed so that applications written to use it can easily scale out. A standalone cache (the default in ColdFusion 9) can easily be distributed. A distributed cache solves database bottleneck problems, cache drift (where the data cached in individual application server nodes becomes out of sync), and also (when using the recommended 2-tier Terracotta distributed cache) provides the ability to have a highly available, coherent in-memory cache that is far larger than can fit in any single JVM heap. See Getting Started for details.

Note: Ehcache 1.7 and higher support the Terracotta distributed cache out of the box. Due to Ehcache's API backward compatibility, it is easy to swap out older versions of ehcache with newer ones to leverage the features of new releases.

Using Ehcache with ColdFusion 9 and 9.0.1

The ColdFusion community has actively engaged with Ehcache and have put out lots of great blogs. Here are three to get you started. For a short introduction, see Raymond Camden's blog. For more in-depth analysis, see Rob Brooks-Bilson's nine-part Blog Series or 14 days of ColdFusion caching, by Aaron West, covering a different topic each day.

Switching from a local cache to a distributed cache with ColdFusion 9.0.1

1. Download the Terracotta kit. Click the link to the open-source kit if you are using open source and get `terracotta-<version>-installer.jar`.
2. Install the kit with `java -jar terracotta-<version>-installer.jar`. We will refer to the directory you installed it into as TCHOME. Similarly, we will refer to the location of ColdFusion as CFHOME. These instructions assume you are working with a standalone server install of ColdFusion; if working with a EAR/WAR install you will need to modify the steps accordingly (file

locations may vary and additional steps may be needed to rebuild the EAR/WAR).

Before integrating the distributed cache with ColdFusion, you may want to follow the simple self-contained tutorial which uses one of the samples in the kit to demonstrate distributed caching: http://www.terracotta.org/start/distributed-cache-tutorial

3. Copy TCHOME/ehcache/lib/ehcache-terracotta-\<version>.jar into CFHOME/lib
4. Edit the CFHOME/lib/ehcache.xml to add the necessary two lines of config to turn on distributed caching

```
<terracottaConfig url="localhost:9510"/>
<defaultCache
   ...
   >
           <terracotta clustered="true" />
</defaultCache>
```

5. Edit jvm.config (typically in CFHOME/runtime/bin) properties to ensure that coldfusion.classPath (set with -Dcoldfusion.classPath= in java.args) DOES NOT include any relative paths (ie ../) - ie replace the relative paths with full paths (This is to work around a known issue in ehcache-terracotta-2.0.0.jar).
6. Start the Terracotta server in a *NIX shell or Microsoft Windows:

```
$TCHOME/bin/start-tc-server.sh
start-tc-server.bat
```

Note: In production, you would run your servers on a set of separate machines for fault tolerance and performance.
7. Start ColdFusion, access your application, and see the distributed cache in action.
8. This is just the tip of the iceberg & you'll probably have lots of questions. Drop us an email to info@terracottatech.com to let us know how you got on, and if you have questions you'd like answers to.

Using Ehcache with ColdFusion 8

To integrate Ehcache with ColdFusion 8, first add the ehcache-core jar (and its dependent jars) to your web application lib directory. The following code demonstrates how to call Ehcache from ColdFusion 8. It will cache a CF object in Ehcache and the set expiration time to 30 seconds. If you refresh the page many times within 30 seconds, you will see the data from cache. After 30 seconds, you will see a cache miss, then the code will generate a new object and put in cache again.

```
<CFOBJECT type="JAVA" class="net.sf.ehcache.CacheManager" name="cacheManager">
<cfset cache=cacheManager.getInstance().getCache("MyBookCache")>
<cfset myBookElement=#cache.get("myBook")#>
<cfif IsDefined("myBookElement")>
   <cfoutput>
    myBookElement: #myBookElement#<br />
   </cfoutput>
   <cfif IsStruct(myBookElement.getObjectValue())>
           <strong>Cache Hit</strong><p/>
           <!-- Found the object from cache -->
           <cfset myBook = #myBookElement.getObjectValue()#>
   </cfif>
</cfif>
<cfif IsDefined("myBook")>
<cfelse>
<strong>Cache Miss</strong>
```

```
   <!-- object not found in cache, go ahead create it -->
   <cfset myBook = StructNew()>
   <cfset a = StructInsert(myBook, "cacheTime", LSTimeFormat(Now(), 'hh:mm:sstt'), 1)>
   <cfset a = StructInsert(myBook, "title", "EhCache Book", 1)>
   <cfset a = StructInsert(myBook, "author", "Greg Luck", 1)>
   <cfset a = StructInsert(myBook, "ISBN", "ABCD123456", 1)>
   <CFOBJECT type="JAVA" class="net.sf.ehcache.Element" name="myBookElement">
   <cfset myBookElement.init("myBook", myBook)>
   <cfset cache.put(myBookElement)>
</cfif>
<cfoutput>
Cache time: #myBook["cacheTime"]#<br />
Title: #myBook["title"]#<br />
Author: #myBook["author"]#<br />
ISBN: #myBook["ISBN"]#
</cfoutput>
```

Using Spring and Ehcache

Introduction

Ehcache has had excellent Spring integration for many years. This page demonstrates two new ways of using Ehcache with Spring.

Spring 3.1

Spring Framework 3.1 added a new generic cache abstraction for transparently applying caching to Spring applications. It adds caching support for classes and methods using two annotations:

@Cacheable

Cache a method call. In the following example, the value is the return type, a Manual. The key is extracted from the ISBN argument using the id.

```
@Cacheable(value="manual", key="#isbn.id")
public Manual findManual(ISBN isbn, boolean checkWarehouse)
```

@CacheEvict

Clears the cache when called.

```
@CacheEvict(value = "manuals", allEntries=true)
public void loadManuals(InputStream batch)
```

Spring 3.1 includes an Ehcache implementation. See the Spring 3.1 JavaDoc.

It also does much more with SpEL expressions. See http://blog.springsource.com/2011/02/23/spring-3-1-m1-caching/ for an excellent blog post covering this material in more detail.

Spring 2.5 - 3.1: Ehcache Annotations For Spring

This open source, led by Eric Dalquist, predates the Spring 3.1 project. You can use it with earlier versions of Spring or you can use it with 3.1.

@Cacheable

As with Spring 3.1 it uses an @Cacheable annotation to cache a method. In this example calls to findMessage are stored in a cache named "messageCache". The values are of type `Message`. The id for each entry is the `id` argument given.

```
@Cacheable(cacheName = "messageCache")
public Message findMessage(long id)
```

@TriggersRemove

And for cache invalidation, there is the @TriggersRemove annotation. In this example, `cache.removeAll()` is called after the method is invoked.

```
@TriggersRemove(cacheName = "messagesCache",
when = When.AFTER_METHOD_INVOCATION, removeAll = true)
public void addMessage(Message message)
```

See http://blog.goyello.com/2010/07/29/quick-start-with-ehcache-annotations-for-spring/ for a blog post explaining its use and providing further links.

Hibernate Second-Level Cache

IMPORTANT NOTICES - PLEASE READ

Users of Ehcache and/or Terracotta Ehcache for Hibernate prior to Ehcache 2.0 should read:

Upgrade Notes for Ehcache versions prior to 2.0. These instructions are for Hibernate 3.

For older instructions on how to use Hibernate 2.1, please refer to: Guide for Version 1.1

Introduction

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. Configuring Ehcache for Hibernate is simple. The basic steps are:

- Download and install Ehcache into your project
- Configure Ehcache as a cache provider in your project's Hibernate configuration.
- Configure second-level caching in your project's Hibernate configuration.
- Configure Hibernate caching for each entity, collection, or query you wish to cache.
- Configure ehcache.xml as necessary for each entity, collection, or query configured for caching.

For more information regarding cache configuration in Hibernate see the Hibernate documentation.

Downloading and Installing Ehcache

The Hibernate provider is in the ehcache-core module. Download the latest version of the Ehcache core module. For Terracotta clustering, download a full Ehcache distribution.

Maven

Dependency versions vary with the specific kit you intend to use. Since kits are guaranteed to contain compatible artifacts, find the artifact versions you need by downloading a kit. Configure or add the following repository to your build (pom.xml):

```
<repository>
   <id>terracotta-releases</id>
   <url>http://www.terracotta.org/download/reflector/releases</url>
   <releases><enabled>true</enabled></releases>
   <snapshots><enabled>false</enabled></snapshots>
</repository>
```

Configure or add the the ehcache core module defined by the following dependency to your build (pom.xml):

```
<dependency>
   <groupId>net.sf.ehcache</groupId>
   <artifactId>ehcache-core</artifactId>
   <version>${ehcacheVersion}</version>
</dependency>
```

Maven

If you are configuring Hibernate and Ehcache for Terracotta clustering, add the following dependencies to your build (pom.xml):

```
<dependency>
   <groupId>net.sf.ehcache</groupId>
   <artifactId>ehcache-terracotta</artifactId>
   <version>${ehcacheVersion}</version>
</dependency>
<dependency>
   <groupId>org.terracotta</groupId>
   <artifactId>terracotta-toolkit-${toolkitAPIversion}-runtime</artifactId>
   <version>${toolkitVersion}</version>
</dependency>
```

Configure Ehcache as the Second-Level Cache Provider

To configure Ehcache as a Hibernate second-level cache, set the region factory property (for Hibernate 3.3 and above) or the factory class property (Hibernate 3.2 and below) to one of the following in the Hibernate configuration. Hibernate configuration is configured either via hibernate.cfg.xml, hibernate.properties or Spring. The format given is for hibernate.cfg.xml.

