

Universidad de Sevilla
Escuela Técnica Superior de Ingeniería Informática

Profiling and optimization



Grado en Ingeniería Informática – Ingeniería del Software
Diseño y Pruebas II
Curso 2019 – 2020

Miembros del equipo

Jorge Andrea Molina

Juan Carlos Cortés Muñoz

María Elena Molino Peña

Alejandro Muñoz Aranda

Mario Ruano Fernández

Fernando Ruiz Robles

<https://github.com/dp2-g3-7/petclinic.git>

Índice

Profiling	2
Resumen.....	2
Banners	3
Treatments.....	6
Visits	12
Pets.....	21

Profiling

Se ha obtenido una mejora del rendimiento en varias partes de la aplicación gracias a las optimizaciones realizadas tras el proceso de *profiling* a cinco historias de usuario. Se mostrarán los datos de las peticiones antes y después de las mejoras, mediante capturas de *Glowroot*, los elementos que producen problemas y la solución que se le ha dado.

Resumen

Tras la realización de los análisis de Profiling que se detallan posteriormente, hemos ejecutado los script de Gatling de las historias seleccionadas, con el fin de corroborar que las optimizaciones desarrolladas han surgido efecto.

Visualizando las tablas podemos extraer varios aspectos importantes:

- 1- A la historia banner no se le ha realizado un análisis a posterior de Gatling porque el objetivo de mejorar el rendimiento de banner era disminuir el tiempo de ejecución en las peticiones del resto de historias.
- 2- En todos los casos, los usuarios concurrentes óptimos obtenidos a posterior superan el mínimo de ruptura alcanzado antes de realizar las optimizaciones.

Historia de usuario	Óptimo	Mínimo Ruptura
CrearTratamiento	100	1500
RegistrarVisita	150	1000
SolicitarRegistroMascota	50	300
AceptarRechazarSolicitudMascota	10	500

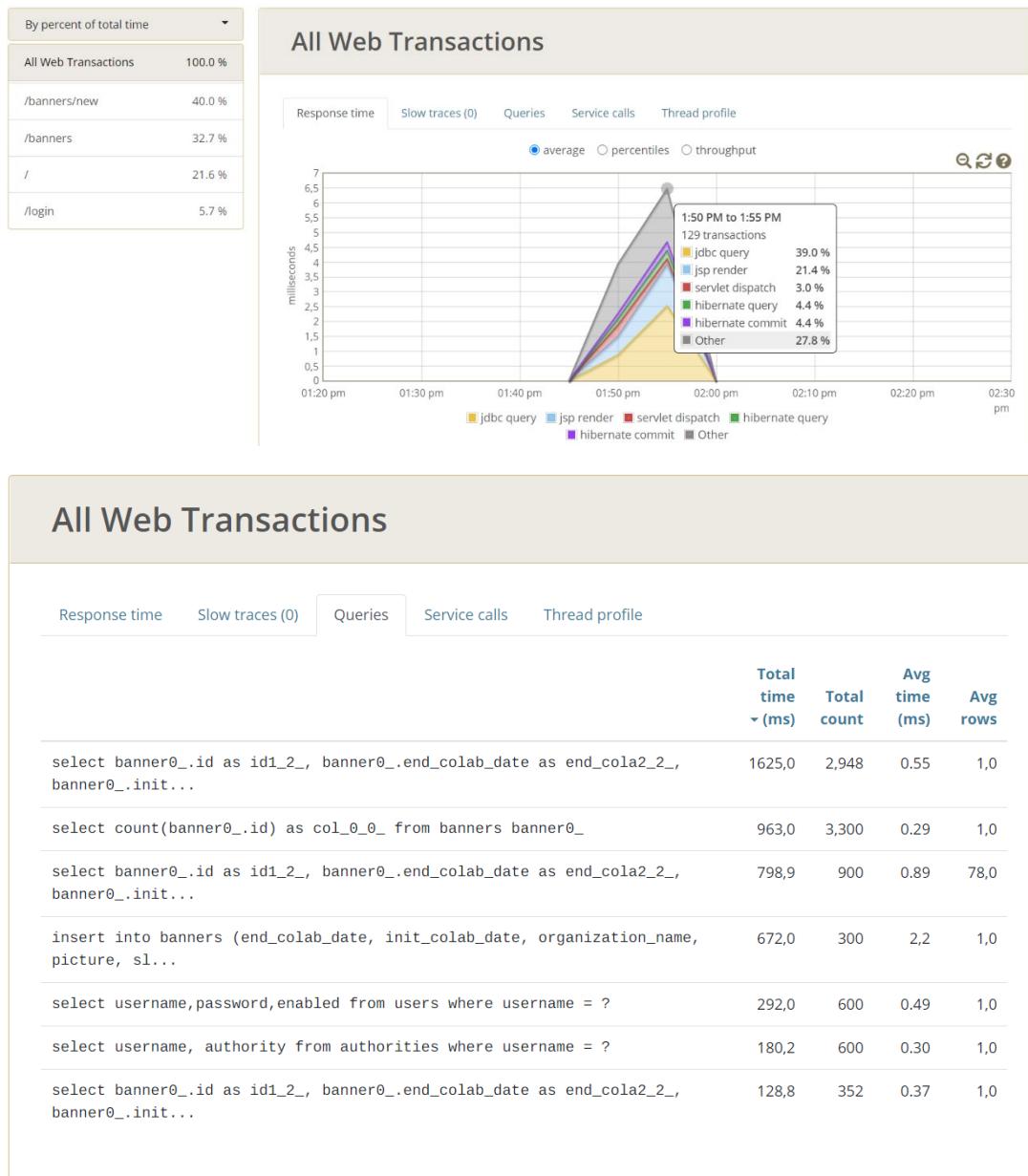
Historia de usuario	Óptimo post-profiling	Mínimo Ruptura post-profiling
CrearTratamiento	1600	2500
RegistrarVisita	1200	5000
SolicitarRegistroMascota	5000	10000
AceptarRechazarSolicitudMascota	3500	10000

