Hibernate 4 Second Level Caching With EHCache

To use EHCache library with Hibernate 4 we should add following dependencies:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>4.3.8.Final</version>
</dependency>
<dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache-core</artifactId>
    <version>2.6.10</version>
</dependency>
```

In Spring we can configure **hibernate + ehcache** as follows:

```
<bean id="sessionFactory"
     class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource"/>
    <property name="packagesToScan"
              value="olyanren.java.web.telif.problemsolution.model"/>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.hbm2ddl.auto">update</prop>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <prop key="hibernate.connection.CharSet">utf8</prop>
            <prop key="hibernate.connection.characterEncoding">utf8</prop>
            <prop key="hibernate.connection.useUnicode">true</prop>
            <prop key="hibernate.show_sql">true</prop>
            <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">
                org.hibernate.cache.ehcache.EhCacheRegionFactory</prop>
                <prop key="net.sf.ehcache.configurationResourceName">/ehcache.xml</prop>
            </props>
        </property>
    </bean>
```

`packagesToScan` property is used for Hibernate entity classes. By using this property, we get rid of classic entity definition which uses `<mapping>` element.

First level cache is implemented by Hibernate Framework. However, second level cache is implemented by some third party jars such as ehcache. After Hibernate 4, ehcache became **default** second level cache of Hibernate.

1. **hibernate.cache.use_second_level_cache** is used to enable second level cache, we should set `hibernate.cache.use_second_level_cache` property value to true, default is false.
2. **hibernate.cache.use_query_cache** is used to enable query caching, so we should set `hibernate.cache.use_query_cache` property value to true.
3. **hibernate.cache.region.factory_class** is used to define the **Factory** class for Second level caching.

   EhCache will ensure that all instances of **SingletonEhCacheRegionFactory** use the same actual CacheManager internally, no matter how many instances of **SingletonEhCacheRegionFactory** you create, making it a crude version of the Singleton design pattern.

   The plain **EhCacheRegionFactory**, on the other hand, will get a new CacheManager every time.

   If you have two Hibernate session factories in Spring, each using their own instance of **SingletonEhCacheRegionFactory**, then they're actually going to end up sharing a lot of their cache state, which may account for your problem.

This isn't really a good match for Spring, where the singletons are supposed to be managed by the container. If you use **EhCacheRegionFactory**, then you'll likely get more predictable results. I suggest giving it a go and see how you get on.

If you are using Hibernate 3, corresponding classes will be **net.sf.ehcache.hibernate.EhCacheRegionFactory** and **net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory.**

4. **net.sf.ehcache.configurationResourceName** is used to define the EHCache configuration file location, it is optional parameter and if it's not used, EHCache will try to find **ehcache.xml** file in the classpath. Therefore, we should create **ehcache.xml** file in the classpath. If you're using Maven, you can add this file into **resources** folder.

Note: If you encounter `net.sf.ehcache.CacheException: Another unnamed CacheManager already exists in the same VM. Please provide unique names for each CacheManager in the config or do one of following:|1. Use one of the CacheManager.create() static factory methods to reuse same CacheManager with same name or create one if necessary|2. Shutdown the earlier cacheManager before creating new one with same name.|The source of the existing CacheManager is: URLConfigurationSource [url=file:/....]` exception message, you can use `org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory`

ehcache.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="ehcache.xsd" updateCheck="true"
        monitoring="autodetect" dynamicConfig="true">
    <diskStore path="java.io.tmpdir/ehcache" />
    <defaultCache maxEntriesLocalHeap="10000" eternal="false"
                  timeToIdleSeconds="120" timeToLiveSeconds="120" diskSpoolBufferSizeMB="30"
                  maxEntriesLocalDisk="10000000" diskExpiryThreadIntervalSeconds="120"
                  memoryStoreEvictionPolicy="LRU" statistics="true">
        <persistence strategy="localTempSwap"/>
    </defaultCache>
    <cache name="olyanren.java.cache.ehcache.model.Admin"
          maxElementsInMemory="500"
          eternal="true"
          timeToIdleSeconds="0"
          timeToLiveSeconds="100"
          overflowToDisk="false"
           />
    <cache
       name="org.hibernate.cache.StandardQueryCache"
       maxEntriesLocalHeap="5"
       eternal="false"
       timeToLiveSeconds="120">
        <persistence strategy="localTempSwap"/>
    </cache>
    <cache name="org.hibernate.cache.spi.UpdateTimestampsCache"
          maxEntriesLocalHeap="5000" eternal="true">
        <persistence strategy="localTempSwap"/>
    </cache>
</ehcache>
```

diskStore is used when cached objects to be stored in memory exceed maxEntriesLocalHeap value, then the objects which cannot be stored into memory will be saved **diskStore** path.

maxEntriesLocalHeap specifies how many entity which is serializable or not will be saved into memory.

eternal is used to decide cached entities life will be eternal or not. **Notice** that when set to "true", overrides **timeToLive** and **timeToIdle** so that no expiration can take place.

diskExpiryThreadIntervalSeconds sets the interval between runs of the expiry thread. Setting diskExpiryThreadIntervalSeconds to a low value is not recommended. It can cause excessive DiskStore locking and high CPU utilization. The default value is 120 seconds. If a cache's DiskStore 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.

maxEntriesLocalDisk is used to decide how many entity will be stored in the local disk when overflow is happend. This property works with **localTempSwap** option.

localTempSwap strategy allows the cache to overflow to disk during cache operation, providing an extra tier for cache storage. This disk storage is **temporary and is cleared after a restart.** If the disk store path is not specified, a default path is used, and the default will be auto-resolved in the case of a conflict with another CacheManager. The TempSwap DiskStore creates a data file for each cache on startup called **"<cache_name>.data"**.

timeToIdleSeconds enables cached object to be kept in as long as it is requested in periods shorter than timeToIdleSeconds.

timeToLiveSeconds will make the cached object be invalidated after that many seconds irregardless of how many times or when it was requested.

