DP2-1920-GI-02: Sprint 4 June 1, 2020

PROFILING AND OPTIMIZATION

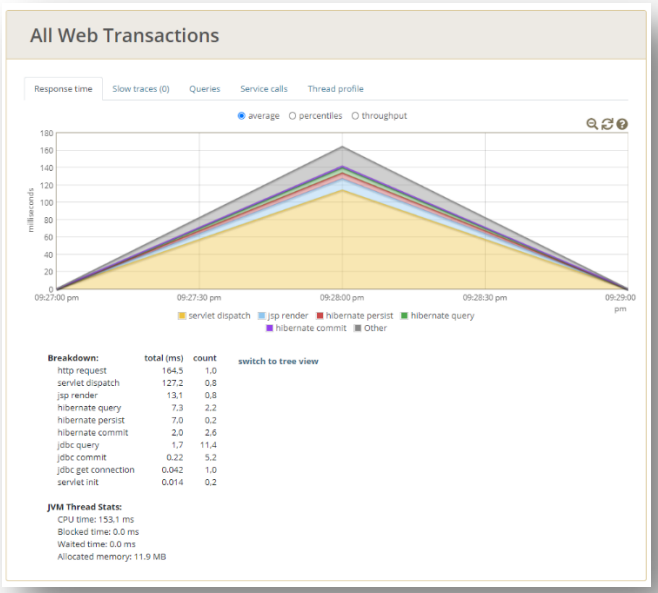
In part I of this document, we perform four different profilings using glowroot and analyze the inefficiencies detected. In part II we optimize the application by applying refactorizations based on three of those profilings.

PART I: PROFILING

Profiling 1

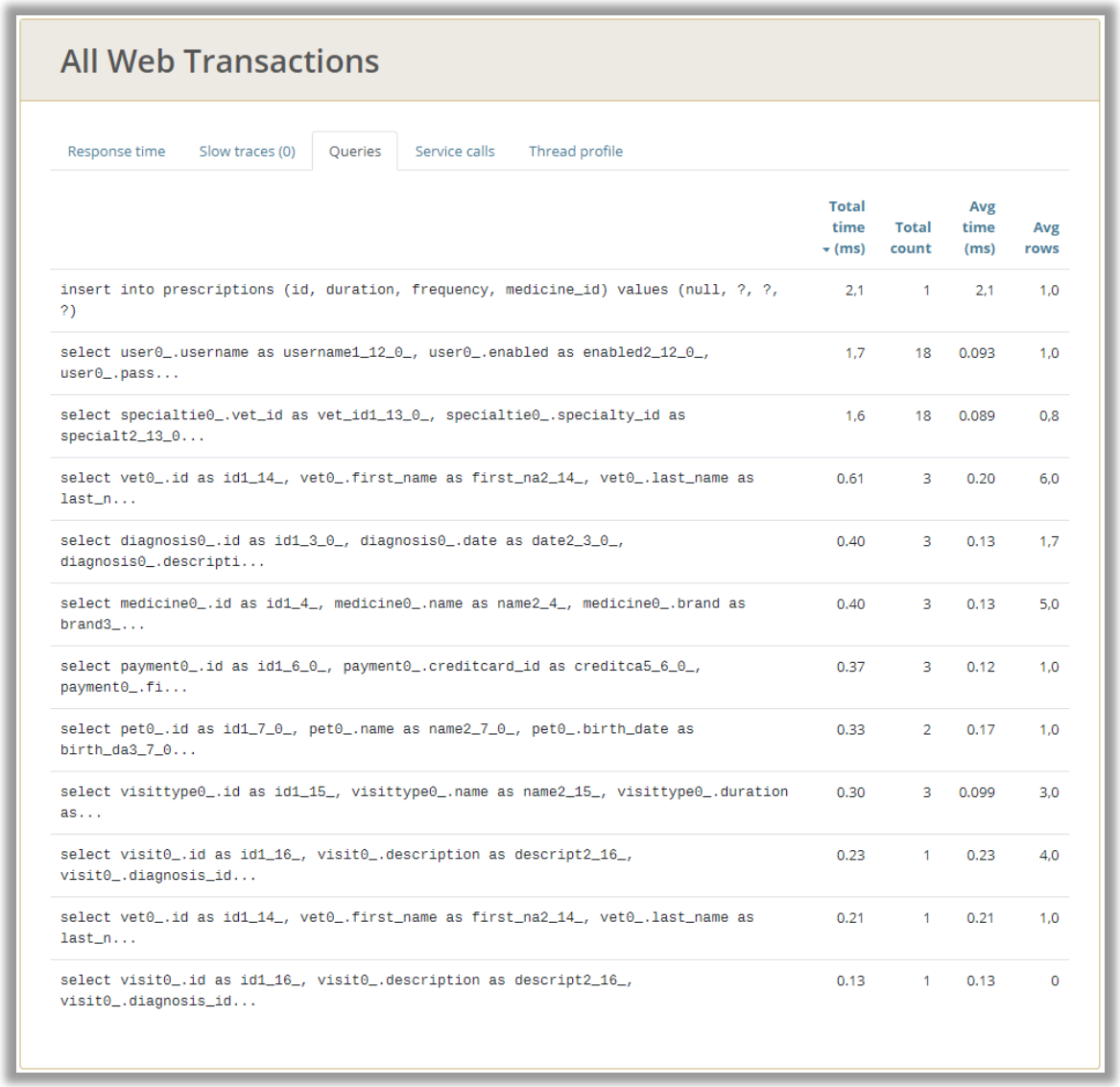
In the performance report, we identified the user story US-14 (a vet adds a new prescription to an existing diagnosis) as the bottleneck of the whole application. Therefore, we selected it for a more thorough analysis.

