DP2 – PROFILING

US 011 Homeless Pet Management

Yoana Dimitrova Penkova
yoadimpen@alum.us.es

The US that was chosen for the profiling task was US 011 Homeless Pet Management because it was the one with the worst performance during the tests performed with Gatling. Getting deeper into the source of these problems I chose Glowroot to see what exactly was causing them.

I launched the application and performed every step of a single homeless pet creation as the specified scenario in Gatling scripts. Actually, the problem wasn't the inserting at all, it was the listing of the homeless pets (view from where I can access the new pet form). The main problem was that there were several queries that were getting executed needlessly because in the listing the info obtained from them wasn't used at all. It was a typical N+1 query problem.

Before performing any optimization refinements whenever the query for listing the pets was executed, several queries for their visits, interventions, rehabs and adoptions were getting executed as well and we don't need that. We can see that in the following screenshots:

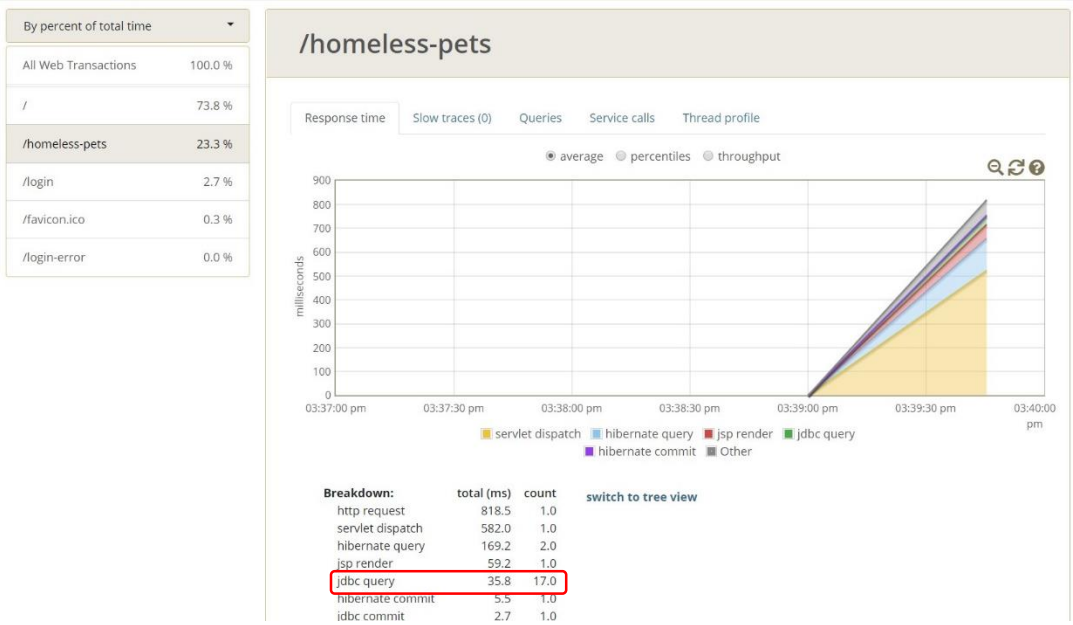

Figure 1 Response time before optimization

In Figure 1 we can see how long the queries of the listing view take, exactly 35.8 ms. When we look into the different queries that get executed we see what was mentioned before:

Figure 2 Queries before optimization

We can fix this by either lazy loading or using a join fetch. In this case it's better to use the lazy loading because we won't even use that information in the listing, we don't need it. Therefore, we need to modify the Pet entity as follows:

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "pet", fetch = FetchType.LAZY)
private Set<Visit>          visits;

@OneToMany(cascade = CascadeType.ALL, mappedBy = "pet", fetch = FetchType.LAZY)
private Set<Intervention>   interventions;

@OneToMany(cascade = CascadeType.ALL, mappedBy = "pet", fetch = FetchType.LAZY)
private Set<Rehab> rehabs;

@OneToMany(cascade = CascadeType.ALL, mappedBy = "pet", fetch = FetchType.LAZY)
private Set<Adoption> adoptions;
```

Figure 3 Optimization modifications

After this little change we can instantly see the difference in the time taken by the queries execution:

Figure 4 Response time after optimization

And also in the queries section (no unnecessary queries executed):

	Total time (ms)	Total count	Avg time (ms)	Avg rows
`select pet0_.id as id1_6_, pet0_.name as name2_6_, pet0_.birth_date as birth_da3_6_, pet0...`	2.2	1	2.2	3.0
`select pettype0_.id as id1_11_, pettype0_.name as name2_11_ from types pettype0_ order by...`	2.1	1	2.1	6.0

Figure 5 Queries after optimization

After this optimization I thought “why don’t we cache the listing?”. To perform that task we need to make little adjustments in the code first.

As step 1 we need to add the following annotations to certain methods in PetService:

```
        //This method allows us to find all homeless pets
        @Cacheable("homelessPets")
        public List<Pet> findHomelessPets() throws DataAccessException {
            return this.petRepository.findHomelessPets();
        }
```

Figure 6 Cacheable annotation

```
@Transactional(rollbackFor = DuplicatedPetNameException.class)
@CacheEvict(cacheNames = "homelessPets", allEntries = true)
public void savePet(final Pet pet) throws DataAccessException, DuplicatedPetNameException {
    Pet otherPet = new Pet();

    if(pet.getOwner() != null) {
        otherPet = pet.getOwner().getPetwithIdDifferent(pet.getName(), pet.getId());
    } else {
        otherPet = null;
    }

    if (pet.getOwner() != null && StringUtils.hasLength(pet.getName())
            && otherPet != null && otherPet.getId() != pet.getId()) {
        throw new DuplicatedPetNameException();
    } else {
        this.petRepository.save(pet);
    }
}
```

Figure 7 CacheEvict annotation

After that, as Step 2, we need to add two classes to the configuration package:

CacheLogger.java

```
1  package org.springframework.samples.petclinic.configuration;
2
3  import org.ehcache.event.CacheEvent;
6
7  public class CacheLogger implements CacheEventListener<Object, Object> {
8
9      private final org.slf4j.Logger LOG = LoggerFactory.getLogger(CacheLogger.class);
10
11     @Override
12     public void onEvent(CacheEvent<? extends Object, ? extends Object> event) {
13         LOG.info("Key: {} | EventType: {} | Old value: {} | New value: {}",
14                 event.getKey(), event.getType(), event.getOldValue(), event.getNewValue());
15     }
16
17 }
18
```

Figure 8 CacheLogger

A class that indicates us in the logs what is currently in the cache.

CacheConfiguration.java

```java
package org.springframework.samples.petclinic.configuration;

import org.springframework.cache.annotation.EnableCaching;

@Configuration
@EnableCaching
public class CacheConfiguration {

}

```

Figure 9 CacheConfiguration

As step 3 we have to add an xml file like this one in src/main/resources:

ehcache3.xml

```xml
<config
        xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
        xmlns='http://www.ehcache.org/v3'
        xsi:schemaLocation="
            http://www.ehcache.org/v3
            http://www.ehcache.org/schema/ehcache-core-3.7.xsd">

    <!-- Persistent cache directory -->
    <!--<persistence directory="spring-boot-ehcache/cache" />-->

    <!-- Default cache template -->
    <cache-template name="default">
        <expiry>
            <ttl unit="seconds">120</ttl>
        </expiry>
        <listeners>
            <listener>
                <class>org.springframework.samples.petclinic.configuration.CacheLogger</class>
                <event-firing-mode>ASYNCHRONOUS</event-firing-mode>
                <event-ordering-mode>UNORDERED</event-ordering-mode>
                <events-to-fire-on>CREATED</events-to-fire-on>
                <events-to-fire-on>EXPIRED</events-to-fire-on>
                <events-to-fire-on>EVICTED</events-to-fire-on>
            </listener>
        </listeners>
        <resources>
            <heap>1000</heap>
        </resources>
    </cache-template>

    <cache alias="homelessPets" uses-template="default">
        <key-type></key-type>
        <value-type>java.util.Collection</value-type>
    </cache>
</config>
```

Design Source

Figure 10 ehcache3.xml

And as step 4, indicate in the properties file that we will be using a cache file as follows:

```
15
16 spring.cache.jcache.config=classpath:ehcache3.xml
17
```

After this, if we launch the application again to check if it works there's a notable difference. The following screenshot is from the first time we access the listing:

	Total time (ms)	Total count	Avg time (ms)	Avg rows
select pettype0_.id as id1_11_, pettype0_.name as name2_11_ from types pettype0_ order by...	2.6	1	2.6	6.0
select pet0_.id as id1_6_, pet0_.name as name2_6_, pet0_.birth_date as birth_da3_6_, pet0...	2.2	1	2.2	3.0

Figure 11 Queries first time with cache

And the next one is from the second time we access the listing:

	Total time (ms)	Total count	Avg time (ms)	Avg rows
select pet0_.id as id1_6_, pet0_.name as name2_6_, pet0_.birth_date as birth_da3_6_, pet0...	2.2	1	2.2	3.0
select pettype0_.id as id1_11_, pettype0_.name as name2_11_ from types pettype0_ order by...	5.3	2	2.7	6.0

Figure 12 Queries second time with cache

We can see that it gets executed only the first because the second time it gets the information from the cache, as it's shown in the logs:

```
2020-05-22 16:28:16.540  INFO 15904 --- [e [_default_]-0] o.s.s.p.configuration.CacheLogger        : Key: SimpleKey [] | EventType
: CREATED | Old value: null | New value: [Tucker, Lekay, Miss]
```

Figure 13 Cache

And more or less this makes this user story much more optimized.