Let's say that `timeToIdleSeconds = 3`. Object will be invalidated if it hasn't been requested for 4 seconds.

If `timeToLiveSeconds = 90` then the object will be removed from cache after 90 seconds even if it has been requested few milliseconds in the 90th second of its short life.

StandardQueryCache class is used to store any query results. However setting this is not enough. Also we should use **Query**#setCachable() method as follows:

```
1     Query employeeTaskQuery = currentSession().createQuery(
2                "select a.username, a.password,ar.adminRoleId from Admin " +
3                        "a inner join a.adminRole ar where a.username=:username");
4        employeeTaskQuery.setParameter("username", username);
5        employeeTaskQuery.setCacheable(true);
```

Also, if you want to set **cacheMode** and **cacheRegion,** you can use **setCacheMode()** method and **setCacheRegion()** methods, respectively. Cache region can be defined in the **ehcache.xml** file as follows:

```
1   <cache name="admin" maxEntriesLocalHeap="10000" eternal="false"
2          timeToIdleSeconds="5" timeToLiveSeconds="10">
3           <persistence strategy="localTempSwap" />
4       </cache>
```

Then, we use `setCacheRegion("admin");` method.

Admin.java

```
1   @Entity
2   @org.hibernate.annotations.Cache(usage=CacheConcurrencyStrategy.READ_WRITE)
3   public class Admin {
4        @Id
5        @GeneratedValue(strategy= GenerationType.AUTO)
6        private int adminId;
7        @NotEmpty
8        @Size(min = 4, max =20)
9        @Column(length = 20,unique = true)
10       private String username;
11  }
```

As seen above Hibernate entity, we used `@org.hibernate.annotations.Cache` annotation with **READ_WRITE** concurrency strategy. Also, we declared this entity in the ehcache.xml file.

There are four strategies we can use:

Read-only: Useful for data that is **read frequently but never updated** (e.g. referential data like Countries). It is simple. It has the best performances of all (obviously).

Read/write: Desirable if your data **needs to be updated**. But it doesn't provide a SERIALIZABLE isolation level, phantom reads can occur (you may see at the end of a transaction something that wasn't there at the start). It has more overhead than read-only.

Nonstrict read/write: Alternatively, if it's unlikely two separate transaction threads could update the same object, you may use the nonstrict–read–write strategy. It has less overhead than read-write. This one is useful for data that are **rarely updated**.

Transactional: If you need a fully transactional cache. Only **suitable in a JTA** environment.

You can read **Hibernate documentation** for more information.

Evict Second Level Cache

To remove completely second level cache, we can use following codes:

```
1   @Autowired
2   private SessionFactory sessionFactory;
3   public void evictAll() {
4        SessionFactory sf = currentSession().getSessionFactory();
5        Cache cache = sf.getCache();
6        cache.evictQueryRegions();
7        cache.evictDefaultQueryRegion();
8        cache.evictCollectionRegions();
9        cache.evictEntityRegions();
10  }
11  protected Session currentSession() {
12       return sessionFactory.getCurrentSession();
13  }
```

If we want to **evict** single entity, all entities or collections from cache, we can use other **evict** methods as follows:

```
1   SessionFactory sf = currentSession().getSessionFactory();
2   cache.evictEntity(Cat.class, catId); //evict a particular Cat
3   cache.evictEntityRegion(Cat.class); //evict all Cats
4   cache.evictCollection("Cat.kittens", catId); //evict a particular collection of kittens
5   cache.evictCollectionRegion("Cat.kittens"); //evict all kitten collections
```

First Level Cache

Hibernate first level cache is session specific, so when calling same entity data in same session, there will be no hit to database. If you update the entity in the same session and calling it, we will see old data because this is default behavior of the first level cache. However in other session query we will see updated data.

We can use session **evict()** method to remove a single object from the hibernate first level cache.

We can use session **clear()** method to **clear** the cache i.e delete all the objects from the cache.

We can use session **contains()** method to check if an object is present in the hibernate cache or not, if the object is found in cache, it returns true or else it returns false.
Since hibernate cache all the objects into session first level cache, while running bulk queries or batch updates it's necessary to **clear** the cache at certain intervals to avoid memory issues.

@Transactional Annotation

If this is the transaction boundary for your service layer then Hibernate will create a new Session, meaning there's nothing in the first level cache. If you try to call the **findOne()** method twice within the same service method, the second call will fetch the entity from the cache.

Successive service method calls (e.g. **getEntity**) always end up with a new Hibernate Session, so a fresh entity is loaded from the database.

If you employ a 2nd level cache and activate it for this entity, then Hibernate will always hit the cache first and fallback to database loading, on a cache miss.

Result

EHCache is default second level cache implementation of Hibernate from version 4. Therefore, If you want to cache entity classes and sql query results, you can use EHCache library. You can read similar tutorial from **this** link.