Hibernate 3.3 and higher

ATTENTION HIBERNATE 3.2 USERS: Make sure to note the change to BOTH the property name and value.

Use:

```
<property name="hibernate.cache.region.factory_class">
         net.sf.ehcache.hibernate.EhCacheRegionFactory</property>
```

for instance creation, or

```
<property name="hibernate.cache.region.factory_class">
         net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory</property>
```

to force Hibernate to use a singleton of Ehcache CacheManager.

Hibernate 3.0 - 3.2

Use:

```
<property name="hibernate.cache.provider_class">
         net.sf.ehcache.hibernate.EhCacheProvider</property>
```

for instance creation, or

```
<property name="hibernate.cache.provider_class">
         net.sf.ehcache.hibernate.SingletonEhCacheProvider</property>
```

to force Hibernate to use a singleton Ehcache CacheManager.

Enable Second-Level Cache and Query Cache Settings

In addition to configuring the second-level cache provider setting, you will need to turn on the second-level cache (by default it is configured to off - 'false' - by Hibernate). This is done by setting the following property in your hibernate config:

```
<property name="hibernate.cache.use_second_level_cache">true</property>
```

You may also want to turn on the Hibernate query cache. This is done by setting the following property in your hibernate config:

```
<property name="hibernate.cache.use_query_cache">true</property>
```

Optional

The following settings or actions are optional.

Ehcache Configuration Resource Name

The `configurationResourceName` property is used to specify the location of the ehcache configuration file to be used with the given Hibernate instance and cache provider/region-factory. The resource is searched for in the root of the classpath. It is used to support multiple CacheManagers in the same VM. It tells Hibernate which configuration to use. An example might be "ehcache-2.xml". When using multiple Hibernate instances it is therefore recommended to use multiple non-singleton providers or region factories, each with a dedicated Ehcache configuration resource.

```
net.sf.ehcache.configurationResourceName=/name_of_ehcache.xml
```

Set the Hibernate cache provider programmatically

The provider can also be set programmatically in Hibernate by adding necessary Hibernate property settings to the configuration before creating the SessionFactory:

```
Configuration.setProperty("hibernate.cache.region.factory_class",
                     "net.sf.ehcache.hibernate.EhCacheRegionFactory")
```

Putting it all together

If you are using Hibernate 3.3 and enabling both second-level caching and query caching, then your hibernate config file should contain the following:

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
<property name="hibernate.cache.region.factory_class">net.sf.ehcache.hibernate.EhCacheRegionFacto
```

An equivalent Spring configuration file would contain:

```
<prop key="hibernate.cache.use_second_level_cache">true</prop>
<prop key="hibernate.cache.use_query_cache">true</prop>
<prop key="hibernate.cache.region.factory_class">net.sf.ehcache.hibernate.EhCacheRegionFactory</p
```

Configure Hibernate Entities to use Second-Level Caching

In addition to configuring the Hibernate second-level cache provider, Hibernate must also be told to enable caching for entities, collections, and queries. 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"
>
...
</class>
</hibernate-mapping>
```

To enable caching, add the following element.

```
<cache usage="read-write|nonstrict-read-write|read-only" />
```

For example:

```
<hibernate-mapping>
<class
name="com.somecompany.someproject.domain.Country"
table="ut_Countries"
dynamic-update="false"
dynamic-insert="false"
>
 <cache usage="read-write" />
...
</class>
</hibernate-mapping>
```

This can also be achieved using the @Cache annotation, e.g.

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.READ_WRITE)
public class Country {
...
}
```

Definition of the different cache strategies

read-only

Caches data that is never updated.

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!

Definition of the different cache strategies

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.

Configure

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 maxEntriesLocalHeap and specifying whether to overflowToDisk beyond that. Hibernate uses a specific convention for the naming of caches of Domain Objects, Collections, and Queries.

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.

```
   <?xml version="1.0" encoding="UTF-8"?>
<ehcache>
 <cache
name="com.somecompany.someproject.domain.Country"
maxEntriesLocalHeap="10000"
eternal="false"
timeToIdleSeconds="300"
timeToLiveSeconds="600"
overflowToDisk="true"
 />
</ehcache>
```

Hibernate CacheConcurrencyStrategy

read-write, nonstrict-read-write and read-only policies apply to Domain Objects.

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;
}
```

Collections

You need an additional cache configured for the set. The ehcache.xml configuration looks like:

```
   <?xml version="1.0" encoding="UTF-8"?>
<ehcache>
 <cache name="com.somecompany.someproject.domain.Country"
maxEntriesLocalHeap="50"
eternal="false"
timeToLiveSeconds="600"
overflowToDisk="true"
/>
 <cache
name="com.somecompany.someproject.domain.Country.advancedSearchFacilities"
maxEntriesLocalHeap="450"
eternal="false"
timeToLiveSeconds="600"
overflowToDisk="true"
/>
</ehcache>
```

Hibernate CacheConcurrencyStrategy

read-write, nonstrict-read-write and read-only policies apply to Domain Object collections.

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.

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"
maxEntriesLocalHeap="5"
eternal="false"
timeToLiveSeconds="120"
overflowToDisk="true"/>
```

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"
maxEntriesLocalHeap="5000"
eternal="true"
overflowToDisk="true"/>
```

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"
maxEntriesLocalHeap="5"
eternal="false"
timeToLiveSeconds="86400"
overflowToDisk="true"/>
```

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. Alex Miller has a good article on the query cache here.

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.

Demo Apps

We have demo applications showing how to use the Hibernate 3.3 CacheRegionFactory.

Hibernate Tutorial

Check out from the Terracotta Forge.

Examinator

Examinator is our complete application that shows many aspects of caching, all using the Terracotta Server Array. Check out from the Terracotta Forge.

Performance Tips

Session.load

`Session.load` will always try to use the cache.

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.

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.

How to Scale

Configuring each Hibernate instance with a standalone ehcache will dramatically improve performance. With an application deployed on multiple nodes, using standalone Ehcache means that each instance holds its own (unshared) data. When data is written in one node, the other nodes are unaware of the data write, and thus subsequent reads of this data on other nodes will result in stale reads. On a cache miss on any node, Hibernate will read from the database, which generally results in N reads where N is the number of nodes in the cluster. With each new node, the database's workload goes up.

Most production applications use multiple application instances for redundancy and for scalability, which requires applications to be horizontally scalable because adding more application instances linearly improves throughput. The solution is to turn on distributed caching or replicated caching.

Ehcache comes with native cache distribution using the following mechanism:

- Terracotta Server Array

Ehcache supports the following methods of cache replication:

- RMI
- JGroups
- JMS replication

Selection of the distributed cache or replication mechanism may be made or changed at any time. There are no changes to the application. Only changes to ehcache.xml file are required. This allows an application to easily scale as it grows without expensive re-architecting.

Using Distributed Ehcache

Ehcache provides built-in support for Terracotta distributed caching, providing the following advantages:

- Simple snap-in configuration with one line of configuration
- Practically unlimited scale with BigMemory and the Terracotta Server Array
- Wealth of "CAP" configuration options allow you to configure your cache for whatever it needs – speed, consistency, availability, asynchronous updates, dirty reads, and more
- Automatic Resource Control (ARC) for automatically maintaining cache sizes

Configuring Terracotta distributed caching for Hibernate is described in the Terracotta Documentation. A sample cache configuration is provided here:

```
   <?xml version="1.0" encoding="UTF-8"?>
<ehcache>
 <terracottaConfig url="localhost:9510" />
 <cache
name="com.somecompany.someproject.domain.Country"
maxEntriesLocalHeap="10000"
maxBytesLocalOffHeap="10G"
eternal="false"
timeToIdleSeconds="300"
timeToLiveSeconds="600"
overflowToDisk="true">
<terracotta/>
 </cache>
</ehcache>
```

Configuring Replicated Caching using RMI, JGroups, or JMS

Ehcache can use JMS, JGroups or RMI as a cache replication scheme. The following are the key considerations when selecting this option:

- The consistency is weak. Nodes might be stale, have different versions or be missing an element that other nodes have. Your application should be tolerant of weak consistency.
- `session.refresh()` should be used to check the cache against the database before performing a write that must be correct. This can have a performance impact on the database.
- Each node in the cluster stores all data, thus the cache size is limited to memory size, or disk if disk overflow is selected.

Configuring for RMI Replication

RMI configuration is described in the Ehcache User Guide - RMI Distributed Caching. A sample cache configuration (using automatic discovery) is provided here:

```
   <?xml version="1.0" encoding="UTF-8"?>
<ehcache>
 <cacheManagerPeerProviderFactory
  class="net.sf.ehcache.distribution.RMICacheManagerPeerProviderFactory"
properties="peerDiscovery=automatic, multicastGroupAddress=230.0.0.1,
  multicastGroupPort=4446, timeToLive=32"/>
 <cache
```

