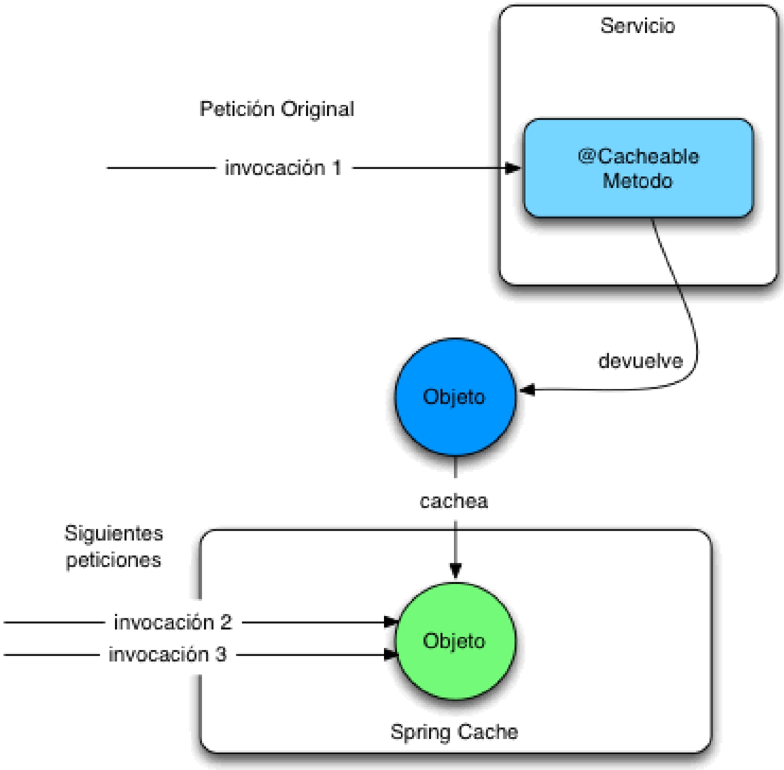


Spring Cache Memory

Design and Testing II

Group 6

Ángel Delgado Luna

Belen Garrido López

Ezequiel Portillo Jurado

Alejandro Rodríguez Díaz

Mª de Gracia Piñero Pastor

Index

Foundations

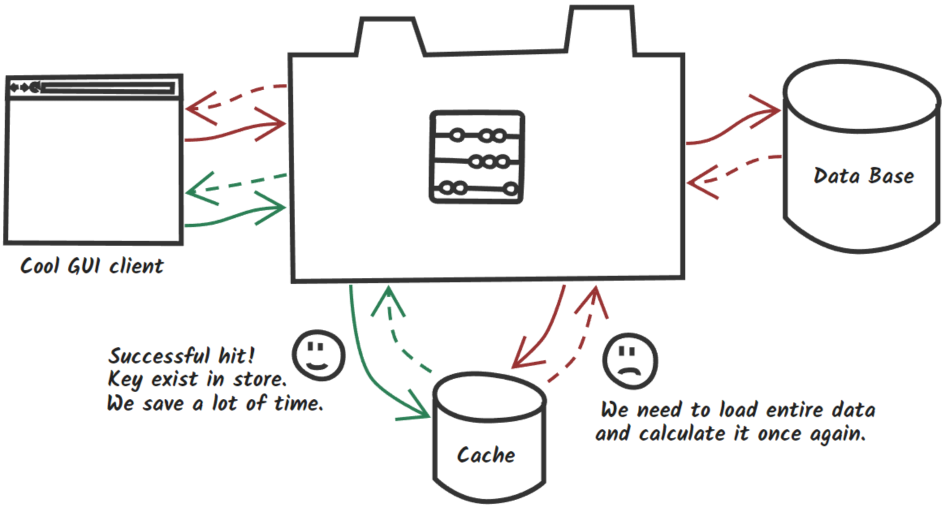
A big data amount is loaded every time by users when listing is requested. That´s why some web information systems has a special volatile memory which lets getting a list of data faster than directly requesting to database.

In our project, this is implemented in two use cases:

- Customisation System.
- Proclaims

The first is not too logical according to the said in the first paragraph. But we must considered a little detail: System parameters are not modified periodically. Every time our system is loaded in any of its sections, cache first acts if any of its parameters get modified.

The second one loads different kind of proclaims. Every time a listing query is done, a call to database is produced. To avoid overflows over the database, this web information system is provided of a little cache that only gets directly modified when a creation or edition is done. Also when this creation or edition is made, the database saves the proclaim.



