

DP2-G3-14

REPORT DE PROFILING Y OPTIMIZACIÓN

<https://github.com/fersolesp/DP2-G3-14>

Reyes Blasco Cuadrado
Pablo Cardenal Gamito
José María Cornac Fisas
Vanessa Pradas Fernández
Fernando Luis Sola Espinosa

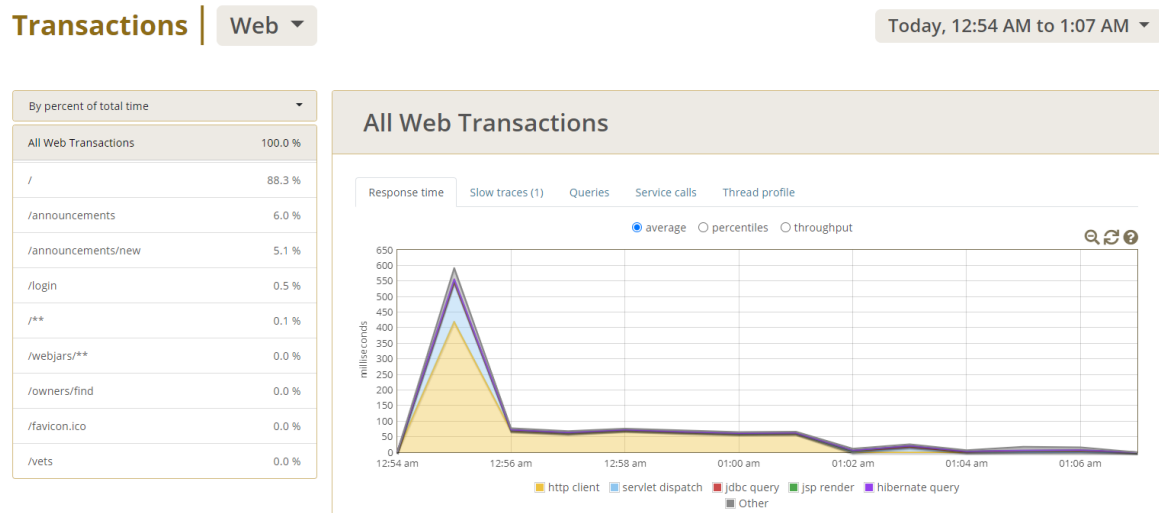
Contenido

1.	HU01.....	3
1.1.	PRIMERAS PRUEBAS.....	3
1.2.	CAMBIOS REALIZADOS (PROYECCIÓN).....	4
1.3.	RESULTADOS	5
2.	HU05.....	7
2.1.	PRIMERAS PRUEBAS.....	7
2.2.	CAMBIOS REALIZADOS (N+1 QUERIES)	9
2.3.	RESULTADOS	10
3.	HU19.....	12
3.1.	PRIMERAS PRUEBAS.....	12
3.2.	CAMBIOS REALIZADOS (N+1 QUERIES)	13
3.3.	RESULTADOS	14
4.	HU21.....	15
4.1.	PRIMERAS PRUEBAS.....	15
4.2.	CAMBIOS REALIZADOS (N+1 QUERIES)	17
4.3.	RESULTADOS	17
5.	API	19
5.1.	PRIMERAS PRUEBAS.....	19
5.2.	CAMBIOS REALIZADOS (CACHÉS)	20
5.3.	RESULTADOS	21

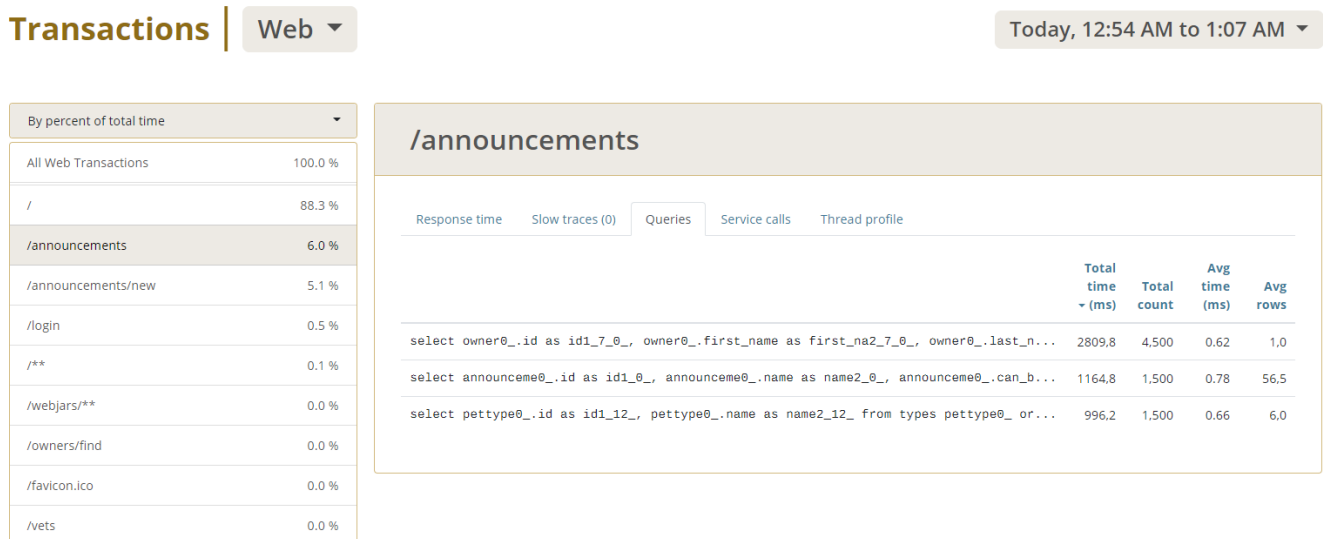
1. HU01

1.1. PRIMERAS PRUEBAS

Al hacer el análisis en “GrowRoot” de los escenarios definidos en formato “scala”, ejecutados con la ayuda de “Gatling” obtenemos los siguientes resultados:



Podemos ver que, tras la página principal, que supone el 88,3% de las peticiones, es el acceso a “/announcements” (listado de “announcements”) lo que se realiza más veces, con los accesos a base de datos que supone, trayéndonos para cada uno de los “announcements” la entidad en sí, como el “owner” que lo publicó y el “pet type” de la mascota:



$$\text{Tiempo total invertido en las "queries"} = 2.809,8 + 1.164,8 + 996,2 = 4.970,8 \text{ ms}$$

Aunque realmente no obtenemos malos resultados, como podemos ver, estamos realizando “queries” de más. Es por ello, que vamos a aplicar una optimización de tipo “proyección”, de forma que seleccionemos concretamente los atributos que se muestran en el listado de “announcements” evitando de esta forma traernos de la base de datos entidades completas como el “owner” cuando realmente no lo mostraremos ni usaremos.