Por tanto, esto demuestra que realizar optimizaciones al código tras su desarrollo es muy importante. Permitiéndonos que el producto mejore el rendimiento notablemente.

Banners

Se escogió este subsistema porque está presente en todas las consultas de la aplicación; en todas se escoge un banner de forma aleatoria que se muestra en todas las vistas. El principal problema consiste en que se realizan gran cantidad de estas consultas a la base de datos. La solución es implementar una caché para reducir el número de transacciones con la base de datos.

Resultados Pre-Optimización



En la clase BannerService se crea la caché para los banners publicitarios de todas las vistas y se elimina en caso de que se añada un nuevo banner o se elimine uno existente.

```

@Cacheable("cacheFindRandomBanner")
@Transactional(readOnly = true)
public Banner findRandomBanner() {
    return this.bannerRepository.findRandomBanner();
}

@CacheEvict(cacheNames="cacheFindRandomBanner", allEntries = true)
@Transactional
public void saveBanner(@Valid Banner banner) {
    bannerRepository.save(banner);
}

@CacheEvict(cacheNames="cacheFindRandomBanner", allEntries = true)
@Transactional
public void deleteBannerById(int bannerId) {
    this.bannerRepository.deleteById(bannerId);
}

```

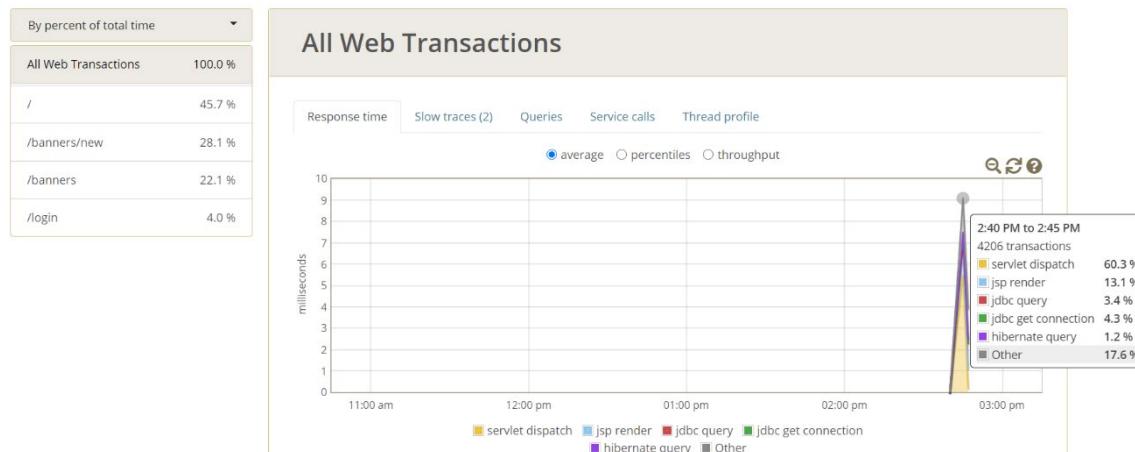
Añadimos la caché al fichero ehcache3.xml:

```

<cache alias="cacheFindRandomBanner" uses-template="default">
    <key-type>org.springframework.cache.interceptor.SimpleKey</key-type>
    <value-type>org.springframework.samples.petclinic.model.Banner</value-type>
</cache>

```

Resultados Post-Optimización



All Web Transactions

Response time

Slow traces (2)

Queries

Service calls

Thread profile

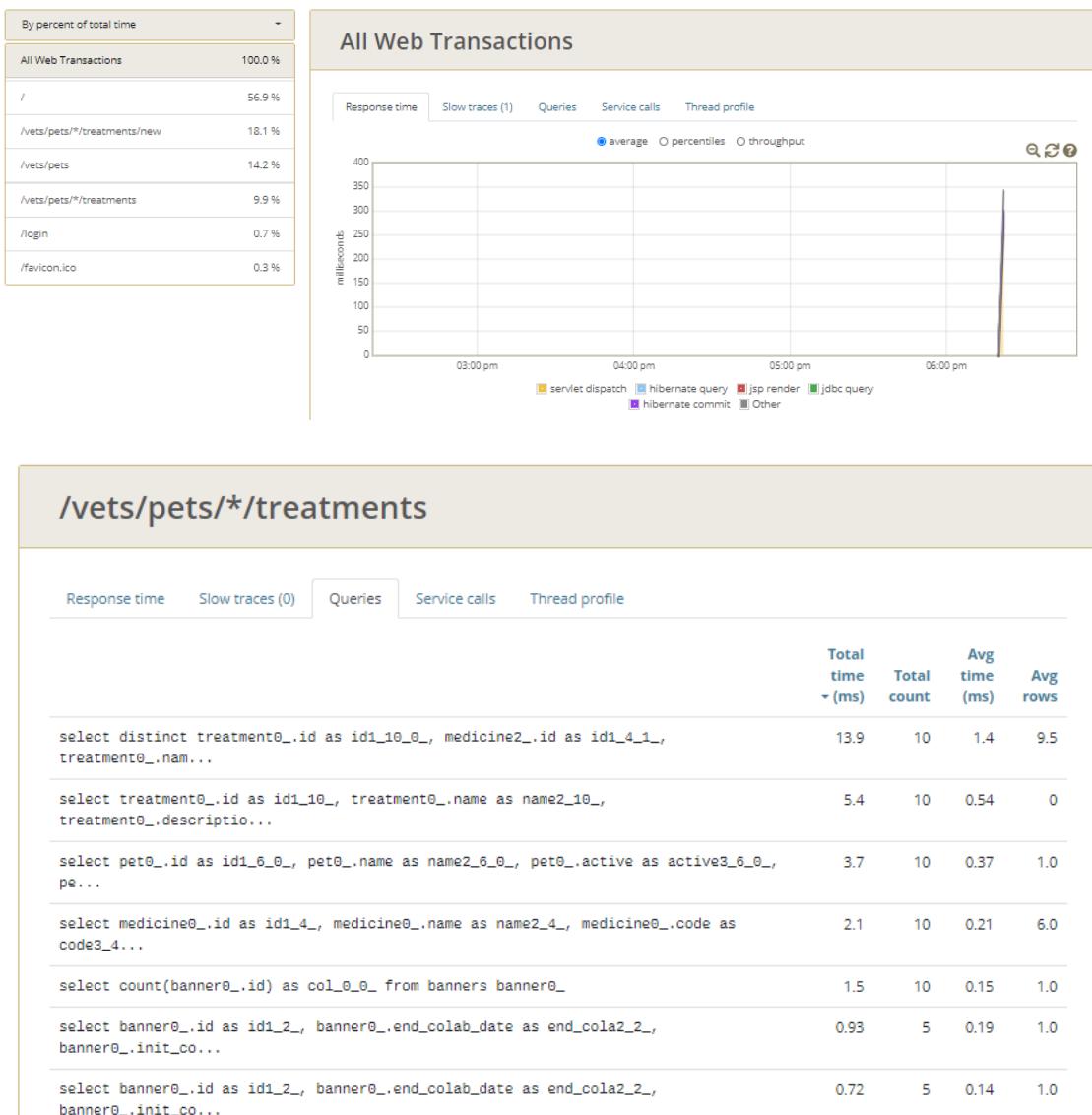
	Total time ▾ (ms)	Total count	Avg time (ms)	Avg rows
select banner0_.id as id1_2_, banner0_.end_colab_date as end_colab2_2_, banner0_.init_c...	832,1	900	0.92	78,0
insert into banners (end_colab_date, init_colab_date, organization_name, picture, slog...	718,2	300	2,4	1,0
select username,password,enabled from users where username = ?	316,7	600	0.53	1,0
select banner0_.id as id1_2_, banner0_.end_colab_date as end_colab2_2_, banner0_.init_c...	192,6	301	0.64	1,0
select username, authority from authorities where username = ?	177,7	600	0.30	1,0
select count(banner0_.id) as col_0_0_ from banners banner0_	84,0	307	0.27	1,0
select banner0_.id as id1_2_, banner0_.end_colab_date as end_colab2_2_, banner0_.init_c...	1,5	6	0.24	1,0

