Integrations

Version 2.10.3

October 2016

EHCACHE

This document applies to Ehcache Version 2.10.3 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: EHC-INT-2103-20161018

Table of Contents

1 Using Ehcache with Hibernate

About Using Ehcache with Hibernate

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.

Ehcache easily integrates with the Hibernate Object/Relational persistence and query service. Gavin King, the maintainer of Hibernate, is also a committer to the BigMemory Go's Ehcache project. This ensures Ehcache will remain a first-class data store for Hibernate.

Configuring Ehcache for Hibernate is simple. The basic steps are as follows:

- Download and install Ehcache in your project as described in "Downloading and Installing Ehcache for Hibernate" on page 7.
- Configure Ehcache as a cache provider in your project's Hibernate configuration as described in "Configuring Ehcache as the Second-Level Cache Provider" on page 8.
- Enable second-level caching in your project's Hibernate configuration as described in "Enabling Second-Level Cache and Query Cache Settings" on page 8.
- Configure Hibernate caching for each entity, collection, or query that you want to cache as described in "Configuring Hibernate Entities to use Second-Level Caching" on page 9.
- Configure the `ehcache.xml` file for each entity, collection, or query configured for caching as described in "Configuring ehcache.xml Settings" on page 10.

For additional information about cache configuration in Hibernate, see the Hibernate product documentation at http://www.hibernate.org/.

Important Notices - PLEASE READ

In order for Hibernate 4.x to work with Ehcache 2.7.5 or higher, you need to install a patch. The problem and the indicated solution are described on the Hibernate website at https://hibernate.atlassian.net/browse/HHH-8732.

Note: The patch is a CODE PATCH that you need to apply to the Hibernate 4 source. Afterwards you need to re-build the Hibernate artifacts.

Installing the patch involves the following steps:

1. Access the patch at https://github.com/hibernate/hibernate-orm/pull/643.patch.
2. Get the Hibernate 4.x source code, for example at https://github.com/hibernate/hibernate-orm/tree/4.3.
3. Patch the Hibernate source code with the patch.
4. Build the Hibernate artifacts. This is explained on the github source page referenced above.
5. Reference the custom-built Hibernate artifacts in your project.

Users of Ehcache for Hibernate prior to Ehcache 2.0 should read "Upgrading from Ehcache Versions Prior to 2.0" on page 14. These instructions are for Hibernate 3.

Downloading and Installing Ehcache for Hibernate

The Hibernate provider is in the ehcache-core module. You can download the latest version of this module from here: http://sourceforge.net/projects/ehcache/files/ehcache-core/.

Building with 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 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>
```

For the Hibernate-Ehcache integration, add the following dependency:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-ehcache</artifactId>
  <version>${hibernateVersion}</version>
</dependency>
```

For example, the Hibernate-Ehcache integration dependency for Hibernate 4.0.0 is:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-ehcache</artifactId>
  <version>4.0.0</version>
</dependency>
```

Note: Some versions of Hibernate-Ehcache may have a dependency on a specific version of Ehcache. Check the Hibernate-Ehcache POM for more information.

Configuring Ehcache as the Second-Level Cache Provider

To configure Ehcache as a Hibernate second-level cache, set the region factory property to one of the following in the Hibernate configuration. The Hibernate configuration is specified either via hibernate.cfg.xml, hibernate.properties or Spring. The format shown below is for hibernate.cfg.xml.

Hibernate 3.3 and higher

For instance creation, use:

```
<property name="hibernate.cache.region.factory_class">
         net.sf.ehcache.hibernate.EhCacheRegionFactory</property>
```

To force Hibernate to use a singleton of Ehcache CacheManager, use:

```
<property name="hibernate.cache.region.factory_class">
         net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory</property>
```

Hibernate 4.x

For instance creation, use:

```
<property name="hibernate.cache.region.factory_class">
         org.hibernate.cache.ehcache.EhCacheRegionFactory</property>
```

To force Hibernate to use a singleton of Ehcache CacheManager, use:

```
<property name="hibernate.cache.region.factory_class">
         org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory</property>
```

Enabling 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). To do this, set the following property in your Hibernate config:

```
<property name="hibernate.cache.use_second_level_cache">true</property>
```

You might also want to turn on the Hibernate query cache. To do this, set the following property in your Hibernate config:

```
<property name="hibernate.cache.use_query_cache">true</property>
```

Setting the ConfigurationResourceName Property

You can optionally set the `ConfigurationResourceName` property to specify the location of the Ehcache configuration file to use 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 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
```

Setting the Hibernate Cache Provider Programmatically

You can optionally specify the provider 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")
```

For Hibernate 4, use `org.hibernate.cache.ehcache.EhCacheRegionFactory` instead of `net.sf.ehcache.hibernate.EhCacheRegionFactory`.

Putting it all Together

If you are 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.EhCacheRegionFactory</property>
```

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</prop>
```

For Hibernate 4, use `org.hibernate.cache.ehcache.EhCacheRegionFactory` instead of `net.sf.ehcache.hibernate.EhCacheRegionFactory` in both samples given above.

Configuring 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, the mapping entry for a domain object called, com.somecompany.someproject.domain.Country, looks something like this:

```
<hibernate-mapping>
<class
name="com.somecompany.someproject.domain.Country"
```

```
table="ut_Countries"
dynamic-update="false"
dynamic-insert="false"
>
...
</class>
</hibernate-mapping>
```

To enable caching for this domain object, you add the following element to its mapping entry:

```
<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>
```

You can also enable caching using the @Cache annotation as shown below.

```
@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.
- `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.

Configuring ehcache.xml Settings

Because the ehcache.xml file has a defaultCache, caches will always be created when required by Hibernate. However you can gain more control over Hibernate caches by configuring each cache based on its name. Doing this is particularly important, because Hibernate caches are populated from databases, and there is potential for them to become very large. You can control the size of a Hibernate cache by capping its `maxEntriesLocalHeap` property and specifying whether to swap to disk beyond that.

Ehcache Settings for Domain Objects

Hibernate bases the names of Domain Object caches on the fully qualified name of Domain Objects. So, for example, a cache for com.somecompany.someproject.domain.Country would be represented by a cache configuration entry in ehcache.xml similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
 <cache
    name="com.somecompany.someproject.domain.Country"
    maxEntriesLocalHeap="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    <persistence strategy="localTempSwap"/>
 />
</ehcache>
```

Hibernate CacheConcurrencyStrategy for Domain Objects

The read-write, nonstrict-read-write and read-only policies apply to Domain Objects.

Ehcache Settings for Collections

Hibernate creates collection cache names based on the fully qualified name of the Domain Object followed by "." and the collection field name. For example, a Country domain object has a set of advancedSearchFacilities. The Hibernate doclet for the accessor looks like this:

```
/**
* Returns the advanced search facilities that should appear for this country.
* @hibernate.set cascade="all" inverse="true"
* @hibernate.collection-key column="COUNTRY_ID"
* @hibernate.collection-one-to-many class="com.wotif.jaguar.domain.AdvancedSearchFacility"
* @hibernate.cache usage="read-write"
*/
public Set getAdvancedSearchFacilities() {
return advancedSearchFacilities;
}
```

You need an additional cache configured for the set. The ehcache.xml configuration looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
 <cache name="com.somecompany.someproject.domain.Country"
    maxEntriesLocalHeap="50"
    eternal="false"
    timeToLiveSeconds="600"
    <persistence strategy="localTempSwap"/>
/>
 <cache
name="com.somecompany.someproject.domain.Country.advancedSearchFacilities"
    maxEntriesLocalHeap="450"
    eternal="false"
    timeToLiveSeconds="600"
```

```
    <persistence strategy="localTempSwap"/>
/>
</ehcache>
```

Hibernate CacheConcurrencyStrategy for Collections

The read-write, nonstrict-read-write and read-only policies apply to Domain Object collections.

Ehcache Settings for Queries

Hibernate allows the caching of query results.