1.2. CAMBIOS REALIZADOS (PROYECCIÓN)

Para realizar la proyección, en primer lugar, debemos crear una interfaz que pueda contener los atributos que realmente vamos a mostrar:

```
1 package org.springframework.samples.petclinic.projections;
2
3
4 public interface PetAnnouncement {
5     String getId();
6     String getName();
7     String getPetName();
8     String getCanBeAdopted();
9     String getType();
10 }
```

Una vez hecho esto, añadimos al repositorio de “Pet” la consulta que nos permita acceder a dichos datos:

```
70 Collection<PetAnnouncement> findAllPetAnnouncements() throws DataAccessException;
71 }
```

```
@Override
@Query("SELECT a.id as id, a.name AS name, a.petName AS petName, a.canBeAdopted AS canBeAdopted, t.name AS type FROM Announcement a INNER JOIN a.type t")
Collection<PetAnnouncement> findAllPetAnnouncements() throws DataAccessException;
```

Obviamente, tendría más lógica añadir este método de consulta a la base de datos en el repositorio de “Announcement”, pero nos hemos visto obligados a añadirlo a este de “Pet”, porque si intentábamos crearlo en un repositorio que nosotros hayamos añadido, es decir, que no viniera en la base del proyecto “PetClinic”, lanzaba un error alegando que no encontraba el atributo “findAllPetAnnouncements” en la clase “Announcement”, por ejemplo. Esto viene dado porque “Spring Data”, por defecto, genera las “queries” a partir del nombre del método, y claro, en este caso no lo lograba porque “PetAnnouncements” no pertenece a “Announcement”. Un ejemplo de esto es que para el método de repositorio:

```
User findByEmailAddress(String emailAddress);
```

Spring generaría la “query”:

```
"SELECT * FROM USERS WHERE EMAIL_ADDRESS = ?1"
```

Existen formas de desactivar este comportamiento de “Spring” pero no hemos logrado conseguirlo. Sin embargo, como hemos dicho, trasladando el método a una clase de repositorio que ya venía en el proyecto base, logramos que “Spring” no genere ese error al no intentar formar la “query” a partir del nombre del método y ejecutar la “query” que nosotros hemos indicado. Finalmente, descubrimos que la razón de este comportamiento en los repositorios, es que nuestros repositorios extienden de “CrudRepository”.

Una vez aclarado esto, debemos añadir un método al servicio de “Announcements” para poder traernos el resultado de nuestra nueva “query”:

```
@Transactional(readOnly=true)
public Iterable<PetAnnouncement> findAllAnnouncements() {
    Iterable<PetAnnouncement> res = this.petRepo.findAllPetAnnouncements();
    if (StreamSupport.stream(res.spliterator(), false).count() == 0) {
        throw new NoSuchElementException();
    }
    return res;
}
```

Y, a continuación, indicamos al controlador que, para mostrar el listado de “Announcements” invoque al nuevo método del servicio:

```
@GetMapping()
public String mostrarAnnouncements(final ModelMap modelMap) {

    String vista = "announcements/announcementsList";
    boolean isempty = false;
    try {
        Iterable<PetAnnouncement> announcements = this.announcementService.findAllAnnouncements();
        modelMap.addAttribute("announcements", announcements);
        modelMap.addAttribute("isanonymoususer", SecurityContextHolder.getContext().getAuthentication().getName().equals("anonymousUser"));
    } catch (NoSuchElementException e) {
        isempty = true;
        modelMap.addAttribute("isempty", isempty);
    }

    return vista;
}
```

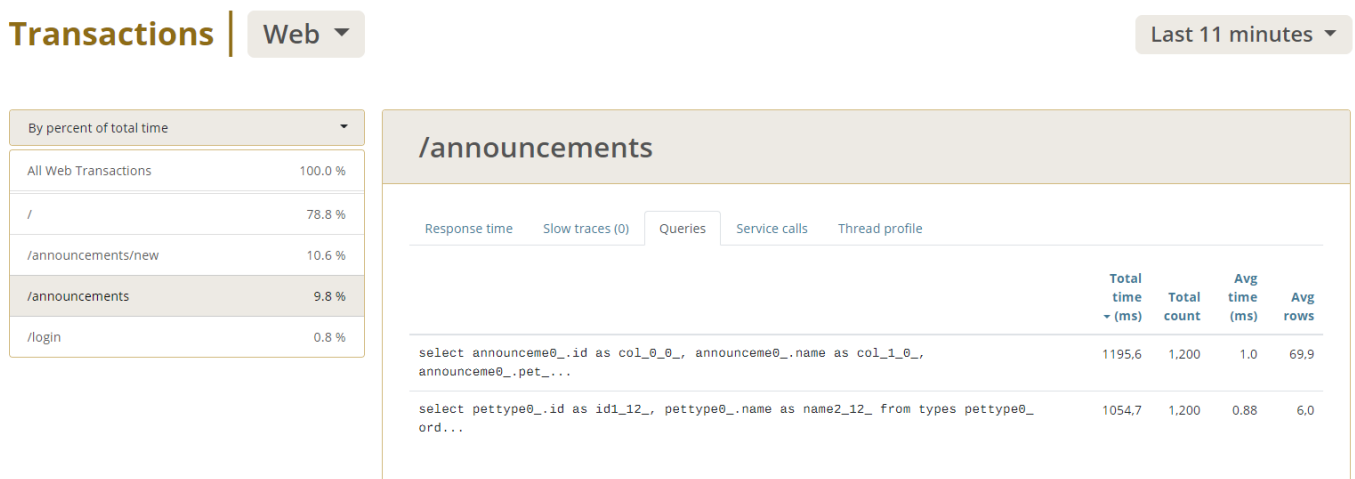
Por último, algo de menor relevancia, es que debemos actualizar la llamada que hace el “Mock” en nuestras pruebas de controlador para que invoque a este nuevo método:

```
@WithMockUser(value = "spring")
@Test
void shouldNotShowAnnouncements() throws Exception {
    Mockito.when(this.announcementService.findAllAnnouncements()).thenReturn(new ArrayList<>());

    this.mockMvc.perform(MockMvcRequestBuilders.get("/announcements"))
        .andExpect(MockMvcResultMatchers.status().isOk())
        .andExpect(MockMvcResultMatchers.model().attributeDoesNotExist("announcements"))
        .andExpect(MockMvcResultMatchers.view().name("announcementsList"))
    }
}
```

1.3. RESULTADOS

Una vez hemos realizado estos cambios, lanzaremos de nuevo la aplicación, junto con los scripts de prueba de “Gatling” que usamos anteriormente, y comprobaremos si ha habido alguna mejora en cuanto al rendimiento con la ayuda de “GlowRoot”. Estos son los resultados que obtenemos ahora:



Tiempo total invertido en las “queries” = 1195,6 + 1054,7 = 2250,3 ms

Como podemos ver, ahora hemos invertido menos de la mitad del tiempo total en la ejecución de las “queries” necesarias para el listado de “Announcements” y esto se debe, en gran parte, porque ahora evitamos traernos toda la información de “Owners” que antes sí que hacíamos, ahorrándonos una “query” bastante costosa temporalmente.