```
name="com.somecompany.someproject.domain.Country"
maxEntriesLocalHeap="10000"
eternal="false"
timeToIdleSeconds="300"
timeToLiveSeconds="600"
overflowToDisk="true">
<cacheEventListenerFactory
       class="net.sf.ehcache.distribution.RMICacheReplicatorFactory"/>
 </cache>
</ehcache>
```

Configuring for JGroups Replication

Configriung JGroups replication is described in the Ehcache User Guide - Distributed Caching with JGroups. A sample cache configuration is provided here:

```
   <?xml version="1.0" encoding="UTF-8"?>
<ehcache>
 <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="com.somecompany.someproject.domain.Country"
maxEntriesLocalHeap="10000"
eternal="false"
timeToIdleSeconds="300"
timeToLiveSeconds="600"
overflowToDisk="true">
<cacheEventListenerFactory
class="net.sf.ehcache.distribution.jgroups.JGroupsCacheReplicatorFactory"
properties="replicateAsynchronously=true, replicatePuts=true,
replicateUpdates=true, replicateUpdatesViaCopy=false,
replicateRemovals=true" />
 </cache>
</ehcache>
```

Configuring for JMS Replication

Configuring JMS replication is described in the Ehcache User Guide - JMS Distributed Caching. A sample cache configuration (for ActiveMQ) is provided here:

```
   <?xml version="1.0" encoding="UTF-8"?>
<ehcache>
 <cacheManagerPeerProviderFactory
       class="net.sf.ehcache.distribution.jms.JMSCacheManagerPeerProviderFactory"
       properties="initialContextFactoryName=ExampleActiveMQInitialContextFactory,
           providerURL=tcp://localhost:61616,
           topicConnectionFactoryBindingName=topicConnectionFactory,
           topicBindingName=ehcache"
       propertySeparator=","
       />
 <cache
name="com.somecompany.someproject.domain.Country"
maxEntriesLocalHeap="10000"
eternal="false"
timeToIdleSeconds="300"
timeToLiveSeconds="600"
overflowToDisk="true">
<cacheEventListenerFactory
     class="net.sf.ehcache.distribution.jms.JMSCacheReplicatorFactory"
     properties="replicateAsynchronously=true,
                  replicatePuts=true,
                  replicateUpdates=true,
                  replicateUpdatesViaCopy=true,
                  replicateRemovals=true,
                  asynchronousReplicationIntervalMillis=1000"
      propertySeparator=","/>
 </cache>
</ehcache>
```

FAQ

If I'm using Ehcache with my app and with Hibernate for second-level caching, should I try to use the CacheManager created by Hibernate for my app's caches?

While you could share the resource file between the two CacheManagers, a clear separation between the two is recommended. Your app may have a different lifecycle than Hibernate, and in each case your CacheManager ARC settings may need to be different.

Should I use the provider in the Hibernate distribution or in Ehcache?

Since Hibernate 2.1, Hibernate has included an Ehcache `CacheProvider`. That provider is periodically synced up with the provider in the Ehcache Core distribution. New features are generally added in to the Ehcache Core provider and then the Hibernate one.

What is the relationship between the Hibernate and Ehcache projects?

Gavin King and Greg Luck cooperated to create Ehcache and include it in Hibernate. Since 2009 Greg Luck has been a committer on the Hibernate project so as to ensure Ehcache remains a first-class 2nd level cache for Hibernate.

Does Ehcache support the new Hibernate 3.3 2nd level caching SPI?

Yes. Ehcache 2.0 supports this new API.

Does Ehcache support the transactional strategy?

Yes. It was introduced in Ehcache 2.1.

Why do certain caches sometimes get automatically cleared by Hibernate?

Whenever a `Query.executeUpdate()` is run, for example, Hibernate invalidates affected cache regions (those corresponding to affected database tables) to ensure that no data stale data is cached. This should also happen whenever stored procedures are executed.

For more information, see this Hibernate bug report.

Is Ehcache Cluster Safe?

hibernate.org maintains a table listing the providers. While ehcache works as a distributed cache for Hibernate, it is not listed as "Cluster Safe". What this means is that `Hibernate's lock and unlock methods are not implemented. Changes in one node will be applied without locking. This may or may not be a noticeable problem. In Ehcache 1.7 when using Terracotta, this cannot happen as access to the clustered cache itself is controlled with read locks and write locks. In Ehcache 2.0 when using Terracotta, the lock and unlock methods tie-in to the underlying clustered cache locks. We expect Ehcache 2.0 to be marked as cluster safe in new versions of the Hibernate documentation.

How are Hibernate Entities keyed?

Hibernate identifies cached Entities via an object id. This is normally the primary key of a database row.

Can you use Identity mode with the Terracotta Server Array

You cannot use identity mode clustered cache with Hibernate. If the cache is exclusively used by Hibernate we will convert identity mode caches to serialization mode. If the cache cannot be determined to be exclusively used by Hibernate (i.e. generated from a singleton cache manager) then an exception will be thrown indicating the misconfigured cache. Serialization mode is in any case the default for Terracotta clustered caches.

I get `org.hibernate.cache.ReadWriteCache - An item was expired by the cache while it was locked` error messages. What is it?

Soft locks are implemented by replacing a value with a special type that marks the element as locked, thus indicating to other threads to treat it differently to a normal element. This is used in the Hibernate Read/Write strategy to force fall-through to the database during the two-phase commit - since we don't know exactly what should be returned by the cache while the commit is in process (but the db does). If a soft-locked Element is evicted by the cache during the 2 phase commit, then once the 2 phase commit completes the cache will fail to update (since the soft-locked Element was evicted) and the cache entry will be reloaded from the database on the next read of that object. This is obviously non-fatal (we're a cache failure here so it should not be a problem). The only problem it really causes would I imagine be a small rise in db load. So, in summary the Hibernate messages are not problematic. The underlying cause is the probabilistic evictor can theoretically evict recently loaded items. This evictor has been tuned over successive ehcache releases. As a result this warning will happen most often in 1.6, less often in 1.7 and very rarely in 1.8. You can also use the deterministic evictor to avoid this problem. Specify the `java -Dnet.sf.ehcache.use.classic.lru=true` system property to turn on classic LRU which contains a deterministic evictor.

I get java.lang.ClassCastException: org.hibernate.cache.ReadWriteCache$Item incompatible with net.sf.ehcache.hibernate.strategy.AbstractReadWriteEhcacheAccessStrategy$

This is the tell-tale error you get if you are:

- using a Terracotta cluster with Ehcache
- you have upgraded part of the cluster to use net.sf.ehcache.hibernate.EhCacheRegionFactory but part of it is still using the old SPI of EhCacheProvider.
- you are upgrading a Hibernate version Ensure you have changed all nodes and that you clear any caches during the upgrade.

Are compound keys supported?

For standalone caching yes. With Terracotta a larger amount of memory is used.

Why do I not see replicated data when using nonstrict mode?

You may think that Hibernate's <nonstrict> mode is just like <read-write> but with dirty reads. The truth is far more complex than that. Suffice to say, in <nonstrict> mode, Hibernate puts the object in the appropriate cache but then IMMEDIATELY removes it. The PUT and the REMOVE are BOTH replicated by ehcache so the net effect of that is the new object is copied to remote cache but then it's immediately followed by a replicated remove so the next time you try get the object it's not in cache and hibernate goes back to the DB. So, practically there is no point using nonstrict mode with replicated or distributed caches. If you want the updated entry to be replicated or distributed use <readwrite> or <transactional>.

JRuby and Rails Caching

Introduction

jruby-ehcache is a JRuby Ehcache library which makes a commonly used subset of Ehcache's API available to JRuby. All of the strength of Ehcache is there, including BigMemory and the ability to cluster with Terracotta. It can be used directly via its own API, or as a Rails caching provider.

Installation

Installation for JRuby

Ehcache JRuby integration is provided by the jruby-ehcache gem. To install it, simply execute:

```
jgem install jruby-ehcache
```

Note that you may need to use "sudo" to install gems on your system.

Installation for Rails

If you want Rails caching support, you should also install the correct gem for your Rails version:

```
jgem install jruby-ehcache-rails2 # for Rails 2
jgem install jruby-ehcache-rails3 # for Rails 3
```

An alternative installation is to simply add the appropriate jruby-ehcache-rails dependency to your Gemfile, and then run a Bundle Install. This will pull in the latest jruby-ehcache gem.

Dependencies

- JRuby 1.5 and higher
- Rails 2 for the jruby-ehcache-rails2
- Rails 3 for the jruby-ehcache-rails3
- Ehcache 2.4.6 is the declared dependency, although any version of Ehcache will work.

The jruby-ehcache gem comes bundled with the ehcache-core.jar. To use a different version of Ehcache, place the Ehcache jar in the same Classpath as JRuby (for standalone JRuby) or in the Rails lib directory (for Rails).

Configuring Ehcache

Configuring Ehcache for JRuby is done using the same ehcache.xml file as used for native Java Ehcache. The ehcache.xml file can be placed either in your Classpath or, alternatively, can be placed in the same directory as the Ruby file in which you create the CacheManager object from your Ruby code. For a Rails application, the ehcache.xml file should reside in the config directory of the Rails application.

Using the jruby-ehcache API directly

Basic Operations

To make Ehcache available to JRuby:

```
require 'ehcache'
```

Note that, because jruby-ehcache is provided as a Ruby Gem, you must make your Ruby interpreter aware of Ruby Gems in order to load it. You can do this by either including -rubygems on your jruby command line, or you can make Ruby Gems available to JRuby globally by setting the RUBYOPT environment variable as follows:

```
export RUBYOPT=rubygems
```

To create a CacheManager, which you do once when the application starts:

```
manager = Ehcache::CacheManager.new
```

To access an existing cache (call it "sampleCache1"):

```
cache = manager.cache("sampleCache1")
```

To create a new cache from the defaultCache:

```
cache = manager.cache
```

To put into a cache:

```
cache.put("key", "value", {:ttl => 120})
```

To get from a cache:

```
cache.get("key")  # Returns the Ehcache Element object
cache["key"]      # Returns the value of the element directly
```

To shut down the CacheManager: This is only when you shut your application down. It is only necessary to call this if the cache is `diskPersistent` or is clustered with Terracotta, but it is always a good idea to do it.