Se puede observar como la consulta “select count(...)” ha disminuido de 963 ms a 84 ms, gracias a la cache añadida en el servicio. Esto ocurrirá en toda la aplicación, lo que permite ganar un segundo en todas las peticiones de la aplicación.

Treatments

El principal problema para este subsistema es tener que listar las mascotas del sistema sumado a que, por la implementación, se cogen todos los atributos de cada una de ellas. Para solucionar este apartado, nos hemos basado en una caché de las mascotas que hace que dicha petición se suavice.

Resultados Pre-Optimización



/vets/pets				
	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_, owner0_.first_name as first_na2_5_0_, owner0_.last_name ...				5.9
select pettype0_.id as id1_12_0_, pettype0_.name as name2_12_0_ from types pettype0_ wh...				2.8
select specialty0_.id as id1_7_, specialty0_.name as name2_7_ from specialties specialt...				2.3
select banner0_.id as id1_2_, banner0_.end_colab_date as end_colab2_2_, banner0_.init_co...				1.1
select count(banner0_.id) as col_0_0_ from banners banner0_				1.1
select pet0_.id as id1_6_, pet0_.name as name2_6_, pet0_.active as active3_6_, pet0_.bi...				0.86

Se añaden las anotaciones correspondientes a los métodos de PetService:

```

@Cacheable("allPets")
public List<Pet> findAll() {
    return this.petRepository.findAll();
}

@Transactional(rollbackFor = DuplicatedPetNameException.class)
@CacheEvict(cacheNames = {"allPets"}, allEntries = true)
public void savePet(Pet pet, Owner owner) throws DataAccessException, DuplicatedPetNameException {
    if (existOtherPetWithSameName(pet, owner.getId())) {
        throw new DuplicatedPetNameException();
    } else
        owner.addPet(pet);
    petRepository.save(pet);
}

@Transactional(rollbackFor = DuplicatedPetNameException.class)
@CacheEvict(cacheNames = {"allPets"}, allEntries = true)
public void EditPet(Pet petToUpdate) throws DataAccessException, DuplicatedPetNameException{
    if (existOtherPetWithSameName(petToUpdate, petToUpdate.getOwner().getId())) {