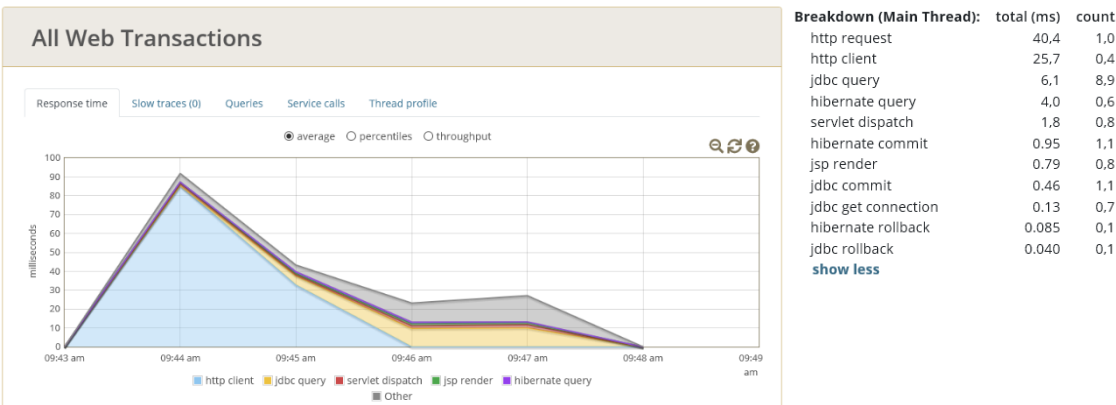
Como conclusión, podemos decir que este método de optimización denominado proyección, es útil y eficaz, que tiene un mayor rendimiento en situaciones donde mostremos por pantalla datos de diversas entidades, pero muy pocos de cada una, y que, por lo tanto, debe seleccionarse bien cuándo usarlo.

2. HU05

2.1. PRIMERAS PRUEBAS

80 usuarios concurrentes durante 50 segundos

Ejecutando el test de rendimiento de la HU05 con 80 usuarios durante 50 segundos obtenemos:

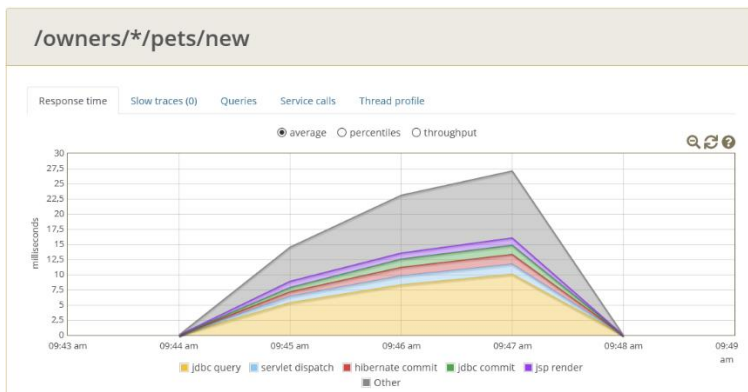


En la tabla se representan los distintos tiempos que tardan en responder las distintas peticiones realizadas en las pruebas de rendimiento. A partir de ellas tenemos que empezar a ver cuáles partes de nuestro código son las que provocan esos retrasos.

All Web Transactions	100.0 %
/	68.0 %
/owners/*/pets/new	17.0 %
/owners	10.2 %
/owners/*	3.3 %
/owners/find	0.8 %
/login	0.7 %

En la tabla que aparece en el lateral izquierdo de la gráfica, observamos el porcentaje de tiempo consumido en las distintas partes del sistema a la hora de realizar las pruebas de rendimiento.

Podemos observar que la ruta raíz es la que más tiempo ha consumido debido a que la primera conexión siempre suele tardar más. Obviando este caso, el siguiente que más tiempo ha consumido ha sido la creación de mascotas con un 17.0 %.



Breakdown:	total (ms)	count
http request	22,9	1,0
jdbc query	8,3	11,0
hibernate commit	2,7	3,0
servlet dispatch	2,5	1,0
hibernate query	1,3	1,5
jdbc commit	1,3	3,0
jsp render	1,1	1,0
jdbc get connection	0,31	1,0
hibernate rollback	0,28	0,3
jdbc rollback	0,13	0,3

Esta gráfica y la tabla a su derecha representan los distintos tiempos de las peticiones solamente en el área de creación de mascotas. Podemos observar que las “queries” se están llevando una parte importante de nuestro tiempo. Si nos vamos a este apartado sabremos que “queries” tardan más y así tener una idea de donde y como poder optimizarlo.

Response time	Slow traces (0)	Queries	Service calls	Thread profile		
			Total time ▼ (ms)	Total count	Avg time (ms)	Avg rows
select visits0_.pet_id as pet_id4_16_0_, visits0_.id as id1_16_0_, visits0_.id as id...			1739,6	2,400	0.72	0
select owner0_.id as id1_7_0_, pets1_.id as id1_9_1_, owner0_.first_name as first_na...			807,8	720	1,1	5,0
select pettype0_.id as id1_12_0_, pettype0_.name as name2_12_0_ from types pettype0_...			612,4	960	0.64	1,0
select pettype0_.id as id1_12_, pettype0_.name as name2_12_ from types pettype0_ ord...			493,9	720	0.69	6,0
select user0_.username as username1_13_0_, user0_.enabled as enabled2_13_0_, user0_...			335,5	480	0.70	1,0

Podemos observar que la primera “query” consume la gran parte del tiempo. El tiempo promedio de respuesta no es alto, pero el problema viene de que esta “query” se realiza muchas veces, y ese es el porqué de la situación.


2.2. CAMBIOS REALIZADOS (N+1 QUERIES)

Estas peticiones se realizan al acceder a las mascotas, ya que se hacen peticiones que en nuestro caso no son necesarias. Estas “queries” son realizadas automáticamente siempre que se accede a una. Por lo que los cambios a realizar serán en:

- **Pet -> PetType**
No nos es necesario buscar el “petType” de la mascota que estamos creando, ya que somos nosotros los que lo tenemos que indicar en el formulario.
- **Pet -> Visit**
No nos es necesario buscar todas las visitas de una mascota que estamos creando puesto que esta inicialmente siempre será de cero.

La solución será añadir el “FetchType.LAZY” en ambos casos para que sólo se realice la consulta si es estrictamente necesario.

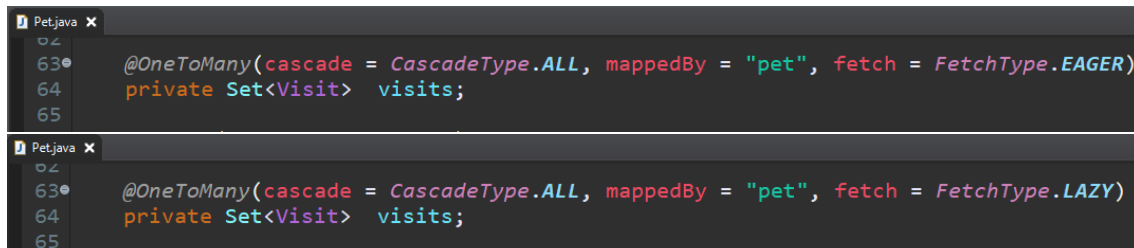
- Pet -> PetType:



```
Pet.java x
55 @ManyToOne
56 @JoinColumn(name = "type_id")
57 private PetType type;
58

Pet.java x
55 @ManyToOne(fetch = FetchType.LAZY)
56 @JoinColumn(name = "type_id")
57 private PetType type;
58
```

- Pet -> Visit:

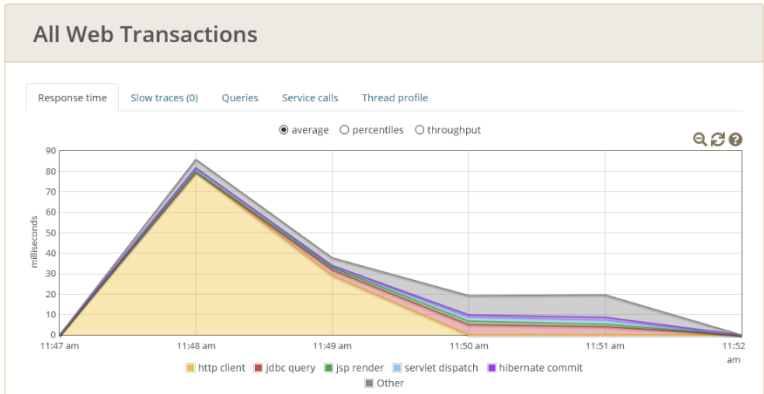


```
Pet.java x
63 @OneToMany(cascade = CascadeType.ALL, mappedBy = "pet", fetch = FetchType.EAGER)
64 private Set<Visit> visits;
65

Pet.java x
63 @OneToMany(cascade = CascadeType.ALL, mappedBy = "pet", fetch = FetchType.LAZY)
64 private Set<Visit> visits;
65
```

Con esto, al acceder a una mascota no nos traerá automáticamente la lista de todas sus visitas ni se pondrá en busca del “PetType” de esa mascota, y nos ahorrará mucho tiempo con dichas peticiones constantes.

2.3. RESULTADOS



Breakdown (Main Thread):

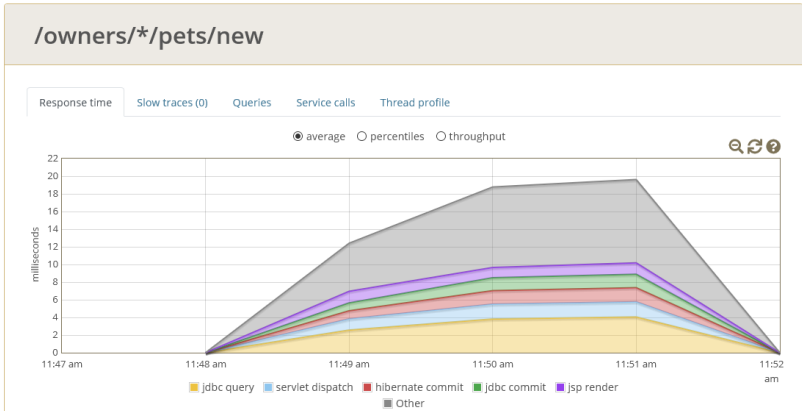
	total (ms)	count
http request	36,5	1,0
http client	24,7	0,4
jdbc query	3,2	3,8
servlet dispatch	3,0	0,8
jsp render	1,9	0,8
hibernate query	1,9	0,6
hibernate commit	1,0	1,1
jdbc commit	0,51	1,1
jdbc get connection	0,18	0,7
hibernate rollback	0,092	0,1

[show more](#) / [show all](#)

Debido a la escala no se puede ver gran diferencia, pero podemos observar que el tiempo consumido en “queries” (color rojo) es más pequeño.

All Web Transactions	100.0 %
/	72.6 %
/owners/*/pets/new	14.8 %
/owners	5.9 %
/owners/*	5.0 %
/login	0.9 %
/owners/find	0.7 %

A simple vista se puede empezar a ver el descenso en el consumo de tiempo de la parte que hemos optimizado. No parece un gran cambio debido a que al reducir esta, los porcentajes tienen que repartirse en otros lugares. Si lo comparamos con la captura tomada anteriormente, los porcentajes de la mayoría de las partes han bajado también, lo cual ha elevado el porcentaje de la creación de mascotas. Pero si entramos en dicha sección nos daremos cuenta de cómo ha influido este cambio/optimización.



Breakdown:

	total (ms)	count
http request	18,0	1,0
jdbc query	3,7	4,0
hibernate commit	2,8	3,0
servlet dispatch	2,8	1,0
jdbc commit	1,4	3,0
hibernate query	1,4	1,5
jsp render	1,2	1,0
jdbc get connection	0,36	1,0
hibernate rollback	0,31	0,3
jdbc rollback	0,14	0,3

Aquí si se puede ver mejor la reducción del tiempo consumido por las “queries” (color amarillo) con respecto a la gráfica antes de la modificación. También en la tabla se ha visto una reducción en los tiempos, en el caso de las “queries” de 4.6 ms, que a pesar de tener pocos usuarios con los que realizar la prueba, cuando existan más peticiones este ahorro de tiempo será mucho más beneficioso.

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_7_0_, pets1_.id as id1_9_1_, owner0_.first_name as first_na2_...	876,8	720	1,2	7,0
		select pettype0_.id as id1_12_, pettype0_.name as name2_12_ from types pettype0_ order...	537,8	720	0.75	6,0
		select user0_.username as username1_13_0_, user0_.enabled as enabled2_13_0_, user0_.pa...	351,6	480	0.73	1,0

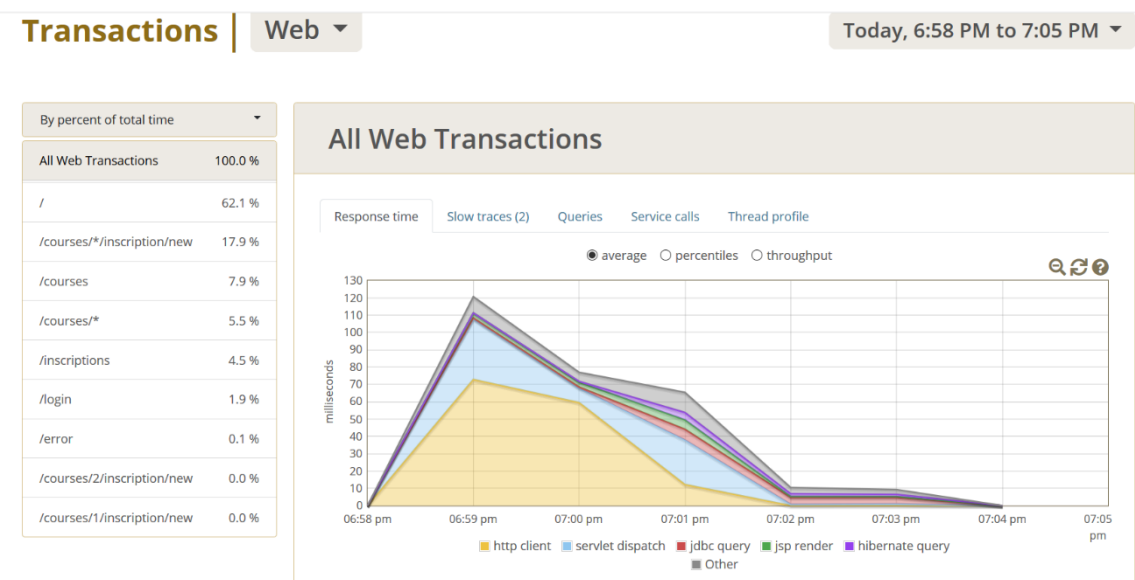
Por último, podemos comprobar que la petición de las visitas al acceder a una mascota ha desaparecido resultando en un ahorro de tiempo altamente positivo.