Using glowroot, we can visualize the proportion each stage of the process adds to the total loading time:



As can be seen in the above figure, we began to simulate the scenario at 9:27:00 pm. The peak of 160 milliseconds, one minute later, reflects the time the application took to respond to our requests.

Compared to the response time of other scenarios, including the ones in this document, this delay is extraordinarily long. In order to find out its origin, we investigate the queries performed to the database:

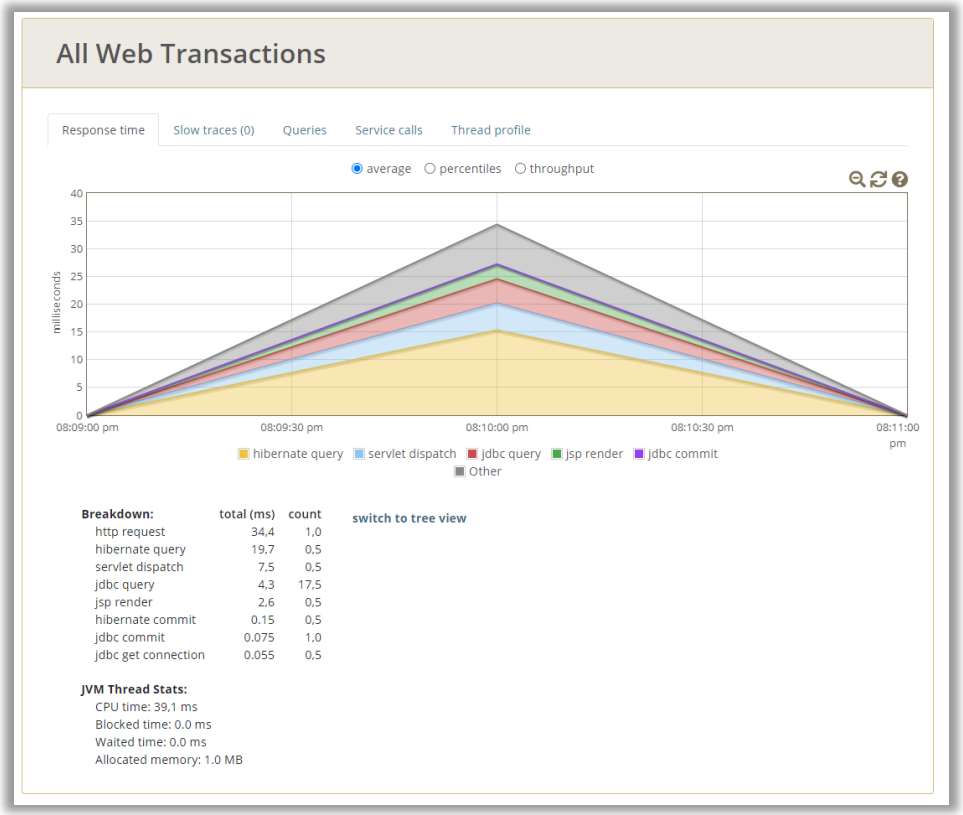


	Total time (ms)	Total count	Avg time (ms)	Avg rows
insert into prescriptions (id, duration, frequency, medicine_id) values (null, ?, ?, ?)	2,1	1	2,1	1,0
select user0_.username as username1_12_0_, user0_.enabled as enabled2_12_0_, user0_.pass...	1,7	18	0.093	1,0
select specialtie0_.vet_id as vet_id1_13_0_, specialtie0_.specialty_id as specialt2_13_0...	1,6	18	0.089	0,8
select vet0_.id as id1_14_, vet0_.first_name as first_na2_14_, vet0_.last_name as last_n...	0.61	3	0.20	6,0
select diagnosis0_.id as id1_3_0_, diagnosis0_.date as date2_3_0_, diagnosis0_.descripti...	0.40	3	0.13	1,7
select medicine0_.id as id1_4_, medicine0_.name as name2_4_, medicine0_.brand as brand3_...	0.40	3	0.13	5,0
select payment0_.id as id1_6_0_, payment0_.creditcard_id as creditca5_6_0_, payment0_.fi...	0.37	3	0.12	1,0
select pet0_.id as id1_7_0_, pet0_.name as name2_7_0_, pet0_.birth_date as birth_da3_7_0...	0.33	2	0.17	1,0
select visittype0_.id as id1_15_, visittype0_.name as name2_15_, visittype0_.duration as...	0.30	3	0.099	3,0
select visit0_.id as id1_16_, visit0_.description as descript2_16_, visit0_.diagnosis_id...	0.23	1	0.23	4,0
select vet0_.id as id1_14_, vet0_.first_name as first_na2_14_, vet0_.last_name as last_n...	0.21	1	0.21	1,0
select visit0_.id as id1_16_, visit0_.description as descript2_16_, visit0_.diagnosis_id...	0.13	1	0.13	0