```
manager.shutdown
```

Supported Properties

The following caching options are supported in JRuby:

PropertyArgument TypeDescription unlessExist, ifAbsentboolean

If true, use the putIfAbsent method. elementEvictionDataElementEvictionData

Sets this elementâ— s eviction data instance.

Supported Properties

eternalboolean

Sets whether the element is eternal.

timeToIdle, ttiint

Sets time to idle.

timeToLive, ttl, expiresInint

Sets time to Live.

versionlong

Sets the version attribute of the ElementAttributes object.

Example Configuration

```
class SimpleEhcache
 #Code here
 require 'ehcache'
 manager = Ehcache::CacheManager.new
 cache = manager.cache
 cache.put("answer", "42", {:ttl => 120})
 answer = cache.get("answer")
 puts "Answer: #{answer.value}"
 question = cache["question"] || 'unknown'
 puts "Question: #{question}"
 manager.shutdown
end
```

As you can see from the example, you create a cache using CacheManager.new, and you can control the "time to live" value of a cache entry using the :ttl option in cache.put.

Using Ehcache from within Rails

General Overview

The ehcache.xml file

Configuration of Ehcache is still done with the ehcache.xml file, but for Rails applications you must place this file in the config directory of your Rails app. Also note that you must use JRuby to execute your Rails application, as these gems utilize JRuby's Java integration to call the Ehcache API. With this configuration out of the way, you can now use the Ehcache API directly from your Rails controllers and/or models. You could of course create a new Cache object everywhere you want to use it, but it is better to create a single instance and make it globally accessible by creating the Cache object in your Rails environment.rb file. For example, you could add the following lines to config/environment.rb:

```
require 'ehcache'
EHCACHE = Ehcache::CacheManager.new.cache
```

By doing so, you make the EHCACHE constant available to all Rails-managed objects in your application. Using the Ehcache API is now just like the above JRuby example. If you are using Rails 3 then you have a better option at your disposal: the built-in Rails 3 caching API. This API provides an abstraction layer for caching underneath which you can plug in any one of a number of caching providers. The most common provider to date has been the memcached provider, but now you can also use the Ehcache provider. Switching to the Ehcache provider requires only one line of code in your Rails environment file (e.g. development.rb or production.rb):

```
config.cache_store = :ehcache_store, {
                        :cache_name => 'rails_cache',
                        :ehcache_config => 'ehcache.xml'
                    }
```

This configuration will cause the Rails.cache API to use Ehcache as its cache store. The :cache_name and :ehcache_config are both optional parameters, the default values for which are shown in the above example. The value of the :ehcache_config parameter can be either an absolute path or a relative path, in which case it is interpreted relative to the Rails app's config directory. A very simple example of the Rails caching API is as follows:

```
Rails.cache.write("answer", "42")
Rails.cache.read("answer")  # => '42'
```

Using this API, your code can be agnostic about the underlying provider, or even switch providers based on the current environment (e.g. memcached in development mode, Ehcache in production). The write method also supports options in the form of a Hash passed as the final parameter.

See the Supported Properties table above for the options that are supported. These options are passed to the write method as Hash options using either camelCase or underscore notation, as in the following example:

```
Rails.cache.write('key', 'value', :time_to_idle => 60.seconds, :timeToLive => 600.seconds)
caches_action :index, :expires_in => 60.seconds, :unless_exist => true
```

Turn on caching in your controllers

You can also configure Rails to use Ehcache for its automatic action caching and fragment caching, which is the most common method for caching at the controller level. To enable this, you must configure Rails to perform controller caching, and then set Ehcache as the provider in the same way as for the Rails cache API:

```
config.action_controller.perform_caching = true
config.action_controller.cache_store = :ehcache_store
```

Setting up a Rails Application with Ehcache

Here are the basic steps for configuring a Rails application to use Ehcache:

1. For this example, we will create a new Rails application with the custom template from JRuby.org. The following command creates a "rails-bigmemory" application:

   ```
   jruby -S rails new rails-bigmemory -m http://jruby.org/rails3.rb
   ```
2. The example application will be a simple address book. Generate a scaffold for the address book application, which will create contacts including a first name, last name, and email address.

   ```
   jruby -S rails generate scaffold Contact first_name: string last_name: string email_addres
   ```

3. Add support for caching with Ehcache. There are several ways to do this, but for this example, we will use the Action Controller caching mechanism. Open the ContactsController.rb. Add a call to the Action method to tell it to cache the results of our index and show pages.

   ```
   caches_action :index, :show
   ```

   To expire items from the cache as appropriate, add calls to expire the results of the caching calls.

   Under create, add the following:

   ```
   expire_action :action => 'index'
   ```

   Under update, add the following:

   ```
   expire_action :action => 'show', :id => params[:id]
    expire_action :action => 'index'
   ```

   Under destroy, add the following:

   ```
    expire_action :action => 'index'
   ```
4. Now that the application is configured to support caching, specify Ehcache as its caching provider. Open the Gemfile and declare a dependency on the ehcache-jruby gem. Add the following line:

   ```
   gem 'ehcache-jruby-rails3'
   ```
5. In the development.rb file, enable caching for the Rails Action Controller mechanism, which is disabled by default in developement mode. (Note that caching must be configured for each environment in which you want to use it.) This file also needs a specification for using Ehcache as the cache store provider. Add the following two lines to the .rb file:

   ```
   config.action_controller.perform_caching = true
    config.cache_store = :ehcache_store
   ```
6. Run the Bundle Install command.

   ```
   jruby -S bundle install
   ```
7. Run the Rake command to create the database and populate the initial schema.

   ```
   jruby -S rake db:create db:migrate
   ```
8. (Optional) Set up the Ehcache monitor. This involves the following four steps:

   - Install the Ehcache Monitor from Downloads.
   - Start the Ehcache Monitor server.
   - Connect the application to the monitor server by copying the ehcache-probe JAR (bundled with the Ehcache Monitor) to your Rails lib directory.
   - Create an ehcache.xml file in the Rails application config directory. In the ehcache.xml file, add the following:

     ```
     <cacheManagerPeerListenerFactory
         class="org.terracotta.ehcachedx.monitor.probe.ProbePeerListenerFactory"
         properties="monitorAddress=localhost, monitorPort=9889, memoryMeasurement=true"/
     ```

Now you are ready to start the application with the following command:

```
jruby -S rails server
```

Once the application is started, populate the cache by adding, editing, and deleting contacts. To see the Contacts address book, enter the following in your browser:

```
http://localhost:3000/contacts
```

To view cache activity and statistics in the Ehcache monitor, enter the following in your browser:

```
http://localhost:9889/monitor
```

For more information about how to use the monitor, refer to the Ehcache Monitor page.

Adding BigMemory under Rails

BigMemory provides in-memory data management with a large additional cache located right at the node where your application runs. To upgrade your Ehcache to use BigMemory with your Rails application, follow these steps.

1. Add the ehcache-core-ee.jar to your Rails application lib directory.
2. Modify the ehcache.xml file (in the config directory of your Rails application) by adding the following to each cache where you want to enable BigMemory:

   ```
   overflowToOffHeap="true"
   maxBytesLocalOffHeap="1G"
   ```

   When `overflowToOffHeap` is set to true, it enables the cache to utilize off-heap memory storage to improve performance. Off-heap memory is not subject to Java GC cycles and has a size limit set by the Java property MaxDirectMemorySize.

   `maxBytesLocalOffHeap` sets the amount of off-heap memory available to the cache, and is in effect only if overflowToOffHeap is true. For more information about sizing caches, refer to How To Size Caches.
3. Also in the ehcache.xml file, set `maxEntriesLocalHeap` to at least 100 elements when using an off-heap store to avoid performance degradation. Lower values for `maxEntriesLocalHeap` trigger a warning to be logged.
4. Now that your application is configured to use BigMemory, start it with the following commands:

   ```
   jruby -J-Dcom.tc.productkey.path=/path/to/key -J-XX:MaxDirectMemorySize=2G -S rails server
   ```

   This will configure a system property that points to the location of the license key, and it will set the direct memory size. The `maxDirectMemorySize` must be at least 256M larger than total off-heap memory (the unused portion will still be available for other uses).

For additional configuration options, refer to the BigMemory page.

Note that only Serializable cache keys and values can be placed in the store, similar to DiskStore. Serialization and deserialization take place on putting and getting from the store. This is handled automatically by the jruby-ehcache gem.

Google App Engine (GAE) Caching

Introduction

The ehcache-googleappengine module combines the speed of Ehcache with the scale of Google's memcache and provide the best of both worlds:

- Speed - Ehcache cache operations take a few microseconds, versus around 60ms for Google's provided client-server cache, memcacheg.
- Cost - Because it uses way less resources, it is also cheaper.
- Object Storage - Ehcache in-process cache works with Objects that are not Serializable.

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.

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 this blog for the evidence of this).

Dependencies

Version 2.3 and higher of Ehcache are compatible with Google App Engine. Older versions will not work.

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
- there is no Terracotta configuration

Use following Ehcache configuration 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
       maxEntriesOnHeap="10000"
       eternal="false"
       timeToIdleSeconds="120"
       timeToLiveSeconds="120"
       overflowToDisk="false"
       diskPersistent="false"
       memoryStoreEvictionPolicy="LRU"
       />
<!--Example sample cache-->
    <cache name="sampleCache1"
      maxEntriesOnHeap="10000"
      maxEntriesLocalDisk="1000"
      eternal="false"
      timeToIdleSeconds="300"
      timeToLiveSeconds="600"
      memoryStoreEvictionPolicy="LFU"
       />
</ehcache>
```

Recipes

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 Google App Engine. You get the benefits of the 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.