3. HU19

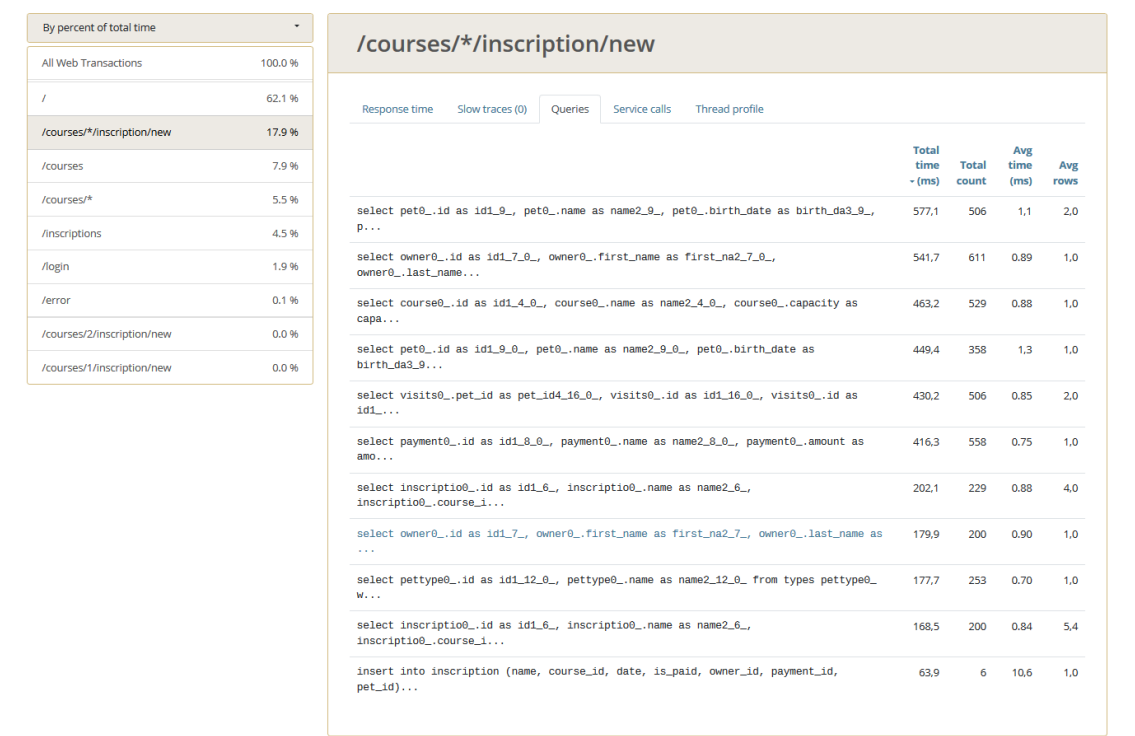
3.1. PRIMERAS PRUEBAS

Ejecución del test de rendimiento de la HU019 con 100 usuarios durante 100 segundos.

Al hacer el análisis en “GrowRoot” de los escenarios definidos en formato “scala”, ejecutados con la ayuda de “Gatling” obtenemos los siguientes resultados:

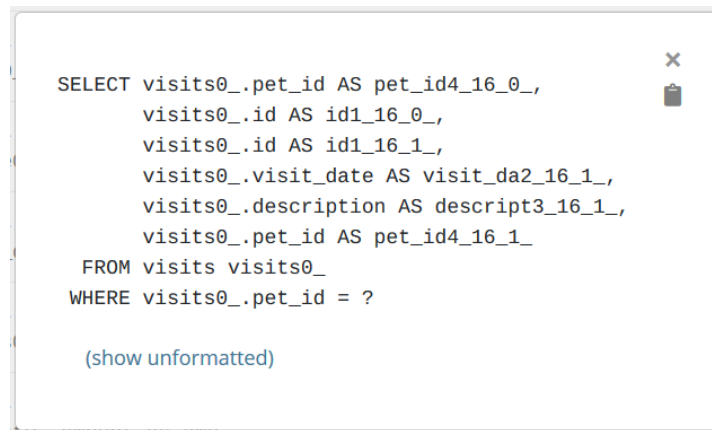


Podemos ver que, tras la página principal, que supone el 62,1% de las peticiones, es el acceso a “/courses/*/inscription/new” lo que se realiza más veces, con lo que a continuación procederemos a observar los tiempos de las consultas a la base de datos que han sido requeridas para acceder a la ya mencionada funcionalidad.



Tiempo total invertido en las “queries” = 577,1 + 541,7 + 463,2 + 449,4 + 430,2 + 416,3 + 202,1 + 179,9 + 177,7 + 168,5 + 63,9 = 3.670,0 ms

Como puede observarse en la tabla anterior de “queries”, se hacen peticiones a la Base de Datos para obtener las visitas. Estas, se realizan automáticamente al acceder a las mascotas, por lo que se trata de un cambio interesante a realizar para optimizar el tiempo.



```
SELECT visits0_.pet_id AS pet_id4_16_0_,
       visits0_.id AS id1_16_0_,
       visits0_.id AS id1_16_1_,
       visits0_.visit_date AS visit_da2_16_1_,
       visits0_.description AS descript3_16_1_,
       visits0_.pet_id AS pet_id4_16_1_
FROM visits visits0_
WHERE visits0_.pet_id = ?

(show unformatted)
```

3.2. CAMBIOS REALIZADOS (N+1 QUERIES)

Dado que se pretende suprimir la llamada al atributo “visits” a la Base de Datos, el cambio a realizar propuesto será la inicialización del campo “fetch” a “FetchType.LAZY” para que sólo se realice la consulta si es estrictamente necesario, que no es nuestro caso.

A continuación, se presenta una captura del cambio realizado propuesto:

```
@Entity
@Table(name = "pets")
public class Pet extends NamedEntity {

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

    @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;
```

Mediante la herramienta “gatling”, nuevamente se ejecutará el archivo “.scala”, esperando una mejora en los resultados obtenidos para las “queries”.