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"
<persistence strategy="localTempSwap"/>
/>
```

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 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"
<persistence strategy="localTempSwap"/>
/>
```

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. For example:

```
<cache name="query.AdministrativeAreasPerCountry"
maxEntriesLocalHeap="5"
eternal="false"
timeToLiveSeconds="86400"
<persistence strategy="localTempSwap"/>
/>
```

Using Query Caches

Let's say you have a common query running against the Country Domain. Here is the code to use a query cache with it:

```
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 for Queries

None of the 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.

The Demo Application and Tutorial

A demo application is available that shows you how to use the Hibernate CacheRegionFactory. You can download the application from here: http://svn.terracotta.org/svn/forge/projects/hibernate-tutorial-web/trunk.

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.

Viewing Hibernate Statistics

It is possible to access the Hibernate statistics and Ehcache statistics using the Java Management Extensions (JMX).

The EhcacheHibernateMBean is the main interface that exposes all the APIs via JMX. It basically extends two interfaces: EhcacheStats and HibernateStats. As the names imply, EhcacheStats contains methods related with Ehcache and HibernateStats contains methods related with Hibernate.

Using these APIs, you can see cache hit/miss/put rates, change config element values (e.g., maxElementInMemory, TTL TTI), enable/disable statistics collection, and various other things. For details, see the specific interface.

For additional information about using JMX to monitor and manage your caches, see the *Ehcache Operations Guide*.

Upgrading from Ehcache Versions Prior to 2.0

This topic contains notes about upgrading from versions of Ehcache prior to 2.0.

Support for Hibernate 3.3 SPI

Beginning with Ehcache 2.0, there is support for the Hibernate 3.3 SPI implementation. This is important because Hibernate 3.3 has an updated caching SPI. Although still present in 3.3, the Hibernate 3.2 caching SPI has been deprecated.

Support for Hibernate 3.5 SPI

The SPI further changes in Hibernate 3.5. The Ehcache 2.0 implementation is forward-compatible with Hibernate 3.5.

Backward Compatibility

The EhCacheProvider class, which implements the 3.2 API, is provided for backward compatibility. Anyone already using Ehcache with Hibernate will be using this version. We encourage you to upgrade to the new class, net.sf.ehcache.hibernate.EhCacheRegionFactory, in preparation for when Hibernate drops support for the old SPI. In recognition of this, we have marked net.sf.ehcache.hibernate.EhCacheProvider as deprecated. The new cache region factory takes advantage of the new SPI to provide higher performance. The old SPI had heavy synchronization to ensure all of the different caching providers were thread-safe. The

new SPI leaves that to the implementer. Ehcache does not require extra synchronization, so this overhead is avoided.

Unification with Terracotta's Hibernate 3.2 Provider

In September 2009, Terracotta released its Hibernate Caching Provider which was set as follows:

```
<property name="hibernate.cache.provider_class">
     org.terracotta.hibernate.TerracottaHibernateCacheProvider</property>
```

It featured high performance clustered Hibernate caching using the Terracotta Server Array. The new 3.3 EhCacheRegionFactory replaces the Terracotta Hibernate Cache Provider as well as the old Ehcache provider. It is superset of the two earlier factories and also implements the new SPI. The earlier Hibernate provider also required a Java agent, which is no longer required in the new provider. We recommend that existing Terracotta Hibernate users upgrade to the Ehcache 2.0 provider.

FAQ

If I use BigMemory Go with my application 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 application may have a different lifecycle than Hibernate, and in each case your CacheManager "Automatic Resource Control" settings might need to be different.

Should I use the provider in the Hibernate distribution or in BigMemory Go's 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 to ensure Ehcache remains a first-class second-level cache for Hibernate.

Does BigMemory Go support the transactional strategy?

Yes. It was introduced in Ehcache 2.1.

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"/>.

Why do certain caches sometimes get automatically cleared by Hibernate?

Whenever a Query.executeUpdate() is run, Hibernate invalidates affected cache regions (those corresponding to affected database tables) to ensure that no stale data is cached. This should also happen whenever stored procedures are executed.

For more information, see the Hibernate issue HHH-2224 at : https://hibernate.atlassian.net/browse/HHH-2224.

How are Hibernate entities keyed?

Hibernate identifies cached entities using an object id. This is normally the primary key of a database row.

Are compound keys supported?

Yes.

I am getting this error message: "An item was expired by the cache while it was locked." 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 than a normal element. This is used in the Hibernate Read/Write strategy to force fall-through to the database during the two-phase commit. Although we don't know exactly what should be returned by the cache while the commit is in process, the database does. If a soft-locked element is evicted by the cache during the two-phase commit, then once the two-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, but could cause a small rise in database load.

So, in summary the Hibernate messages are not problematic. The underlying cause is that the probabilistic evictor can theoretically evict recently loaded items. You can also use the deterministic evictor to avoid this problem. Specify the `-Dnet.sf.ehcache.use.classic.lru=true` system property to turn on classic LRU, which contains a deterministic evictor.

2 Using Ehcache with ColdFusion

About ColdFusion and Ehcache

ColdFusion ships with Ehcache. The ColdFusion community has actively engaged with Ehcache and have put out lots of great blogs. Here are two to get you started. For a short introduction, see Raymond Camden's blog. For more in-depth analysis, see 14 days of ColdFusion caching, by Aaron West, covering a different topic each day.

Example Integration

To integrate Ehcache with ColdFusion, first add the Ehcache jars to your web application lib directory.

The following code demonstrates how to call Ehcache from ColdFusion. It will cache a ColdFusion object in Ehcache and set the 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 it 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>
```