As can be seen in the figure above, a total of 12 queries are made to the database. We assume that the cause of this unusually high number is the complex nature of the user story and the fact that it involves many different objects: A prescription is added by a vet to a diagnosis, which in turn is associated to a visit made by an owner for one of his or her pets.

Profiling 2

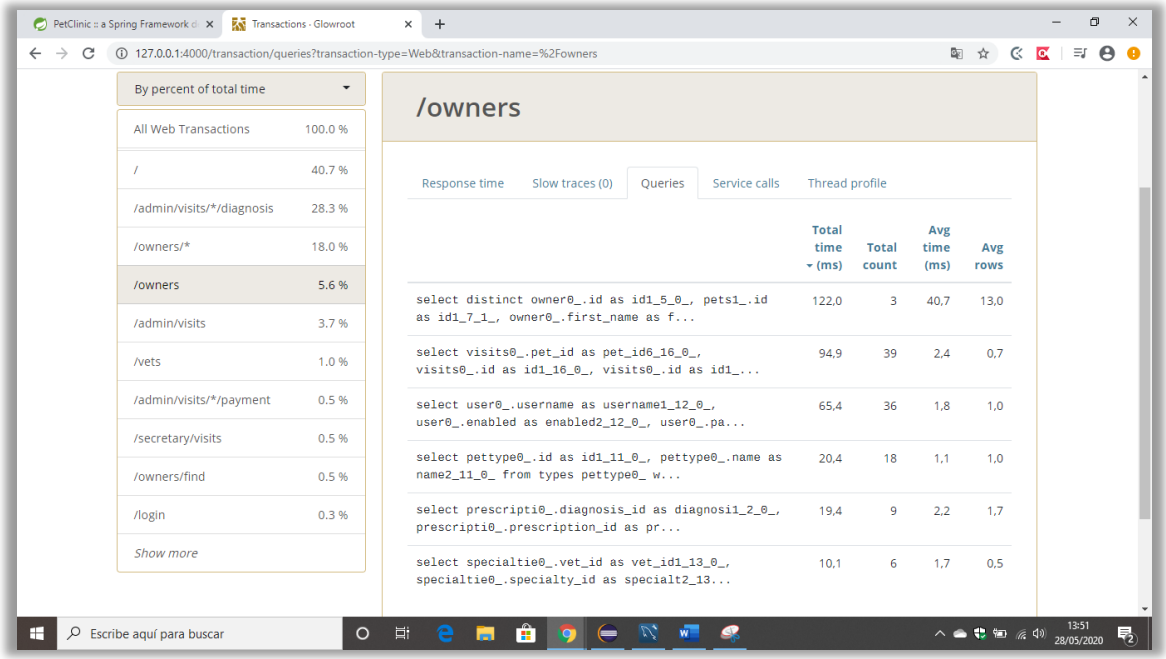
This profiling is based on a functionality that we adapted from the template project and that does not have any corresponding user story in our requirement catalogue: An admin can show a list of all the owners registered in the system.



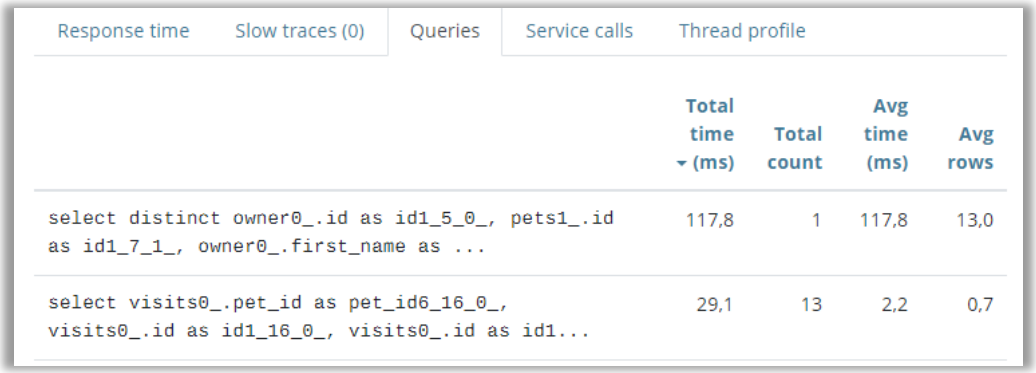
A N+1 Query problem has been detected: When the view of all owners (/owners) is loaded, all the owners and pets of each one appear. It has been detected that, in that view, for each pet that appears the visits of each one are loaded.

In our example data we have 13 pets associated with different owners, so for each pet that we have included in our database, 13 queries are made that return the visits of each pet has.