        throw new DuplicatedPetNameException();
    } else
        petRepository.save(petToUpdate);
}

```

Y se incluye la caché en el fichero correspondiente:

```

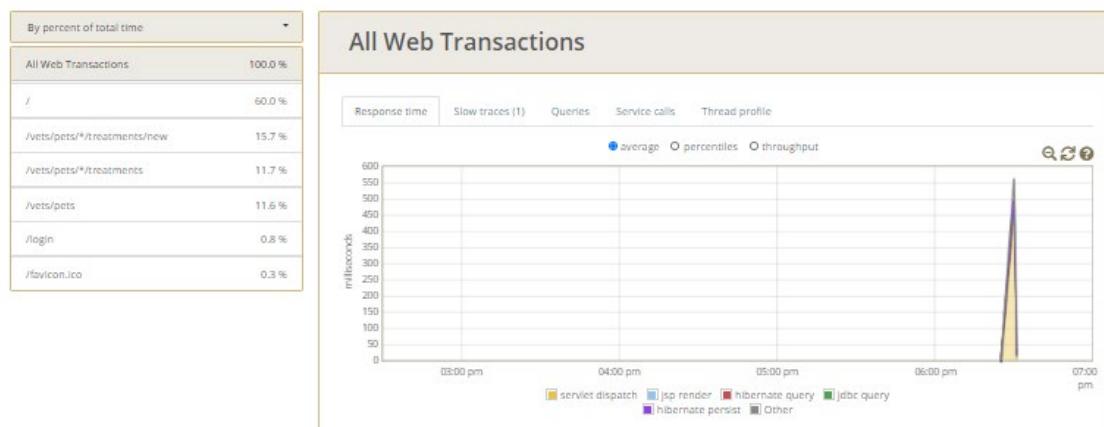
<cache alias="allPets" uses-template="default">
    <key-type>org.springframework.cache.interceptor.SimpleKey</key-type>
    <value-type>java.util.List</value-type>
</cache>

```

Para el listado de tratamientos de una mascota se ha optado por aplicar una N+1 Query, ya que los tratamientos precisan de las medicinas que tienen asociadas.

```
@Query("SELECT DISTINCT t FROM Treatment t LEFT JOIN FETCH t.medicines m WHERE t.pet.id=:petId"
       + " AND t.timeLimit >= CURRENT_DATE ORDER BY t.timeLimit ASC")
public List<Treatment> findCurrenTreatmentWithMedicineByPet(@Param("petId") Integer petId);
```

Resultados Post-Optimización



/vets/pets				
		Total time ▼ (ms)	Total count	Avg time (ms)
		Avg rows		
select owner0_.id as id1_5_0_, owner0_.first_name as first_na2_5_0_, owner0_.last_name ...		3.6	10	0.36
select specialty0_.id as id1_7_, specialty0_.name as name2_7_ from specialties specialt...		1.2	5	0.24
select banner0_.id as id1_2_, banner0_.end_colab_date as end_colab2_2_, banner0_.init_co...		1.1	3	0.36
select count(banner0_.id) as col_0_0_ from banners banner0_		1.1	5	0.22
select pettype0_.id as id1_12_0_, pettype0_.name as name2_12_0_ from types pettype0_		0.93	7	0.13
select pet0_.id as id1_6_, pet0_.name as name2_6_, pet0_.active as active3_6_, pet0_.bi...		0.47	1	0.47
select banner0_.id as id1_2_, banner0_.end_colab_date as end_colab2_2_, banner0_.init_co...		0.28	2	0.14

Treatments fue seleccionada para realizar Profiling porque pertenecía a las historias que tenían menores óptimos, en concreto permitía 100 usuarios. Tras la realización de profiling y optimización podemos contemplar que los cambios han sido mínimos. Esto se debe a que el porcentaje de tiempo que consumían las consultas a la base de datos era del 3%, es decir, un porcentaje muy bajo. Pude ser que la historia no disponga de un buen rendimiento por otros motivos.