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.

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.

Dynamic Web Content Caching

Google App Engine provides acceleration for files declared static in `appengine-web.xml`.

For example:

```
<static-files>
  <include path="/**.png" />
  <exclude path="/data/**.png" />
</static-files>
```

You can get acceleration for dynamic files using Ehcache's caching filters as you usually would. See Web

Caching for more information.

Troubleshooting

NoClassDefFoundError

If you get the error `java.lang.NoClassDefFoundError: java.rmi.server.UID is a restricted class` then you are using a version of Ehcache prior to 1.6.

Sample application

The easiest way to get started is to play with a simple sample app. We provide a simple Rails application which stores an integer value in a cache along with increment and decrement operations. The sample app shows you how to use ehcache as a caching plugin and how to use it directly from the Rails caching API.

Tomcat Issues and Best Practices

Introduction

Ehcache is probably used most commonly with Tomcat. This page 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.

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 unmar
```

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.

Class-loader memory leak

In development, there appears 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`.

RMI CacheException - problem starting listener for RMICachePeer

I get the following error:

```
net.sf.ehcache.CacheException: Problem starting listener for RMICachePeer ... java.rmi.UnmarshalE
```

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 this explanation. The workaround is to remove the spaces in your Tomcat installation path.

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 it is Tomcat-specific.

JDBC Caching

Introduction

Ehcache can easily be combined with your existing JDBC code. Whether you access JDBC directly, or have a DAO/DAL layer, Ehcache can be combined with your existing data access pattern to speed up frequently accessed data to reduce page load times, improve performance, and reduce load from your database.

This page discusses how to add caching to a JDBC application using the commonly used DAO/DAL layer patterns.

Adding JDBC caching to a DAO/DAL layer

If your application already has a DAO/DAL layer, this is a natural place to add caching. To add caching, follow these steps:

- identify methods which can be cached
- instantiate a cache and add a member variable to your DAO to hold a reference to it
- Put and get values from the cache

Identifying methods which can be cached

Normally, you will want to cache the following kinds of method calls:

- Any method which retrieves entities by an Id
- Any queries which can be tolerate some inconsistent or out of date data

Example methods that are commonly cacheable:

```
public V getById(final K id);
public Collection findXXX(...);
```

Instantiate a cache and add a member variable

Your DAO is probably already being managed by Spring or Guice, so simply add a setter method to your DAO layer such as `setCache(Cache cache)`. Configure the cache as a bean in your Spring or Guice config, and then use the the frameworks injection methodology to inject an instance of the cache.

If you are not using a DI framework such as Spring or Guice, then you will need to instantiate the cache during the bootstrap of your application. As your DAO layer is being instantiated, pass the cache instance to it.

Put and get values from the cache

Now that your DAO layer has a cache reference, you can start to use it. You will want to consider using the cache using one of two standard cache access patterns:

- cache-aside

Put and get values from the cache

- cache-as-sor

You can read more about these in the Concepts cache-as-sor and Concepts cache-aside sections.

Putting it all together - an example

Here is some example code that demonstrates a DAO based cache using a cache aside methodology wiring it together with Spring.

This code implements a PetDao modeled after the Spring Framework PetClinic sample application.

It implements a standard pattern of creating an abstract GenericDao implementation which all Dao implementations will extend.

It also uses Spring's SimpleJdbcTemplate to make the job of accessing the database easier.

Finally, to make Ehcache easier to work with in Spring, it implements a wrapper that holds the cache name.

The example files

The following are relevant snippets from the example files. A full project will be available shortly.

CacheWrapper.java

Simple get/put wrapper interface.

```
public interface CacheWrapper<K, V>
{
 void put(K key, V value);
 V get(K key);
}
```

EhcacheWrapper.java

The wrapper implementation. Holds the name so caches can be named.

```
public class EhCacheWrapper<K, V> implements CacheWrapper<K, V>
{
    private final String cacheName;
    private final CacheManager cacheManager;
    public EhCacheWrapper(final String cacheName, final CacheManager cacheManager)
    {
    this.cacheName = cacheName;
    this.cacheManager = cacheManager;
    }
    public void put(final K key, final V value)
    {
    getCache().put(new Element(key, value));
    }
    public V get(final K key, CacheEntryAdapter<V> adapter)
    {
    Element element = getCache().get(key);
    if (element != null) {
        return (V) element.getValue();
    }
    return null;
    }
    public Ehcache getCache()
    {
    return cacheManager.getEhcache(cacheName);
    }
}
```

GenericDao.java

The Generic Dao. It implements most of the work.

```
public abstract class GenericDao<K, V extends BaseEntity> implements Dao<K, V>
{
    protected DataSource datasource;
    protected SimpleJdbcTemplate jdbcTemplate;
    /* Here is the cache reference */
    protected CacheWrapper<K, V> cache;
    public void setJdbcTemplate(final SimpleJdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
    public void setDatasource(final DataSource datasource) {
        this.datasource = datasource;
    }
    public void setCache(final CacheWrapper<K, V> cache) {
        this.cache = cache;
    }
    /* the cacheable method */
    public V getById(final K id) {
    V value;
    if ((value = cache.get(id)) == null) {
        value = this.jdbcTemplate.queryForObject(findById, mapper, id);
        if (value != null) {
        cache.put(id, value);
        }
    }
    return value;
    }
    /** rest of GenericDao implementation here **/
    /** ... **/
    /** ... **/
    /** ... **/
}
```

PetDaoImpl.java

The Pet Dao implementation, really it doesn't need to do anything unless specific methods not available via GenericDao are cacheable.

```
public class PetDaoImpl extends GenericDao<Integer, Pet> implements PetDao
{
/** ... **/
}
```

We need to configure the JdbcTemplate, Datasource, CacheManager, PetDao, and the Pet cache using the spring configuration file.

application.xml

The Spring configuration file.

```
<!-- datasource and friends -->
<bean id="dataSource" class="org.springframework.jdbc.datasource.FasterLazyConnectionDataSourcePr
  <property name="targetDataSource" ref="dataSourceTarget"/>
</bean>
<bean id="dataSourceTarget" class="com.mchange.v2.c3p0.ComboPooledDataSource"
      destroy-method="close">
  <property name="user" value="${jdbc.username}"/>
  <property name="password" value="${jdbc.password}"/>
  <property name="driverClass" value="${jdbc.driverClassName}"/>
  <property name="jdbcUrl" value="${jdbc.url}"/>
  <property name="initialPoolSize" value="50"/>
  <property name="maxPoolSize" value="300"/>
  <property name="minPoolSize" value="30"/>
  <property name="acquireIncrement" value="2"/>
  <property name="acquireRetryAttempts" value="0"/>
</bean>
<!-- jdbctemplate -->
<bean id="jdbcTemplate" class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
  <constructor-arg ref="dataSource"/>
</bean>
<!-- the cache manager -->
<bean id="cacheManager" class="EhCacheManagerFactoryBean">
  <property name="configLocation" value="classpath:${ehcache.config}"/>
</bean>
<!-- the pet cache to be injected into the pet dao -->
<bean name="petCache" class="EhCacheWrapper">
  <constructor-arg value="pets"/>
  <constructor-arg ref="cacheManager"/>
</bean>
<!-- the pet dao -->
<bean id="petDao" class="PetDaoImpl">
  <property name="cache" ref="petCache"/>
  <property name="jdbcTemplate" ref="jdbcTemplate"/>
  <property name="datasource" ref="dataSource"/>
</bean>
```

OpenJPA Caching Provider

Introduction

Ehcache easily integrates with the OpenJPA persistence framework. This page provides installation and configuration information.

Installation

To use OpenJPA, add a Maven dependency for ehcache-openjpa.