Configuration

First of all, we must put into our pom.xml the following dependency:

```
<!-- EHCache -->

<dependency>
        <groupId>net.sf.ehcache</groupId>
        <artifactId>ehcache</artifactId>
        <version>2.10.2.2.21</version>
</dependency>
```

And these lines of code in our data.xml:

```
<bean id="cacheManager" class="org.springframework.cache.ehcache.EhCacheCacheManager"
                    p:cacheManager-ref="ehcache"/>
```

Second, there are two necessary configuration files we must put:

- ehcache.xml

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNamespaceSchemaLocation="ehcache.xsd"
             updateCheck="false">
           <diskStore path="java.io.tmpdir"/>

           <cache name="cust"
              maxElementsInMemory="10"
              eternal="true"
              overflowToDisk="false"
              memoryStoreEvictionPolicy="LRU"/>

           <cache name="proclaims"
              timeToLiveSeconds="40"
              maxElementsInMemory="100"
              overflowToDisk="false"
              maxElementsOnDisk="1000"
              diskPersistent="false"
              timeToIdleSeconds="1"
              memoryStoreEvictionPolicy="LRU"/> </ehcache>
```

Every time we want to use a cache system with this technology, it´s compulsory to write the following lines:

```
<cache name="cust"
      maxElementsInMemory="10"
      eternal="true"
      overflowToDisk="false"
      memoryStoreEvictionPolicy="LRU"/>
```

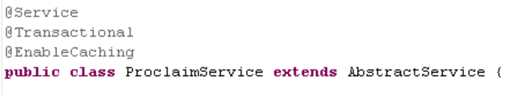
Param	Description
name	The name of the cache. This will be used later in the service (of the entity we are working with) by the annotations. This will let the objects to get saved into the correct cache memory.

maxElementsInMemory	The number of elements which the memory cache will have to save.
Eternal	This param is used when we do not want to update periodically the cache. For example, this cache only gets modified when an edition is done.
overflowToDisk	If cache gets completely filled in, all its data will automatically put into the hard drive disk.
memoryStoreEvictionPolicy	The technique used by the cache to replace the objects. In this case, it is used “Least Recently Used”.

- ehcache.xsd (This file is too long. It´s not properly to copy here. Please, have it a look into the project). This file is used for the main configuration.

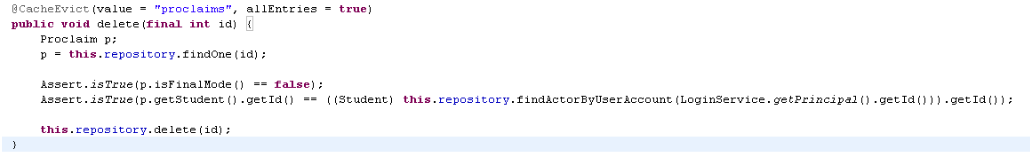
And finally, the following annotations:

- @EnableCaching. This annotation is put at the top of the service class.



```
@Service
@Transactional
@EnableCaching
public class ProclaimService extends AbstractService {
```

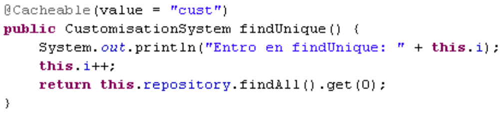
- @CacheEvict. When an object is created or edited, this annotation removes all its content and load all the data again.



```
@CacheEvict(value = "proclaims", allEntries = true)
public void delete(final int id) {
    Proclaim p;
    p = this.repository.findOne(id);

    Assert.isTrue(p.isFinalMode() == false);
    Assert.isTrue(p.getStudent().getId() == ((Student) this.repository.findActorByUserAccount(LoginService.getPrincipal().getId())).getId());

    this.repository.delete(id);
}
```

- @Cacheable. This annotation is used to load the data we want to save into the cache system.

```
@Cacheable(value = "proclaims")
public Collection<Proclaim> findNoAssigned() {
    return this.repository.findAllProclaim();
}

@Cacheable(value = "proclaims")
public Collection<Proclaim> findAllByMember() {
    Member m;
    m = (Member) this.repository.findActorByUserAccount(LoginService.getPrincipal().getId());
    return this.repository.findAllByMember(m.getId());
}

@Cacheable(value = "proclaims")
public Collection<Proclaim> findAllByMemberClosed() {
    Member m;
    m = (Member) this.repository.findActorByUserAccount(LoginService.getPrincipal().getId());
    return this.repository.findAllByMemberClosed(m.getId());
}

@Cacheable(value = "proclaims")
public Collection<Proclaim> findAllByStudent() {

    Assert.isTrue(super.findAuthority(LoginService.getPrincipal().getAuthorities(), Authority.STUDENT));
    Student m;
    m = (Student) this.repository.findActorByUserAccount(LoginService.getPrincipal().getId());

    return this.repository.findAllByStudent(m.getId());
}
```



```
@Cacheable(value = "cust")
public CustomisationSystem findUnique() {
    System.out.println("Entro en findUnique: " + this.i);
    this.i++;
    return this.repository.findAll().get(0);
}
```

- @CachePut. This annotation is used when creating or editing an object. The main difference with @CacheEvict, is the param “key”. This param always looks for the id of the object that the save method receives. If it is found, this will get the modified. If not, the object is directly created.