Resultados Gatling

Tras completar todas las optimizaciones a cada una de las historias de usuario, se ha realizado un análisis de Gatling para comprobar la mejora.

- Antes de realizar Optimización con 100 usuarios:



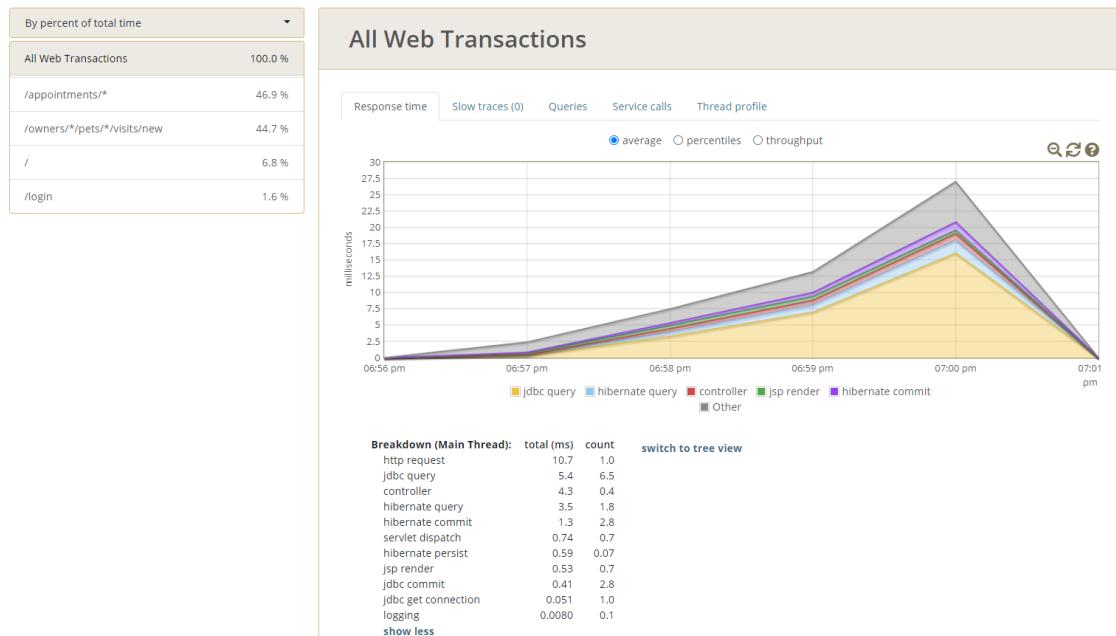
- Tras realizar Optimización con 1600 usuarios:



Visits

El subsistema de visitas de la aplicación tiene como principal problema la creación de las mismas, incluyendo las vistas del listado de citas y la del formulario de creación de una visita. La solución que se le ha dado a este problema se basa en la implementación de cachés para reducir el número de transacciones con la base de datos.

Resultados Pre-Optimización



All Web Transactions

Response time

Slow traces (0)