3 Using Ehcache with Spring

Using Spring 3.1

Ehcache has had excellent Spring integration for years. Spring 3.1 includes an Ehcache implementation. See the Spring 3.1 JavaDoc.

Spring Framework 3.1 has a generic cache abstraction for transparently applying caching to Spring applications. It has 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 2.5 to 3.1

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.

Annotations for Spring Project

To dynamically configure caching of method return values, use the Ehcache Annotations for Spring project at code.google.com at Ehcache Annotations for Spring project at code.google.com. 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 at code.google.com at Ehcache Annotations for Spring project at code.google.com.

Step 2:

Add the Annotation to methods you would like to cache. Let's 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 at code.google.com at http://code.google.com/p/ehcache-spring-annotations.
- The project getting started page at http://code.google.com/p/ehcache-spring-annotations/wiki/UsingCacheable.
- The article "Caching Java methods with Spring 3" at http://www.jeviathon.com/2010/04/caching-java-methods-with-spring-3.html

4 Using Ehcache with JRuby and Rails

About Using Ehcache with JRuby

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.

Installation

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 for JRuby

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

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 not necessary to call this if the cache is configured for persistence 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:

Property	Argument Type	Description
unlessExist, ifAbsent	boolean	If true, use the putIfAbsent method.
elementEvictionData	ElementEvictionData	Sets this element's eviction data instance.
eternal	boolean	Sets whether the element is eternal.

Property	Argument Type	Description
timeToIdle, tti	int	Sets time to idle.
timeToLive, ttl, expiresIn	int	Sets time to live.
version	long	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

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 Ruby'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 in "Using the jruby-ehcache API directly" on page 25 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_address: string
   ```

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 development 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
   ```

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
```

Adding Off-Heap Storage under Rails

Terracotta BigMemory products provide in-memory data management with a large additional cache located right at the node where your application runs. To use the

off-heap storage provided by the Terracotta BigMemory products 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 off-heap storage:

   ```
   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 garbage collection 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 "Sizing Storage Tiers" in the *Configuration Guide* for your BigMemory product.
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 off-heap storage, 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 *Configuration Guide* for your BigMemory product.

Note that only serializable cache keys and values can be placed in the off-heap 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.

5 Using Ehcache with the Google App Engine

About Google App Engine (GAE) and Ehcache

The ehcache-googleappengine module combines the speed of Ehcache with the scale of Google's memcache and provides 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 "Use Cases" on page 33.
As of June 2009, Google App Engine appears to be limited to a heap size of 100MB. (See the article "The Limitations of Google App Engine" at http://gregluck.com/blog/?s=limitations 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 for Google App Engine

Make sure the following elements are commented out:

- <diskStore path="/path/to/store/data"/>
- <cacheManagerPeerProviderFactory class= ../>
- <cacheManagerPeerListenerFactory class= ../>

Within each cache element, ensure that:

- overFlowToDisk and diskPersistent are omitted
- persistence strategy=none
- no replicators are added
- there is no bootstrapCacheLoaderFactory