3.3. RESULTADOS

Tras aplicar el cambio anteriormente descrito, pueden observarse los siguientes resultados:

By percent of total time	
All Web Transactions	100.0 %
/	64.4 %
/courses/*/inscription/new	15.5 %
/inscriptions	7.5 %
/courses	6.5 %
/courses/*	4.9 %
/login	0.8 %
/error	0.2 %
/courses/2/inscription/new	0.1 %
/courses/1/inscription/new	0.1 %

/courses/*/inscription/new				
Response time	Slow traces (0)	Queries	Service calls	Thread profile
	Total time ~ (ms)	Total count	Avg time (ms)	Avg rows
select course0_.id as id1_4_0_, course0_.name as name2_4_0_, course0_.capacity as cap...	248,5	500	0.50	1.0
select pet0_.id as id1_9_0_, pet0_.name as name2_9_0_, pet0_.birth_date as birth_da3_9_0_, ...	248,0	400	0.62	2.0
select owner0_.id as id1_7_0_0_, owner0_.first_name as first_na2_7_0_0_, owner0_.last_name...	241,3	500	0.48	1.0
select payment0_.id as id1_8_0_0_, payment0_.name as name2_8_0_0_, payment0_.amount as am...	148,0	400	0.37	1.0
select pet0_.id as id1_9_0_0_, pet0_.name as name2_9_0_0_, pet0_.birth_date as birth_da3_...	145,1	300	0.48	1.0
select owner0_.id as id1_7_0_, owner0_.first_name as first_na2_7_0_, owner0_.last_name as...	120,5	200	0.60	1.0
select inscriptio0_.id as id1_6_0_, inscriptio0_.name as name2_6_0_, inscriptio0_.course...	114,1	200	0.57	11.0
select inscriptio0_.id as id1_6_0_, inscriptio0_.name as name2_6_0_, inscriptio0_.course...	109,4	200	0.55	6.5
select pettype0_.id as id1_12_0_0_, pettype0_.name as name2_12_0_0_ from types pettype0_...	82,9	200	0.41	1.0

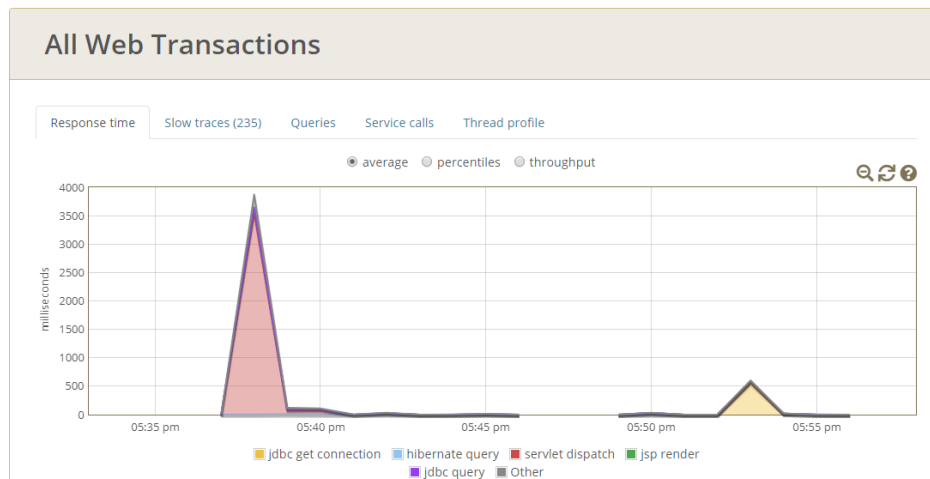
Tiempo total invertido en las “queries” = 248,5 + 248,0 + 241,3 + 148,0 + 120,5 + 114,1 + 109,4 + 82,9 = 1.312,7 ms

En los resultados obtenidos, puede verse que se ha disminuido tanto en el número de “queries” requeridas (de 11 iniciales a 9) como en la tardanza de cada una de ellas, realizando la misma funcionalidad. Por lo tanto, se puede afirmar que el cambio ha sido notoriamente favorable.

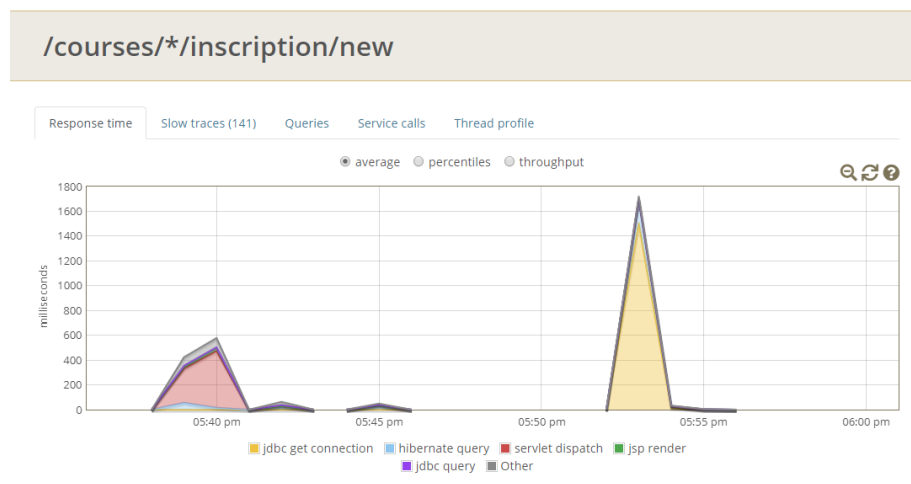
4. HU21

4.1. PRIMERAS PRUEBAS

Como podemos ver en el siguiente gráfico, el aumento del tiempo al principio del mismo se debe al acceso por primera vez en la aplicación, un dato que no se va a poder evitar. En cuanto al segundo incremento del tiempo que se puede observar, se debe a la ejecución de las pruebas definidas durante la ejecución de las pruebas de rendimiento en “scala”.



A continuación, nos centraremos en la consulta que está enfocada a la historia de usuario que queremos optimizar, que consta de un 52.7% del total del tiempo de ejecución. Como podemos ver, el primer incremento es menor que el segundo debido al número de consultas realizadas gracias a “Gatling”.



Vamos a comprobar las queries que se realizan al acceder a la url “/courses/*/inscriptions/new”:

/courses/*/inscription/new				
<div> Response time Slow traces (16) Queries Service calls Thread profile </div>				
	Total time ▼ (ms)	Total count	Avg time (ms)	Avg rows
select pet0_.id as id1_9_, pet0_.name as name2_9_, pet0_.birth_date as birth_da3_9_,...	355,9	10,506	0,034	4,0
select owner0_.id as id1_7_0_, owner0_.first_name as first_na2_7_0_, owner0_.last_na...	336,0	12,259	0,027	1,0
select pet0_.id as id1_9_0_, pet0_.name as name2_9_0_, pet0_.birth_date as birth_da3...	270,6	7,006	0,039	1,2
select visits0_.pet_id as pet_id4_16_0_, visits0_.id as id1_16_0_, visits0_.id as id...	233,7	21,009	0,011	0,3
select pettype0_.id as id1_12_0_, pettype0_.name as name2_12_0_ from types pettype0...	210,9	19,257	0,011	1,0
select course0_.id as id1_4_0_, course0_.name as name2_4_0_, course0_.capacity as ca...	188,9	7,004	0,027	1,0
select owner0_.id as id1_7_, owner0_.first_name as first_na2_7_, owner0_.last_name a...	150,8	3,502	0,043	1,0
select payment0_.id as id1_8_0_, payment0_.name as name2_8_0_, payment0_.amount as a...	103,7	7,007	0,015	1,0
select inscriptio0_.id as id1_6_, inscriptio0_.name as name2_6_, inscriptio0_.course...	102,0	3,503	0,029	2,5
select inscriptio0_.id as id1_6_, inscriptio0_.name as name2_6_, inscriptio0_.course...	72,8	3,502	0,021	1,0
insert into inscription (id, name, course_id, date, is_paid, owner_id, payment_id, p...	3,3	1	3,3	1,0