```
<groupId>net.sf.ehcache</groupId>
<artifactId>ehcache-openjpa</artifactId>
<version>0.1</version>
```

Or, download from Downloads.

Configuration

For enabling Ehcache as second level cache, the persistence.xml file should include the following configurations:

```
<property name="openjpa.Log" value="SQL=TRACE" />
<property name="openjpa.QueryCache" value="ehcache" />
<property name="openjpa.DataCache" value="true"/>
<property name="openjpa.RemoteCommitProvider" value="sjvm"/>
<property name="openjpa.DataCacheManager" value="ehcache" />
```

The ehcache.xml file can be configured like this example:

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="ehc
dynamicConfig="true" name="TestCache">
<diskStore path="java.io.tmpdir"/>

<defaultCache maxEntriesLocalHeap="10000"
            eternal="false"
            timeToIdleSeconds="120"
            timeToLiveSeconds="120"
            overflowToDisk="true"
            diskPersistent="false"
            memoryStoreEvictionPolicy="LRU" />

<cache name="com.terracotta.domain.DataTest"
    maxEntriesLocalHeap="200"
    eternal="false"
    timeToIdleSeconds="2400"
    timeToLiveSeconds="2400"
    memoryStoreEvictionPolicy="LRU">
</cache>

<cache name="openjpa"
    maxEntriesLocalHeap="20000"
    eternal="true"
    memoryStoreEvictionPolicy="LRU">
```

```
</cache>

<cache name="openjpa-querycache"
    maxEntriesLocalHeap="20000"
    eternal="true"
    memoryStoreEvictionPolicy="LRU">
</cache>

<cacheManagerPeerListenerFactory
    class="org.terracotta.ehcachedx.monitor.probe.ProbePeerListenerFactory"
        properties="monitorAddress=localhost, monitorPort=9889, memoryMeasurement=true" />
</ehcache>
```

Default Cache

As with Hibernate, Ehcache's OpenJPA module (from 0.2) uses the `defaultCache` configured in ehcache.xml to create caches. For production, we recommend configuring a cache configuration in ehcache.xml for each cache, so that it may be correctly tuned.

Troubleshooting

To verify that that OpenJPA is using Ehcache:

- look for hits/misses in the Ehcache monitor
- view the SQL Trace to find out whether it queries the database

If there are still problems, verify in the logs and that your ehcache.xml file is being used. (It could be that if the ehcache.xml file is not found, ehcache-failsafe.xml is used by default.)

For Further Information

For more on caching in OpenJPA, refer to the Apache OpenJPA project.

Using Grails and Ehcache

Introduction

Grails 1.2RC1 and higher use Ehcache as the default Hibernate second level cache. However earlier versions of Grails ship with the Ehcache library and are very simple to enable. The following steps show how to configure Grails to use Ehcache. For 1.2RC1 and higher some of these steps are already done for you.

Configuring Ehcache As the Second Level Hibernate Cache

Edit `DataSource.groovy` as follows:

```
hibernate {
cache.use_second_level_cache=true
cache.use_query_cache=true
cache.provider_class='org.hibernate.cache.EhCacheProvider'
}
```

Overriding Defaults

As is usual with Hibernate, it will use the `defaultCache` configuration as a template to create new caches as required. For production use you often want to customise the cache configuration. To do so, add an ehcache.xml configuration file to the `conf` directory (the same directory that contains `DataSource.groovy`). A sample ehcache.xml which works with the Book demo app and is good as a starter configuration for Grails is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="ehcache.xsd" >
<diskStore path="java.io.tmpdir"/>
<cacheManagerEventListenerFactory class="" properties=""/>
<defaultCache
       maxEntriesLocalHeap="10000"
       eternal="false"
       timeToLiveSeconds="120"
       overflowToDisk="false"
       diskPersistent="false"
       />
<cache name="Book"
      maxEntriesLocalHeap="10000"
      timeToIdleSeconds="300"
       />
<cache name="org.hibernate.cache.UpdateTimestampsCache"
      maxEntriesLocalHeap="10000"
      timeToIdleSeconds="300"
       />
cache name="org.hibernate.cache.StandardQueryCache"
      maxEntriesLocalHeap="10000"
      timeToIdleSeconds="300"
       />
</ehcache>
```

Springcache Plugin

The Springcache plugin allows you to easily add the following functionality to your Grails project:

- Caching of Spring bean methods (typically Grails service methods).
- Caching of page fragments generated by Grails controllers.
- Cache flushing when Spring bean methods or controller actions are invoked.

The plugin depends on the EhCache and EhCache-Web libraries. See Springcache Plugin, a part of the Grails project, for more information.

Clustering Web Sessions

This is handled by Terracotta Web Sessions. See this blog for a great intro on getting this going with Grails and Tomcat.

Glassfish How To & FAQ

Introduction

The maintainer uses Ehcache in production with Glassfish. This page explains how to package a sample application using Ehcache and deploy to Glassfish.

Versions

Ehcache has been tested with and is used in production with Glassfish V1, V2 and V3. In particular:

- Ehcache 1.4 - 1.7 has been tested with Glassfish 1 and 2.
- Ehcache 2.0 has been tested with Glassfish 3.

Deployment

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.

You need:

- An Ehcache core installation
- A Glassfish installation
- A `GLASSFISH_HOME` environment variable defined.
- `$GLASSFISH_HOME/bin` added to your `PATH`

Run the following from the Ehcache `core` directory:

```
# 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 this page for a quickstart to Glassfish.

Troubleshooting

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>
```

Ehcache throws an IllegalStateException in Glassfish

Ehcache page caching versions below Ehcache 1.3 get an IllegalStateException in Glassfish. This issue was fixed in Ehcache 1.3.

PayloadUtil reports `Could not ungzip. Heartbeat will not be working. Not in GZIP format`

This exception is thrown when using Ehcache with my Glassfish cluster, but 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 from Glassfish).

JSR107 (JCACHE) Support

{#jsr107-implementation} JSR107 is being currently 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 compatibility between versions of the package.

Information on the Ehcache implementation of JSR107, JCACHE, is available on github.

You can also find out more about the JSR effort on github.

Recipes Overview

The recipes here are concise examples for specific use cases that will help you get started with Ehcache.

The following sections provide a documentation Table of Contents and additional information about Recipes.

Recipes Table of Contents

Recipe	Description
Web Page and Fragment Caching	How to use the included Servlet Filters to cache web pages and web page fragments.
Configure a Grails App for Clustering	How to configure a Grails Application for clustered Hibernate 2nd Level Cache.
Data Freshness and Expiration	How to maintain cache "freshness" by configuring TTL and data expiration properly.
Enable Terracotta Programmatically	How to enable Terracotta support for Ehcache programmatically.
WAN Replication	Three strategies for configuring WAN replication.
Caching Empty Values	Why caching empty values can be desirable to deflect load from the database.
Database Read Overload	When many readers simultaneously request the same data element, it is called the "Thundering Herd" problem. How to prevent it in a single JVM or a clustered configuration.
Database Write Overload	Writing to the database is a bottleneck. Configure the Ehcach Write-behind feature to offload database writes.
Caching methods with Spring Annotations	Adding caching to methods using the Ehcache Annotations for Spring project.
Cache Wrapper	A simple class to make accessing Ehcache easier for simple use cases.

Let's Add More

If you have suggestions or ideas for more recipes, please tell us about them using the forums or mailing list.

Web Page and Web Page Fragment Caching

Introduction

This page provides an example of how to use the included Servlet Filters to cache web pages and web page fragments.

Problem

You'd like to improve the time it takes to return a page from your web application. Many of the pages in your application are not time or user specific and can be cached for a period of time.

Solution

Cache the entirety of the web page, or a fragment of the web page for a period of time. Rather than having to generate the page on each page hit, it will be served out of the cache.

Modern application hardware should be able to server as many as 5,000 pages per second, affording a significant speedup in your application for pages that are frequently read but infrequently change.

Discussion

There are no code changes required for this - your application server should support servlet filtering already. Simply update your web.xml file, re-deploy and you should see the speedup right away.

The basic steps you'll need to follow to configure Ehcache for web page caching are (note that these steps assume you already have Ehcache installed in your application):

1. Configure a servlet page filter in web.xml
2. Configure an appropriate cache in ehcache.xml
3. Start (or re-start) your application

The following settings should help you setup web caching for your application.

Step 1 - Add a filter to your web.xml

The first thing you'll need to do is add a filter to enable page caching.

The following web.xml settings will enable a servlet filter for page caching:

```
<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>SimplePageCachingFilter</filter-name>
    <filter-class>net.sf.ehcache.constructs.web.filter.SimplePageCachingFilter
    </filter-class>
  </filter>


  <filter-mapping>
    <filter-name>SimplePageCachingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>
```

Step 2 - Configure an ehcache.xml

The second step to enabling web page caching is to configure ehcache with an appropriate ehcache.xml.

The following ehcache.xml file should configure a reasonable default ehcache:

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:noNamespaceSchemaLocation="../../main/config/ehcache.xsd">
    <cache name="SimplePageCachingFilter"
           maxEntriesLocalHeap="10000"
           maxElementsOnDisk="1000"
           eternal="false"
           overflowToDisk="true"
           timeToIdleSeconds="300"
           timeToLiveSeconds="600"
           memoryStoreEvictionPolicy="LFU"
            />
</ehcache>
```

Step 3 - Start your application server

Now start your application server. Pages should be cached.

More details

For more details and configuration options pertaining to web page and web page fragment caching, see the Web Caching page in the user documentation.

Using Grails and Ehcache

Introduction

Grails 1.2RC1 and higher use Ehcache as the default Hibernate second level cache. However earlier versions of Grails ship with the Ehcache library and are very simple to enable. The following steps show how to configure Grails to use Ehcache. For 1.2RC1 and higher, some of these steps are already done for you.