- there is no Terracotta configuration

Use the 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
       maxEntriesLocalHeap="10000"
       eternal="false"
       timeToIdleSeconds="120"
       timeToLiveSeconds="120"
       memoryStoreEvictionPolicy="LRU">
       <persistence strategy="none"/>
    </defaultCache>
<!--Example sample cache-->
    <cache name="sampleCache1"
      maxEntriesLocalHeap="10000"
      maxEntriesLocalDisk="1000"
      eternal="false"
      timeToIdleSeconds="300"
      timeToLiveSeconds="600"
      memoryStoreEvictionPolicy="LFU"
       />
</ehcache>
```

Use Cases

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 unlimited size of memcachedg. 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.

Using Distributed Caching

Configure all notifications in CacheEventListener to proxy through 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.

Using 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 the *Ehcache Web Cache User Guide* 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.

6 Using Ehcache with Tomcat

About Using Ehcache with Tomcat

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.

Tomcat Issues and Best Practices

Problem rejoining a cluster after a reload

If I restart/reload a web application in Tomcat that has a CacheManager that is part of a cluster, the CacheManager is unable to rejoin the cluster. If I set logging for `net.sf.ehcache.distribution` to FINE I see the following exception:

```
FINE: Unable to lookup remote cache peer for ....
Removing from peer list.
Cause was: error unmarshalling return;
nested exception is: java.io.EOFException.
```

The Tomcat and RMI class loaders do not get along that well. Move ehcache.jar to `$TOMCAT_HOME/common/lib`. This fixes the problem. This issue happens with anything that uses RMI, not just Ehcache.

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.UnmarshalException: error unmarshalling arguments;
nested exception is: java.net.MalformedURLException: no protocol: Files/Apache.
```

What is going on? This issue occurs to any RMI listener started on Tomcat whenever Tomcat has spaces in its installation path. It can be worked around in Tomcat. 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 occurring. 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.

7 Using Ehcache with JDBC

About JDBC Caching

Ehcache can easily be combined with your existing JDBC code. Whether you access JDBC directly, or use a Data Access Object/Data Access Layer (DAO/DAL), 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.

Adding JDBC caching to DAO/DAL

If your application already has a Data Access Object/Data Access Layer (DAO/DAL), 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 such as `setCache(Cache cache)`. Configure the cache as a bean in your Spring or Guice config, and then use 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/DAL 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
- cache-as-sor

You can read more about these in "Cache Usage Patterns" in the *Ehcache Developer Guide*.

Sample Code

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 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. It doesn't really 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
  .FasterLazyConnectionDataSourceProxy">
  <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>
```

8 Using Ehcache with OpenJPA

Installation and Configuration

Installation

Ehcache easily integrates with the OpenJPCA persistence framework from Apache.

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 http://ehcache.org/downloads/catalog.

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 as shown in this example:

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ehcache.xsd" updateCheck="false"
monitoring="autodetect"
dynamicConfig="true" name="TestCache">
<diskStore path="/path/to/store/data"/>
<defaultCache
    maxEntriesLocalHeap="10000"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    memoryStoreEvictionPolicy="LRU">
    <persistence strategy="localTempSwap"/>
</defaultCache>
<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>
```

The 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 OpenJPA is using Ehcache, 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, see the Apache OpenJPA project at http://openjpa.apache.org/builds/1.0.2/apache-openjpa-1.0.2/docs/manual/ref_guide_caching.html.

9 Using Ehcache with Grails

About Using Ehcache with Grails

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 customize 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="/path/to/store/data"/>
    <cacheManagerEventListenerFactory class="" properties=""/>
    <defaultCache
       maxEntriesLocalHeap="10000"
       eternal="false"
       timeToLiveSeconds="120">
       <persistence strategy="none"/>
    </defaultCache>
    <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>
```

Using the 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. For more information, see the Springcache Plugin (part of the Grails project) at http://grails.org/plugin/springcache.

Using Web Sessions with Grails

Clustering is handled by the Terracotta Web Sessions product. See the article, "Clustering A Grails App with Terracotta" at http://gquick.blogspot.com/2010/03/clustering-grails-app-with-terracotta.html, for information about how to use Web Sessions with Grails and Tomcat.

10 Using Ehcache with GlassFish

Tested Versions of GlassFish

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.

Deploying the Sample Application

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. For more information, see the Glassfish quick-start guides at https://glassfish.java.net/downloads/quickstart/index.html.

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).

11 Using Ehcache with JSR107

About Ehcache Support for JSR107

Information on Ehcache support of JSR107 is available on github at https://github.com/jsr107/ehcache-jcache.