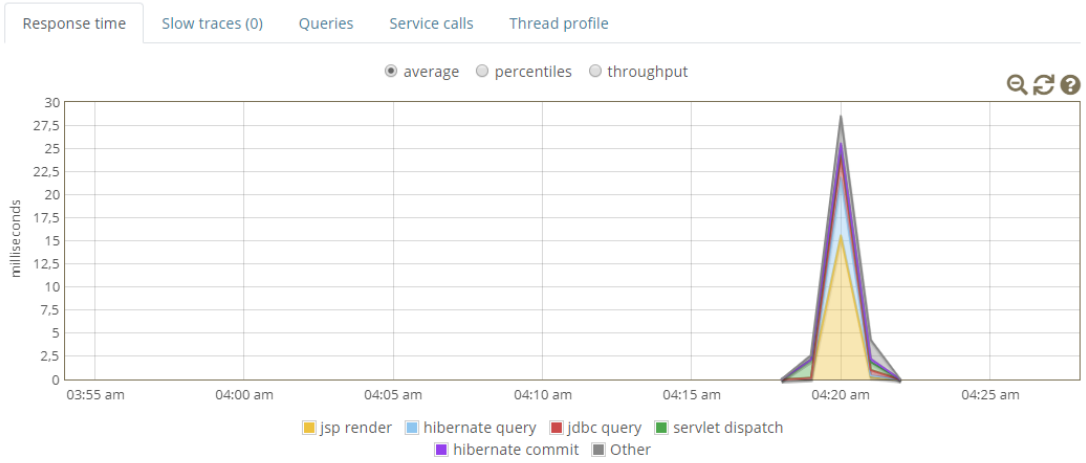
DP2 - PROFILING

US 001 Medicine creation

Manuel Cañizares Juan
mancanjua@alum.us.es

The profiling was done with the US 001 that is related to the creation and show of a medicine. This US is related to the US 002, which feature is about listing all the available medicines in the system. Due to the way is the navigation through the views done, to create a new medicine is mandatory to list all the medicine first.

Because of this casuistry, this is the US which has the worst performance in the load tests I have run, so I studied about its performance in Glowroot simulating about 40 users for 50 seconds, and this is what I got from.



Breakdown:	total (ms)	count
http request	18,9	1,0
servlet dispatch	10,9	0,7
jsp render	9,7	0,7
hibernate query	5,4	0,7
jdbc query	1,6	1,1
hibernate commit	0.34	0,9
jdbc commit	0.14	0,9
jdbc get connection	0.099	0,7
hibernate persist	0.027	0.07

switch to tree view

As we can see in the figure, one of the things that is drawing out the performance, are the jdbc and hibernate queries during the process. If we go in depth in those aspects, the following results are shown up:

	Total time (ms)	Total count	Avg time (ms)	Avg rows
`select medicine0_.id as id1_4_, medicine0_.name as name2_4_, medicine0_.expiration_d...`	786,1	80	9,8	9828,4
`select pettype0_.id as id1_11_, pettype0_.name as name2_11_ from types pettype0_ ord...`	112,0	360	0.31	6,0
`select username, authority from authorities where username = ?`	29,6	80	0.37	1,0
`select username,password,enabled from users where username = ?`	25,2	80	0.32	1,0
`select medicine0_.id as id1_4_0_, medicine0_.name as name2_4_0_, medicine0_.expirati...`	12,7	40	0.32	1,0
`insert into medicine (name, expiration_date, maker, type_id) values (?, ?, ?, ?)`	10,7	40	0.27	1,0

The most prominent aspect is the query to request the pet types to the database done when displaying a list of them in the creation form and the one which is done during the medicine listing process, due to the default fetch configuration. The second reason is an example of a

common problem, the N + 1 queries one, and this can be fixed changing the fetch configuration manually in the entity definition:

```
@ManyToOne(optional = false, fetch = FetchType.LAZY)
@JoinColumn(name = "type_id")
@NotNull
private PetType petType;
```

Once this change is done, we simulate again with the same parameters and we got this:

	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...	139,0	360	0.39	6,0
select username,password,enabled from users where username = ?	55,5	80	0.69	1,0
select medicine0_.id as id1_4_, medicine0_.name as name2_4_, medicine0_.expiration_dat...	40,4	80	0.51	2,4
select username, authority from authorities where username = ?	27,8	80	0.35	1,0
insert into medicine (name, expiration_date, maker, type_id) values (?, ?, ?, ?)	12,7	40	0.32	1,0
select medicine0_.id as id1_4_0_, medicine0_.name as name2_4_0_, medicine0_.expiration...	11,9	40	0.30	1,0

Although the count of queries to get the pet type has not changed at all, we can see an enormous difference in the total elapsed time to request all the medicines for the listing, which variance is of 746 ms.

Even that, the excessive number of queries to request the pet type is present yet, so in order to fix this problem, some cache configuration will be added to the project in the methods that request this info.

First, I added those entries in the *ehchache3.xml* file:

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

Then, I set the methods which will load the cache on their call:

```
@Transactional(readOnly = true)
@Cacheable("petTypes")
public Collection<PetType> findPetTypes() throws DataAccessException {
    return this.petRepository.findPetTypes();
}
```

```java
@Transactional(readOnly = true)
@Cacheable("medicines")
public Collection<Medicine> findManyMedicineByName(String name) {
    return repository.findByNameContaining(name);
}


@Transactional(readOnly = true)
@Cacheable("medicines")
public Collection<Medicine> findByPetType(PetType petType) {
    return repository.findByPetType(petType);
}
```

And finally, I added the necessary annotations to use the cached data when a request method is called:

```java
@Transactional
@CacheEvict(cacheNames = {"petTypes", "medicines"}, allEntries = true)
public void saveMedicine(Medicine medicine) {
    repository.save(medicine);
}

@Transactional
@CacheEvict(cacheNames = {"petTypes", "medicines"}, allEntries = true)
public void deleteMedicine(Medicine medicine) {
    repository.delete(medicine);
}
```

After all those changes, I ran the simulation one more time:

	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 ...	49,4	45	1,1	6,0
select username,password,enabled from users where username = ?	41,0	80	0.51	1,0
select medicine0_.id as id1_4_, medicine0_.name as name2_4_, medicine0_.expiration_date...	40,5	10	4,0	3,0
insert into medicine (id, name, expiration_date, maker, type_id) values (null, ?, ?, ?, ?)	37,0	40	0.93	1,0
select username, authority from authorities where username = ?	11,2	80	0.14	1,0
select medicine0_.id as id1_4_0_, medicine0_.name as name2_4_0_, medicine0_.expiration_...	6,3	40	0.16	1,0
select pettype0_.id as id1_11_0_, pettype0_.name as name2_11_0_ from types pettype0_ wh...	1,5	12	0.13	1,0

This time, we can notice a great difference between some total times and query executions. Specifically, there are 303 less queries to request all the pet types and 70 less queries to request all the medicines, which significantly reduces the amount and time spent executing those queries.