Configuring Ehcache As the Second Level Hibernate Cache

Edit `DataSource.groovy` as follows:

```
hibernate {
cache.use_second_level_cache=true
cache.use_query_cache=true
cache.provider_class='org.hibernate.cache.EhCacheProvider'
}
```

Overriding Defaults

As is usual with Hibernate, it will use the `defaultCache` configuration as a template to create new caches as required. For production use you often want to customise the cache configuration. To do so, add an ehcache.xml configuration file to the `conf` directory (the same directory that contains `DataSource.groovy`). A sample ehcache.xml which works with the Book demo app and is good as a starter configuration for Grails is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="ehcache.xsd" >
<diskStore path="java.io.tmpdir"/>
<cacheManagerEventListenerFactory class="" properties=""/>
<defaultCache
       maxEntriesLocalHeap="10000"
       eternal="false"
       timeToLiveSeconds="120"
       overflowToDisk="false"
       diskPersistent="false"
       />
<cache name="Book"
      maxEntriesLocalHeap="10000"
      timeToIdleSeconds="300"
       />
<cache name="org.hibernate.cache.UpdateTimestampsCache"
      maxEntriesLocalHeap="10000"
      timeToIdleSeconds="300"
       />
cache name="org.hibernate.cache.StandardQueryCache"
      maxEntriesLocalHeap="10000"
      timeToIdleSeconds="300"
       />
</ehcache>
```

Springcache Plugin

The Springcache plugin allows you to easily add the following functionality to your Grails project:

- Caching of Spring bean methods (typically Grails service methods).
- Caching of page fragments generated by Grails controllers.
- Cache flushing when Spring bean methods or controller actions are invoked.

The plugin depends on the EhCache and EhCache-Web libraries. See Springcache Plugin, a part of the Grails project, for more information.

Clustering Web Sessions

This is handled by Terracotta Web Sessions. See this blog for a great intro on getting this going with Grails and Tomcat.

Data Freshness and Expiration

Introduction

This page addresses how to maintain cache "freshness" by configuring TTL and data expiration properly.

Problem

Data in the cache is out of sync with the SOR (the database).

Solution

Databases (and other SORs) weren't built with caching outside of the database in mind, and therefore don't normally come with any default mechanism for notifying external processes when data has been updated or modified.

Use one of the following strategies to keep the data in the cache in sync:

- **data expiration**: use the eviction algorithms included with Ehcache along with the timeToIdleSeconds and timetoLiveSeconds setting to enforce a maximum time for elements to live in the cache (forcing a re-load from the database or SOR).
- **message bus**: use an application to make all updates to the database. When updates are made, post a message onto a message queue with a key to the item that was updated. All application instances can subscribe to the message bus and receive messages about data that is updated, and can synchronize their local copy of the data accordingly (for example by invalidating the cache entry for updated data)
- **triggers**: Using a database trigger can accomplish a similar task as the message bus approach. Use the database trigger to execute code that can publish a message to a message bus. The advantage to this approach is that updates to the database do not have to be made only through a special application. The downside is that not all database triggers support full execution environments and it is often unadvisable to execute heavy-weight processing such as publishing messages on a queue during a database trigger.

Discussion

The data expiration method is the simplest and most straightforward.

It gives you the programmer the most control over the data synchronization, and doesn't require cooperation from any external systems, you simply set a data expiration policy and let Ehcache expire data from the cache, thus allowing fresh reads to re-populate and re-synchronize the cache.

If you choose the data expiration method, you can read more about the cache configuration settings at cache eviction algorithms and timeToIdle and timeToLive configuration settings. The most important concern to consider when using the expiration method is balancing data-freshness with database load. The shorter you make the expiration settings - meaning the more "fresh" you try to make the data - the more load you will incur on the database.

Try out some numbers and see what kind of load your application generates. Even modestly short values such as 5 or 10 minutes will afford significant load reductions.

Enable Terracotta Support Programmatically

Introduction

This page covers how to enable Terracotta support for Ehcache programmatically.

Problem

You configure and use Ehcache programmatically. You'd like to enable Terracotta support.

Solution

You can create a Terracotta configuration programmatically and configure it in your CacheManager.

Discussion

Here is some code that you can use to create a Terracotta Configuration and add it to your Ehcache configuration:

```
import net.sf.ehcache.Cache;
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.config.Configuration;
import net.sf.ehcache.config.CacheConfiguration;
import net.sf.ehcache.config.TerracottaConfiguration;
import net.sf.ehcache.config.TerracottaClientConfiguration;


public class Main
{
    private static final String CACHE_NAME = "myCache";

    public static void main(String args[]) throws Exception
    {
        // The main configuration bean
        Configuration configuration = new Configuration();

        // Setup the Terracotta cluster config
        TerracottaClientConfiguration terracottaConfig
            = new TerracottaClientConfiguration();

        // If you want to point to a different URL, do it here, otherwise the
        // default will point to a local Terracotta server array
        // terracottaConfig.setUrl(...);
        configuration.addTerracottaConfig(terracottaConfig);

        // Setup a default cache and add to the configuration
        CacheConfiguration defaultCache = new CacheConfiguration("default", 1000)
            .eternal(false);
        configuration.addDefaultCache(defaultCache);

        // Setup "myCache", make it clustered and add to the configuration
        CacheConfiguration myCache = new CacheConfiguration(CACHE_NAME, 10000)
            .eternal(false)
            .terracotta(new TerracottaConfiguration());
        configuration.addCache(myCache);

        CacheManager mgr = new CacheManager(configuration);
        Cache exampleCache = mgr.getCache(CACHE_NAME);
        assert (exampleCache != null);
    }
}
```

Strategies For Setting Up WAN Replication

Introduction

This page provides three strategies for configuring WAN replication.

Problem

You have two sites for high availability and/or disaster recovery that remain in sync with one another. The two sites are located in geographically separate areas connected by a WAN link.

Solutions

There are three mechanisms offered by Terracotta to replicate your Ehcache. This recipe highlights the general approach taken by each of these three solutions. It begins with the simplest but least reliable, and concludes with the most robust and comprehensive mechanism.

Solution 1: Terracotta Active/Mirror Replication

This is the simplest configuration of the three solutions. In this solution, the approach is to simply use the built-in replication capabilities of the Terracotta Server Array. In this solution, one Terracotta Server Array Instance is positioned in each data center. At any one moment only one Terracotta Server Instance is active. This solution is ideal for data centers that are connected by a high-speed WAN link and maximum simplicity is required.

Diagram of solution:

Characteristics

This solution has the following characteristics.

Reads

All reads are done from just the one active Terracotta Server Array Instance. This means that clients in data-center will read from the Terracotta Server Array using a LAN connection, and clients in the other data-center will read from the Terracotta Server Array using a WAN connection.

Writes

All writes are performed against just the one active Terracotta Server Array Instance. This means that one clients in one data-center will write to the Terracotta Server Array using a LAN connection, and clients in the other data-center will write to the Terracotta Server Array using a WAN connection.

Summary

Pros:

- Simple
- Easy to manage

Cons:

- Completely dependent on an ideal network connection.
- Even with a fast WAN connection (both high throughput and low-latency), latency issues are not unlikely as unexpected slowdowns in the network or within the cluster occur.
- Split-brain scenarios may occur due to interruptions in the network between the two servers.
- Slowdowns lead to stale data or long pauses for clients in Datacenter B.

Solution 2: Transactional Cache Manager Replication

This solution relies on Ehcache Transaction (JTA) support. In this configuration, two separate caches are created, each one is 'homed' to a specific data-center.

When a write is generated, it is written under a JTA transaction to ensure data integrity. The write is written to both caches, so that when the write completes, each data-center specific cache will have a copy of the write.

This solution trades off some write performance for high read performance. Executing a client level JTA transaction can result in slower performance than Terracotta's built-in replication scheme. The trade-off however results in the ability for both data-centers to read from a local cache for all reads.

This solution is ideal for applications where writes are infrequent and high read throughput and or low read-latency is required.

Diagram of solution:

Characteristics

This solution has the following characteristics.

Reads

All reads are done against a local cache / Terracotta Server Array

Writes

All writes are performed against both caches (one in the local LAN and one across the remote WAN) simultaneously transactionally using JTA.

Summary

Pros:

- High read throughput (all reads are executed against local cache)
- Low read latency (all reads are executed against local cache)

Cons:

- An XA transaction manager is required
- Write cost may be higher
- Some of the same latency and throughput issues that occur in Solution 1 can occur here if writes are delayed.

Solution 3: Messaging based (AMQ) replication

This solution relies on a message bus to send replication events. The advantage of this solution over the previous two solutions is the ability to configure - and fine-tune - the characteristics and behavior of the replication. Using a custom replicator that reads updates from a local cache combined with the ability to schedule and/or batch replication can make replication across the WAN significantly more efficient.

See Terracotta Distributed Ehcache WAN Replication to learn more about the Terracotta version of this solution.

Diagram of solution:

Characteristics

This solution has the following characteristics.

Reads

All reads are done against a local cache / Terracotta Server Array

Writes

All writes are done against a local cache for reliable updates. Write updates are collected and sent at a configurable frequency across the message bus.

This approach allows for batch scheduling and tuning of batch size so that updates can utilize the WAN link efficiently.

Summary

Pros:

- High read throughput (all reads are executed against local cache)
- Low read latency (all reads are executed against local cache)
- Write replication is highly efficient and tunable
- Available as a fully featured solution, Terracotta Distributed Ehcache WAN Replication, which includes persistence, delivery guaranty, conflict resolution, and more.

Cons:

- A message bus is required

Caching Empty Values

Introduction

This page discusses why caching empty values can be desirable to deflect load from the database.

Problem

Your application is querying the database excessively only to find that there is no result. Since there is no result, there is nothing to cache.

How do you prevent the query from being executed unneccesarily?

Solution

Cache a null value, signalling that a particular key doesn't exist.

Discussion

Ehcache supports caching null values. Simply cache a "null" value instead of a real value.

Use a maximum time to live setting in your cache settings to force a re-load every once in a while.

In code, checking for intentional nulls versus non-existent cache entries may look like:

```
// cache an explicit null value:

cache.put(new Element("key", null));

Element element = cache.get("key");

if (element == null) {

    // nothing in the cache for "key" (or expired) ...