4.2. CAMBIOS REALIZADOS (N+1 QUERIES)

Como podemos comprobar, se realizan llamadas a la base de datos para obtener valores que no se necesitan en ningún momento para añadir una nueva inscripción. Tras investigar la causa de esto, y encontrar varias dependencias con Fetch= FetchType.EAGER, decidimos cambiar el tipo a LAZY para que sólo se realice la consulta si es estrictamente necesario. Este cambio se ha aplicado a las siguientes clases:

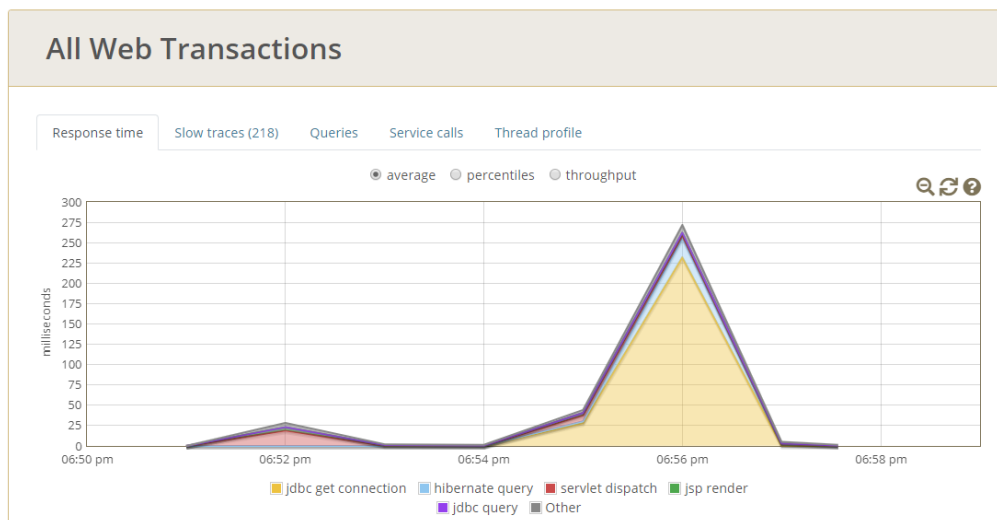
- Pet -> PetType
- Pet -> Visit
- Course -> Trainer
- Inscription -> Payment

Ejemplo:

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

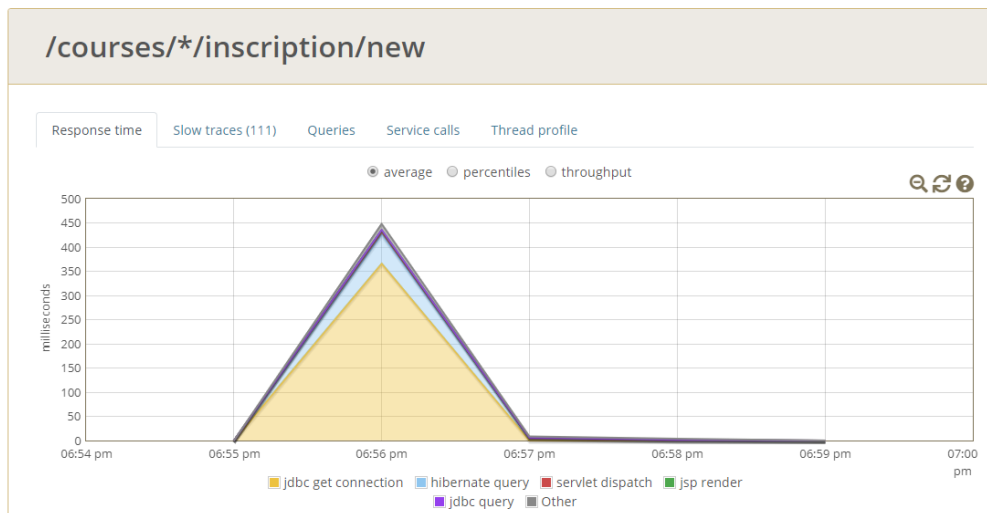
4.3. RESULTADOS

Tras esto, se ha vuelto a analizar el sistema encontrando los siguientes resultados:



Como podemos comprobar, el equivalente a las llamadas realizadas en el primer análisis tiene un tiempo máximo en torno a 500 ms, mientras que este segundo no llega a los 300, ronda concretamente los 275 ms.

Centrándonos en la consulta principal de la historia de usuario, en el primer análisis, el máximo se encuentra en torno a 1700 ms y éste segundo toma 450 ms.



Por último, comprobamos que el número de consultas a la base de datos ha disminuido considerablemente, existiendo ahora siete:

/courses/*/inscription/new

Response time Slow traces (111) **Queries** Service calls Thread profile

	Total time ~ (ms)	Total count	Avg time (ms)	Avg rows
select pet0_.id as id1_9_, pet0_.name as name2_9_, pet0_.birth_date as birth_da3_9_...	517.7	10,500	0.049	4.0
select inscriptio0_.id as id1_6_, inscriptio0_.name as name2_6_, inscriptio0_.course...	490.7	3,500	0.14	2.0
select owner0_.id as id1_7_0_, owner0_.first_name as first_na2_7_0_, owner0_.last_na...	411.3	10,500	0.039	1.0
select pet0_.id as id1_9_0_, pet0_.name as name2_9_0_, pet0_.birth_date as birth_da3...	391.8	5,250	0.075	1.0
select course0_.id as id1_4_0_, course0_.name as name2_4_0_, course0_.capacity as ca...	159.9	5,250	0.030	1.0
select owner0_.id as id1_7_, owner0_.first_name as first_na2_7_, owner0_.last_name a...	145.5	3,500	0.042	1.0
select inscriptio0_.id as id1_6_, inscriptio0_.name as name2_6_, inscriptio0_.course...	100.2	3,500	0.029	0.5

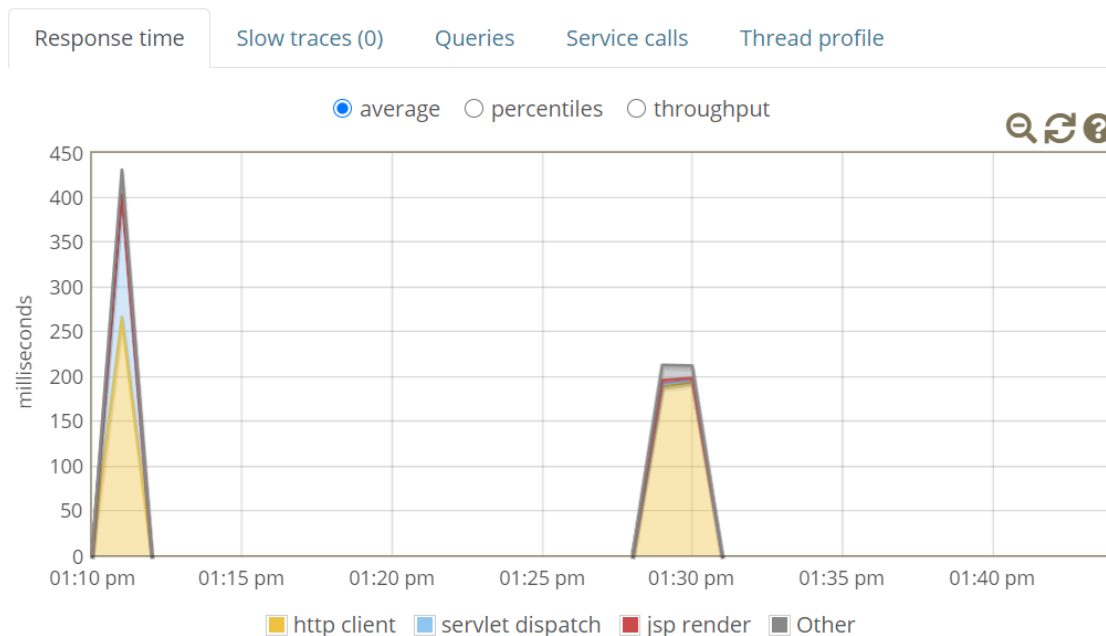
Concluyendo, el tiempo total que tomaba la consulta a la dirección “/courses/*/inscription/new” era de un 52.7% del total, mientras que, tras los cambios realizados, este porcentaje se ha visto reducido en un 6.3%, dejándonos en un 46.4%.