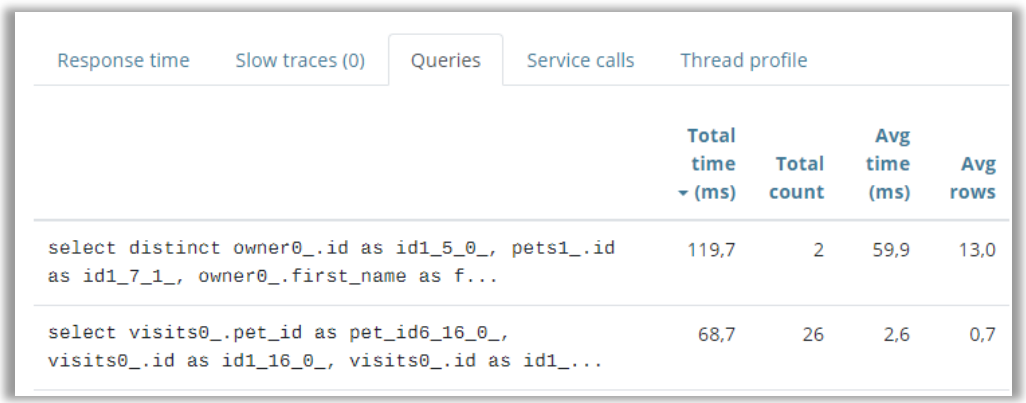
The queries that are made can be seen with:



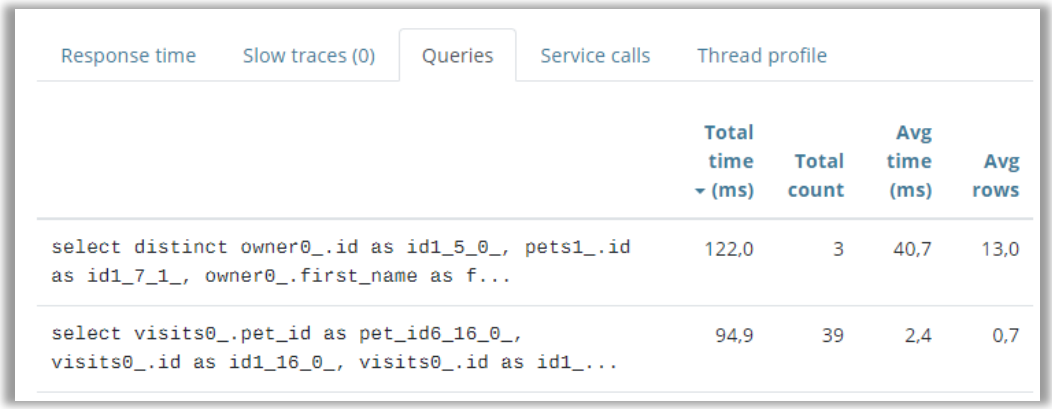
When that view is loaded once:



When that view is loaded twice :



When that view is loaded three times:

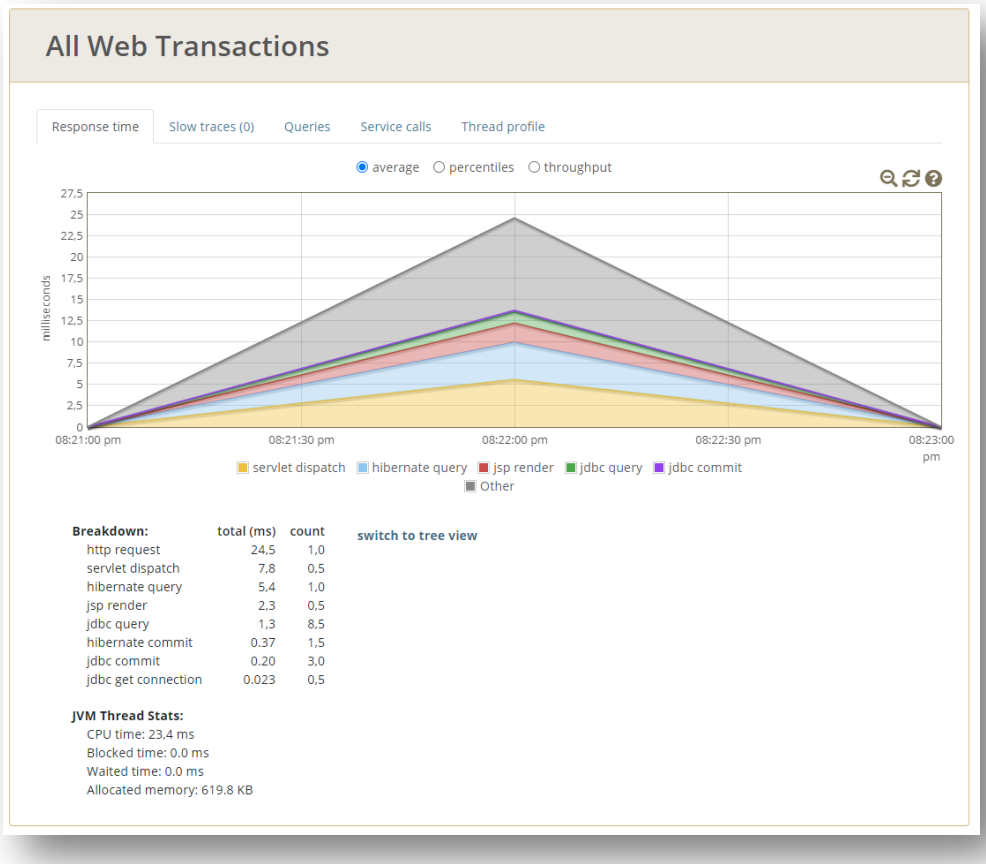


Response time | Slow traces (0) | Queries | Service calls | Thread profile

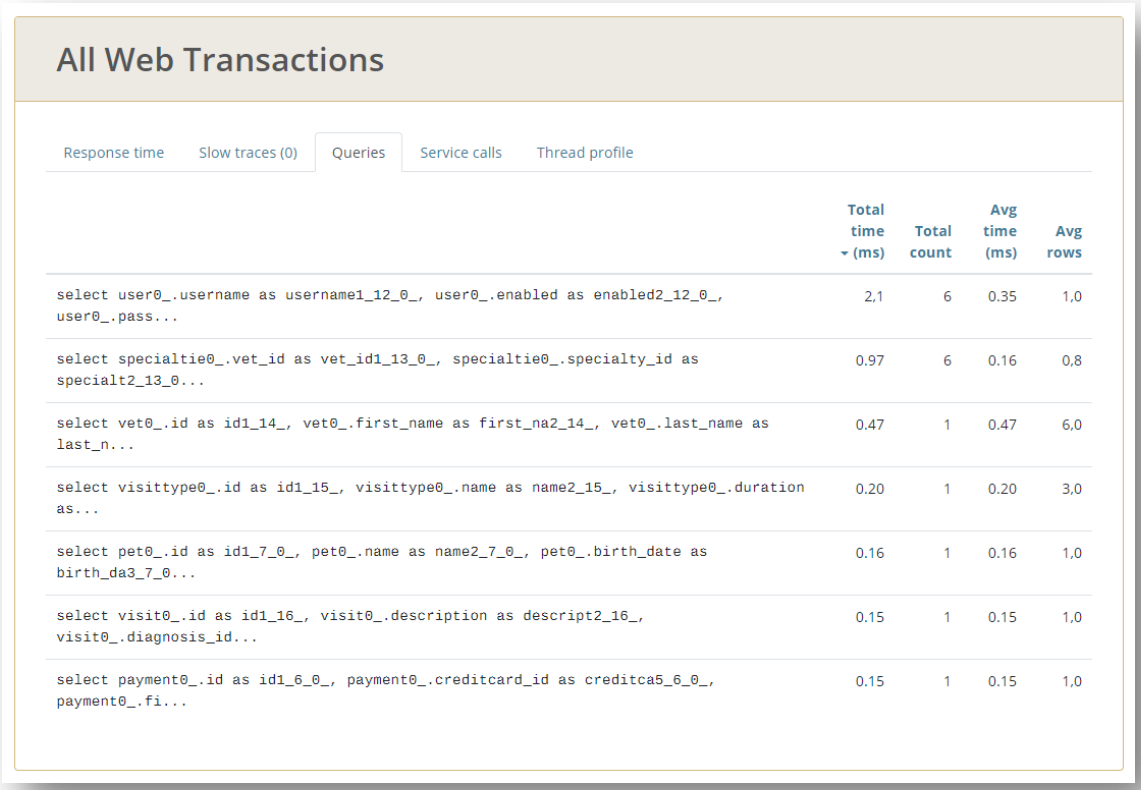
	Total time (ms)	Total count	Avg time (ms)	Avg rows
`select distinct owner0_.id as id1_5_0_, pets1_.id as id1_7_1_, owner0_.first_name as f...`	122,0	3	40,7	13,0
`select visits0_.pet_id as pet_id6_16_0_, visits0_.id as id1_16_0_, visits0_.id as id1_...`	94,9	39	2,4	0,7

Profiling 3

When accessing the view dp2.com/vet/visits/8 as a vet, as specified in the user story US-7:



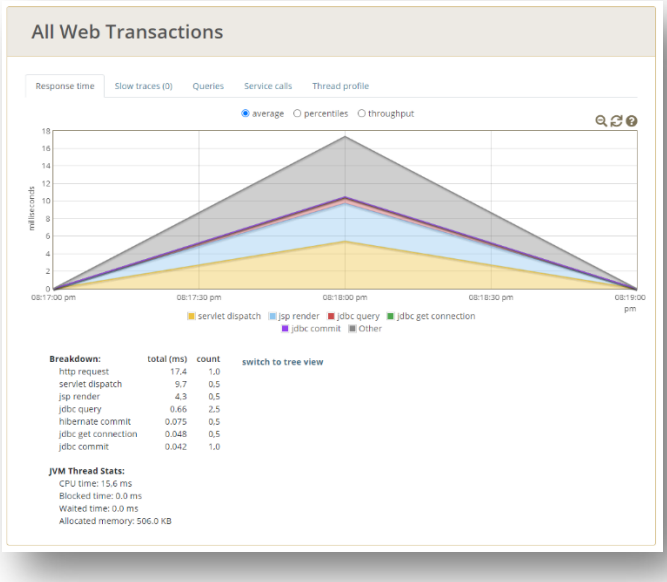
Seven queries are made to the database:



We consider this number of queries to be unnecessary and suggest using a cache in order to optimize the performance.