} else {

    // there is a valid element in the cache, however getObjectValue() may be null:

    Object value = element.getObjectValue();

    if (value == null) {

        // a null value is in the cache ...

    } else {

        // a non-null value is in the cache ...

    }

}
```

Discussion

And the ehcache.xml file may look like this (making sure to set the maximum time to live setting:

```
<cache
    name="some.cache.name"
    maxEntriesLocalHeap="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
/>
```

Thundering Herd

Introduction

When many readers simultaneously request the same data element, there can be a database read overload, sometimes called the "Thundering Herd" problem. This page addresses how to prevent it in a single JVM or a clustered configuration.

Problem

Many readers read an empty value from the cache and subseqeuntly try to load it from the database. The result is unnecessary database load as all readers simultaneously execute the same query against the database.

Solution

Implement the cache-as-sor pattern by using a BlockingCache or SelfPopulatingCache included with Ehcache.

Using the BlockingCache Ehcache will automatically block all threads that are simultaneously requesting a particular value and let one and only one thread through to the database. Once that thread has populated the cache, the other threads will be allowed to read the cached value.

Even better, when used in a cluster with Terracotta, Ehcache will automatically coordinate access to the cache across the cluster, and no matter how many application servers are deployed, still only one user request will be serviced by the database on cache misses.

Discussion

The "thundering herd" problem occurs in a highly concurrent environment (typically, many users). When many users make a request to the same piece of data at the same time, and there is a cache miss (the data for the cached element is not present in the cache) the thundering herd problem is triggered.

Imagine that a popular news story has just surfaced on the front page of a news site. The news story has not yet been loaded in to the cache.

The application is using a cache using a read-through pattern with code that looks approximately like:

```
/* read some data, check cache first, otherwise read from sor */
public V readSomeData(K key)
{
    Element element;
    if ((element = cache.get(key)) != null) {
        return element.getValue();
    }

    // note here you should decide whether your cache
    // will cache 'nulls' or not
    if (value = readDataFromDataStore(key)) != null) {
        cache.put(new Element(key, value));
    }

    return value;
```

```
}
```

Upon publication to the front page of a website, a news story will then likely be clicked on by many users all at approximately the same time.

Since the application server is processing all of the user requests simultaneously, the application code will execute the above code all at approximately the same time. This is especially important to consider, because all user requests will be evaluating the cache (line 105) contents at approximately the same time, and reach the same conclusion: the cache request is a miss!

Therefore all of the user request threads will then move on to read the data from the SOR. So, even though the application designer was careful to implement caching in the application, the database is still subject to spikes of activity.

The thundering herd problem is made even worse when there are many application servers to one database server, as the number of simultaneous hits the database server may receive increases as a function of the number of application servers deployed.

Ehcache Write-Behind

Introduction

This page addresses the problem of database write overload and explains how the Ehcache Write-behind feature can be the solution.

Problem

It's easy to understand how a cache can help reduce database loads and improve application performance in a read-mostly scenario. In read-mostly use-cases, every time the application needs to access data, instead of going to the database, data can be loaded from in-memory cache, which can be hundreds, or even thousands, of times faster than database.

However, for scenarios that require frequent updates to the stored data, to keep the data in cache and database in sync, every update to the cached data must invoke a simultaneous update to the database at the same time. Updates to the database are almost always slower, so this slows the effective update rate to the cache and thus the performance in general. When many write requests come in at the same time, the database can easily become a bottleneck or, even worse, be killed by heavy writes in a short period of time.

Solution

The Write-behind feature provided by Ehcache allows quick cache writes with ensured consistency between cache data and database.

The idea is that when writing data into the cache, instead of writing the data into database at the same time, the write-behind cache saves the changed data into a queue and lets a backend thread to do the writing later. Therefore, the cache-write process can proceed without waiting for the database-write and, thus, be finished much faster. Any data that has been changed can be persisted into database eventually. In the mean time, any read from cache will still get the latest data.

A cache configured to perform asynchronous persistence, such as this, is called a Write-behind Cache.

There are many benefits of a Write-behind Cache. For example:

- Offload database writes
- Spread writes out to flatten peaks
- Consolidate multiple writes into fewer database writes

Discussion

To implement a Write-Behind using Ehcache, one needs to register a CacheWriterFactory for Write-behind Cache and set the writeMode property of the cache to "write_behind".

CacheWriterFactory can create a writer for any data source(s), such as file, email, JMS or database. Typically, the database is the most common example of a data source.

Discussion

Once a cache is configured as a Write-Behind cache, whenever a Cache.put is called to add or modify data, the cache will first update the cache data, just like a normal cache does, then it will save the change into a queue. A backend thread should be started when the cache is initialized and it will keep pulling data from the queue and it will call a Writer instance created by the CacheWriterFactory to persist the new data asynchronously.

In an un-clustered cache, the write-behind queue is stored in local memory. If the JVM dies, any data still in the queue will be lost.

In a clustered cache, the write-behind queue is managed by Terracotta Server Array. The background thread on each JVM will check the shared queue and save each data change left in the queue. With clustered Ehcache, this background process is scaled across the cluster for both performance and high availability reasons. If one client JVM were to go down, any changes it put into the write-behind queue can always be loaded by threads in other clustered JVMs, therefore will be applied to the database without any data loss.

There are many advanced configurations for Write-behind Cache. Because of the nature of asynchronous writing, there are also restrictions on when Write-Behind Cache can be used. For more information, see write-through caching.

Caching Methods with Spring 3 Annotations

Introduction

This page explains adding caching to methods using the Ehcache Annotations for Spring project.

Problem

You'd like to cache methods of your application with minimal code changes and use configuration to control the cache settings.

Solution

Use the Ehcache Annotations for Spring project at code.google.com to dynamically configure caching of method return values.

Discussion

The Ehcache Annotations for Spring project is a successor to the Spring-Modules project. This project will allow you to configure caching of method calls dynamically using just configuration.

The way it works is that the parameter values of the method will be used as a composite key into the cache, caching the return value of the method.

For example, suppose you have a method: `Dog getDog(String name).`

Once caching is added to this method, all calls to the method will be cached using the "name" parameter as a key.

So, assume at time t0 the application calls this method with the name equal to "fido". Since "fido" doesn't exist, the method is allowed to run, generating the "fido" Dog object, and returning it. This object is then put into the cache using the key "fido".

Then assume at time t1 the application calls this method with the name equal to "spot". The same process is repeated, and the cache is now populated with the Dog object named "spot".

Finally, at time t2 the application again calls the method with the name "fido". Since "fido" exists in the cache, the "fido" Dog object is returned from the cache instead of calling the method.

To implement this in your application, follow these steps:

Step 1:

Add the jars to your application as listed on the Ehcache Annotations for Spring project site.

Step 2:

Discussion

Add the Annotation to methods you would like to cache. Lets assume you are using the Dog getDog(String name) method from above:

```
@Cacheable(name="getDog")
Dog getDog(String name)
{
    ....
}
```

Step 3:

Configure Spring. You must add the following to your Spring configuration file in the beans declaration section:

```
<ehcache:annotation-driven cache-manager="ehCacheManager" />
```

More details can be found at:

- Ehcache Annotations for Spring project
- the project getting started page
- this blog

Echache Wrapper

Introduction

This page provides an example of a simple class to make accessing Ehcache easier for simple use cases.

Problem

Using the full Ehcache API can be more tedious than using just a simple, value-based cache (like a HashMap) because of the Element class that holds values.

Solution

Implement a simple cache wrapper to hide the use of the Element class.

Discussion

Here's a simple class you can use to simplify using Ehcache in certain simple use cases.

You can still get the Ehcache cache in case you want access to the full API.

```
public interface CacheWrapper<K, V>
{
  void put(K key, V value);

  V get(K key);
}

import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Ehcache;
import net.sf.ehcache.Element;

public class EhcacheWrapper<K, V> implements CacheWrapper<K, V>
{
    private final String cacheName;
    private final CacheManager cacheManager;

    public EhcacheWrapper(final String cacheName, final CacheManager cacheManager)
    {
        this.cacheName = cacheName;
        this.cacheManager = cacheManager;
    }

    public void put(final K key, final V value)
    {
        getCache().put(new Element(key, value));
    }

    public V get(final K key)
    {
        Element element = getCache().get(key);
        if (element != null) {
            return (V) element.getValue();
        }
        return null;
    }

    public Ehcache getCache()
    {
        return cacheManager.getEhcache(cacheName);
    }
}
```