5. API

5.1. PRIMERAS PRUEBAS

Realizamos el “profiling” y optimización sobre la API (“Random Cat Fact”) implementada en nuestra aplicación. En mi caso particular la petición a la API se realiza en la página principal “Home/Welcome”, es por ello que no debemos realizar ninguna acción más que recargar la página o ejecutar el fichero “.scala” con “gatling.bat”, que hará lo mismo pero con varios usuarios concurrentes.

Como podemos ver en el siguiente gráfico, el aumento del tiempo al principio del mismo se debe al acceso por primera vez en la aplicación, un dato que no se va a poder evitar. A continuación, vemos la petición de la recarga de “Home” y cómo su tiempo de respuesta ha disminuido bastante con respecto a la primera llamada.



Este sería el rendimiento que tiene sin realizar ninguna modificación en el código de la aplicación. A continuación, observamos los tiempos específicos de la llamada a la API la cual está desplegada en “Heroku”, podemos comprobar que el tiempo medio de cada llamada es de unos 86.0 milisegundos.

Response time	Slow traces (0)	Queries	Service calls	Thread profile
		Total time ▼ (ms)	Total count	Avg time (ms)
GET https://arcane-ocean-65006.herokuapp.com/		137.590,5	1,600	86,0

5.2. CAMBIOS REALIZADOS (CACHÉS)

Ahora pasaré a comentar los cambios que he realizado para disminuir en gran medida dichos tiempos. En mi caso específico hemos hecho uso de una caché para almacenar los datos que devuelve la petición a la API, más específicamente, esta cache guarda los datos durante 1 día (24 horas) antes de vaciarse por completo. Aunque se han seguido paso a paso los tutoriales subidos a la enseñanza virtual, a continuación, dejaremos capturas de algunos cambios realizados.

Clase “CatFactService.java”:

```
@Service
public class CatFactService {

    @Cacheable("api")
    public String findcatFactService() {
        RestTemplate template = new RestTemplate();
        CatFact catFact = template.getForObject("https://arcane-ocean-65006.herokuapp.com/", CatFact.class);
        return catFact.getData()[0];
    }
}
```

Fichero “ehcache3.xml”:

```
<cache alias="api" uses-template="default">
    <key-type>org.springframework.cache.interceptor.SimpleKey</key-type>
    <value-type>java.lang.String</value-type>
</cache>
</config>
```

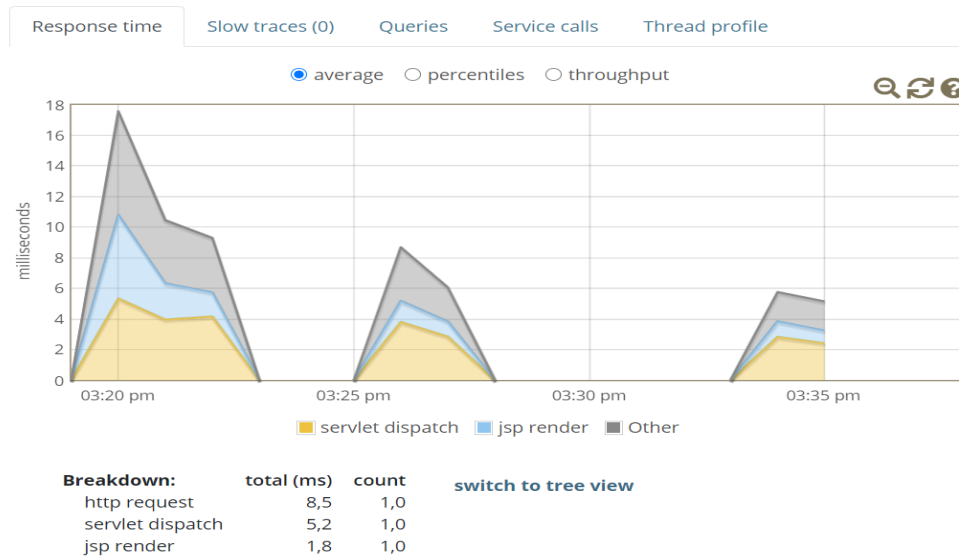
```
<expiry>
    <ttl unit="seconds">86400</ttl>
</expiry>
```

Fichero “application.properties”:

```
spring.cache.jcache.config=classpath:ehcache3.xml
```

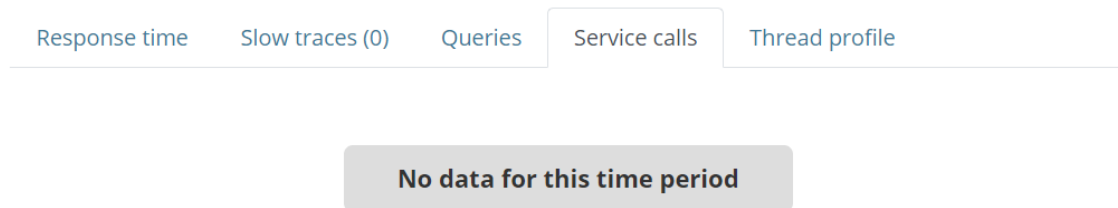
5.3. RESULTADOS

Una vez realizado los cambios mencionados anteriormente, volvemos a lanzar el análisis con GlowRoot para ver de qué manera ha surtido efecto y obtenemos lo siguiente:



Primero se hace la primera petición a la API, la cual cargará los datos solicitados en la caché, dicha petición no se muestra en la captura de arriba. Lo que observamos en la captura es como una vez ya cargados los datos en caché las peticiones a la API son mucho más rápidas, tanto es así que el tiempo de respuesta medio ha decrecido de 86 milisegundos por cada petición, a tan solo 8.5 milisegundos por petición.

También cabe destacar que como ahora se consulta la caché, la aplicación no tiene que hacer una consulta directa a la API, sino que accede y muestra los resultados guardados en caché, esto se puede ver en la siguiente captura.



Es por eso por lo que observamos que no se realizan llamadas al servicio de la API una vez ya tiene los datos en caché.