Profiling 4

When accessing the view dp2.com/owners/1 as an admin:



5 queries are made to the database, even though the data could be stored in a cache:

All Web Transactions

Response time | Slow traces (0) | Queries | Service calls | Thread profile

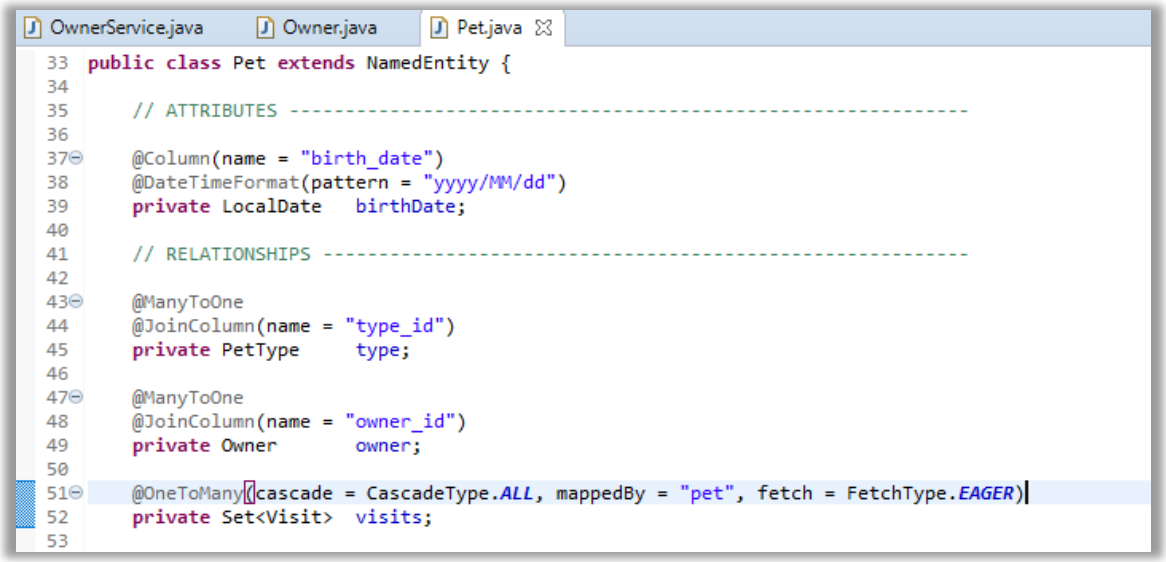
	Total time (ms)	Total count	Avg time (ms)	Avg rows
select owner0_.id as id1_5_0_, pets1_.id as id1_7_1_, owner0_.first_name as first_na2_5...	0.35	1	0.35	1,0
select visits0_.pet_id as pet_id6_16_0_, visits0_.id as id1_16_0_, visits0_.id as id1_1...	0.31	1	0.31	2,0
select pettype0_.id as id1_11_0_, pettype0_.name as name2_11_0_ from types pettype0_ wh...	0.22	1	0.22	1,0
select user0_.username as username1_12_0_, user0_.enabled as enabled2_12_0_, user0_.pas...	0.19	1	0.19	1,0
select specialtie0_.vet_id as vet_id1_13_0_, specialtie0_.specialty_id as specialt2_13_...	0.17	1	0.17	0

PART II: OPTIMIZATION BY REFACTORING

Refactoring based on profiling 2

Problem:

It has been detected in the model that the relationship of pet with visits was of type **.EAGER**, which means that whenever a pet is loaded, it's visits are loaded.



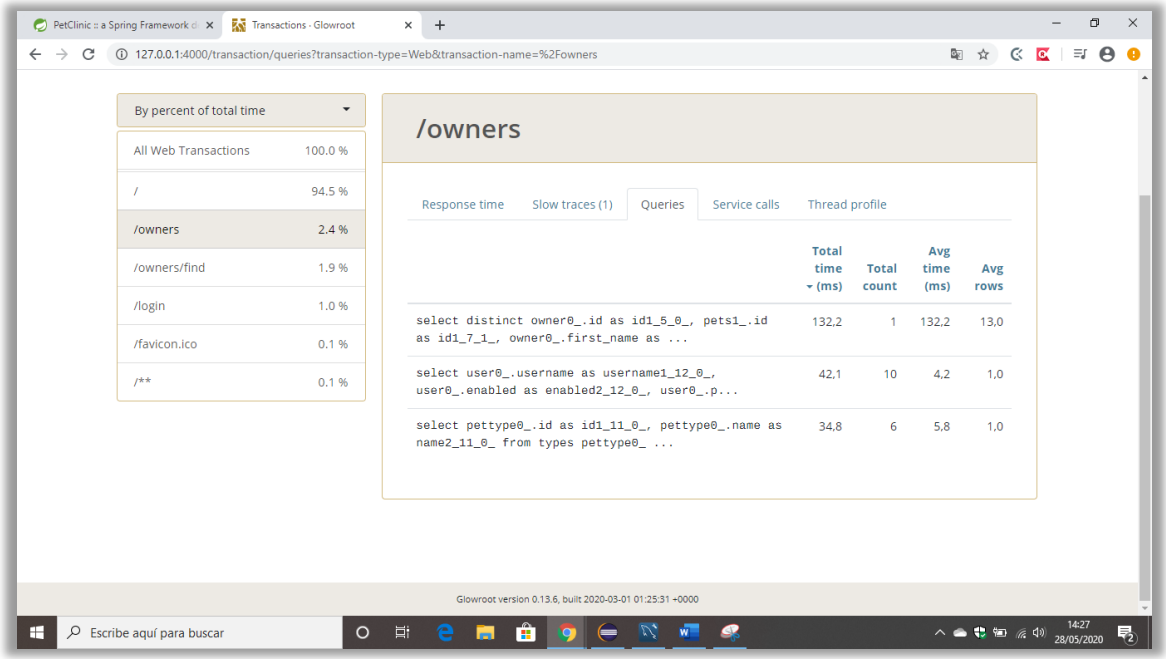
Solution:

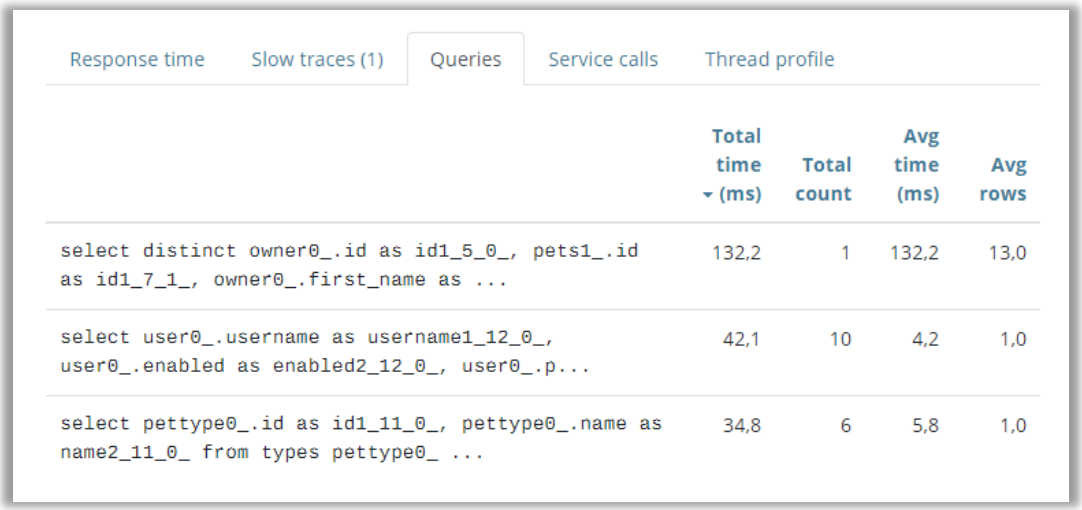
It has been changed and not the relationship pet-visits has been set to type .LAZY so that ii only loads when necessary (since visits is something we don't need in the /owners view we are talking about).

```java
public class Pet extends NamedEntity {

    // ATTRIBUTES -----------------------------------------------------------

    @Column(name = "birth_date")
    @DateTimeFormat(pattern = "yyyy/MM/dd")
    private LocalDate   birthDate;

    // RELATIONSHIPS --------------------------------------------------------

    @ManyToOne
    @JoinColumn(name = "type_id")
    private PetType     type;

    @ManyToOne
    @JoinColumn(name = "owner_id")
    private Owner       owner;

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

Effects:

In this way, now in Glowroot you can see that those N Querys that were made for each pet in our database have disappeared.





Response time | Slow traces (1) | Queries | Service calls | Thread profile

	Total time (ms)	Total count	Avg time (ms)	Avg rows
`select distinct owner0_.id as id1_5_0_, pets1_.id as id1_7_1_, owner0_.first_name as ...`	132,2	1	132,2	13,0
`select user0_.username as username1_12_0_, user0_.enabled as enabled2_12_0_, user0_.p...`	42,1	10	4,2	1,0
`select pettype0_.id as id1_11_0_, pettype0_.name as name2_11_0_ from types pettype0_ ...`	34,8	6	5,8	1,0

Refactoring based on profiling 3

Problem:

As discussed previously, queries are made to the database that could be avoided by using caches.

Solution:

We added a cache for findVisitById.

First, we added the cache configuration as explained in the video on EV:

```
1  package org.group2.petclinic.configuration;
2
3  import org.springframework.cache.annotation.EnableCaching;
5
6  @Configuration
7  @EnableCaching
8  public class CacheConfiguration {
9
10 }
```

We added a cache logger:

```java
package org.group2.petclinic.configuration;

import org.ehcache.event.CacheEvent;

public class CacheLogger implements CacheEventListener<Object, Object> {
    private final Logger LOG = LoggerFactory.getLogger(CacheLogger.class);
    @Override
    public void onEvent(CacheEvent<?, ?> cacheEvent) {
      LOG.info("Key: {} | EventType: {} | Old value: {} | New value: {}",
               cacheEvent.getKey(), cacheEvent.getType(), cacheEvent.getOldValue(),
               cacheEvent.getNewValue());
    }
  }
```

We added the ehcache3 template:

```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.group2.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="visitById" uses-template="default">
            <key-type>java.lang.Integer</key-type>
            <value-type>org.group2.petclinic.model.Visit</value-type>
        </cache>

         <cache alias="ownerById" uses-template="default">
            <key-type>java.lang.Integer</key-type>
            <value-type>org.group2.petclinic.model.Owner</value-type>
        </cache>

    </config>
```

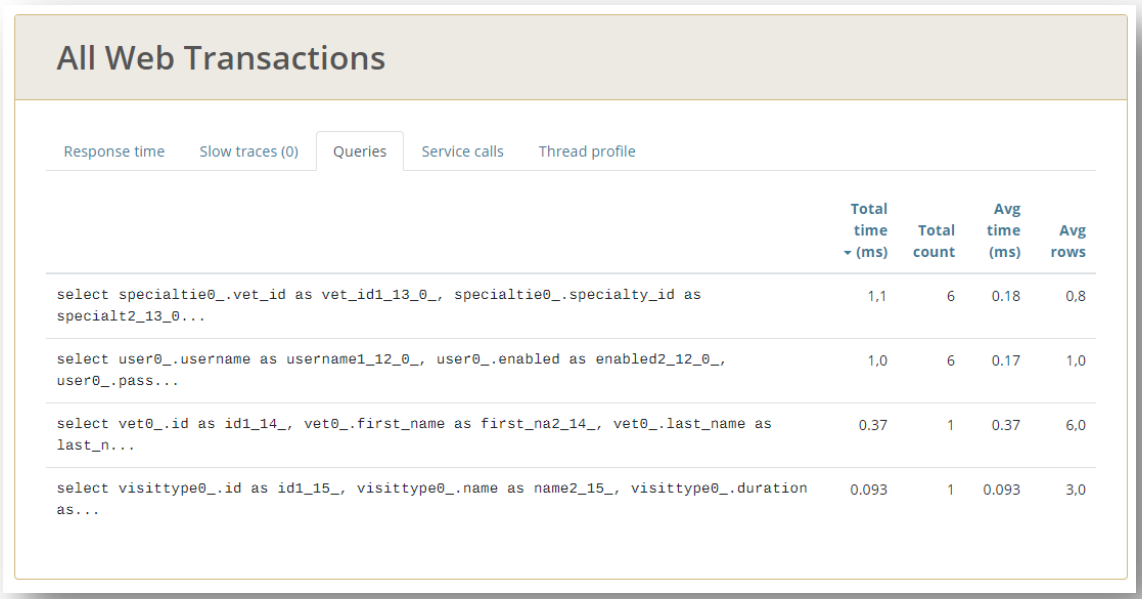
We added the necessary annotations:

```java
// FIND VISIT ---------------------------------------------------------------

@Transactional(readOnly = true)
@Cacheable("visitById")
public Visit findVisitById(int id) throws DataAccessException {
    return visitRepository.findById(id);
}
```

```java
// SAVE VISITS ------------------------------------------------------------------

@Transactional
@CacheEvict(cacheNames="visitById", allEntries=true)
public void saveVisit(final Visit visit) throws DataAccessException {
    this.visitRepository.save(visit);
}
```

Effects:

Now, 4 queries are made to the database when the view dp2.com/vet/visits/8 is loaded, while previously it was 7. With the cache, we were able to avoid 3 queries.



Refactoring based on profiling 4

Problem:

As discussed previously, queries are made to the database that could be avoided by using caches.

Solution:

We added a cache for findOwnerById.

We did not have to add the cache configuration as we already added it during the previous profiling (profiling 2).

We added the necessary annotations:

```java
@Transactional(readOnly = true)
@Cacheable("ownerById")
public Owner findOwnerById(final int id) throws DataAccessException {
    return this.ownerRepository.findById(id);
}
```

```java
@Transactional
@CacheEvict(cacheNames="ownerById", allEntries=true)
public void saveOwner(final Owner owner) throws DataAccessException {
    this.ownerRepository.save(owner);
    this.userService.saveUser(owner.getUser());
    this.authoritiesService.saveAuthorities(owner.getUser().getUsername(), "owner");
}
```

Effects:

With the cache, no more queries are made to the database.