Queries

Service calls

Thread profile

		Total time ▾ (ms)	Total count	Avg time (ms)	Avg rows
select pet0_.id as id1_6_0_, pet0_.name as name2_6_0_, pet0_.active as active3_6_0_...	6368.5	2,400	2.7	102.0	
insert into visits (visit_date, description, pet_id) values (?, ?, ?)	1255.7	150	8.4	1.0	
insert into visit_medical_tests (visit_id, medical_test_id) values (?, ?)	814.9	450	1.8	1.0	
select count(visit0_.id) as col_0_0_ from visits visit0_ where visit0_.pet_id=? and...	707.0	1,350	0.52	1.0	
select owner0_.id as id1_5_0_, owner0_.first_name as first_na2_5_0_, owner0_.last_n...	397.9	1,350	0.29	1.0	
select count(banner0_.id) as col_0_0_ from banners banner0_	348.6	1,650	0.21	1.0	
select banner0_.id as id1_2_, banner0_.end_colab_date as end_colab2_2_, banner0_.ini...	328.2	1,233	0.27	1.0	
select medicaltes0_.id as id1_3_, medicaltes0_.name as name2_3_, medicaltes0_.descr...	313.8	1,050	0.30	6.0	
select specialtie0_.vet_id as vet_id1_14_0_, specialtie0_.specialty_id as specialt2...	311.4	1,050	0.30	0.4	
select user0_.username as username1_13_0_, user0_.enabled as enabled2_13_0_, user0_...	294.3	1,050	0.28	1.0	
select vet0_.id as id1_15_0_, vet0_.first_name as first_na2_15_0_, vet0_.last_name ...	194.0	450	0.43	1.0	
select appointmen0_.id as id1_0_, appointmen0_.appointment_date as appointm2_0_, ap...	182.0	450	0.40	3.0	
select appointmen0_.id as id1_0_, appointmen0_.appointment_date as appointm2_0_, ap...	165.4	450	0.37	2.0	
select specialty0_.id as id1_7_, specialty0_.name as name2_7_ from specialties spec...	129.6	450	0.29	7.0	
select username,password(enabled from users where username = ?	118.8	300	0.40	1.0	
select banner0_.id as id1_2_, banner0_.end_colab_date as end_colab2_2_, banner0_.ini...	107.7	417	0.26	1.0	
select username, authority from authorities where username = ?	75.9	300	0.25	1.0	

En primer lugar, para cada caché que queramos realizar, añadimos las anotaciones necesarias:

```

@Transactional(readOnly = true)
@Cacheable("medicalTests")
public Collection<MedicalTest> findMedicalTests() {
    return this.medicalTestRepository.findAll();
}

@Transactional
@CacheEvict(cacheNames="medicalTests", allEntries=true)
public void saveMedicalTest(MedicalTest medicalTest) {
    this.medicalTestRepository.save(medicalTest);
}

```

```

@Transactional
@CacheEvict(cacheNames = "vetById", allEntries = true)
public void saveVet(Vet vet) throws DataAccessException {
    //creating vet
    vetRepository.save(vet);
    //creating user
    userService.saveUser(vet.getUser());
    //creating authorities
    authoritiesService.saveAuthorities(vet.getUser().getUsername(), "veterinarian");
}

@Transactional(readOnly = true)
@Cacheable("vetById")
public Vet findVetById(Integer vetId) {
    return vetRepository.findById(vetId);
}

@Transactional(readOnly = true)
@Cacheable("countVisitsByPetAndDate")
public Integer countVisitsByDate(Integer petId, LocalDate date) {
    return visitRepository.countByDate(petId, date);
}

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

@Transactional(readOnly = true)
@Cacheable("petById")
public Pet findPetById(Integer id) throws DataAccessException {
    return petRepository.findById(id);
}

@Transactional(rollbackFor = DuplicatedPetNameException.class)
@CacheEvict(cacheNames = {"allPets", "petById"}, allEntries = true)
public void savePet(Pet pet, Owner owner) throws DataAccessException, DuplicatedPetNameException {
    if (existOtherPetWithSameName(pet, owner.getId())) {
        throw new DuplicatedPetNameException();
    } else {
        owner.addPet(pet);
        petRepository.save(pet);
    }
}

@Transactional(rollbackFor = DuplicatedPetNameException.class)
@CacheEvict(cacheNames = {"allPets", "petById"}, allEntries = true)
public void EditPet(Pet petToUpdate) throws DataAccessException, DuplicatedPetNameException{
    if (existOtherPetWithSameName(petToUpdate, petToUpdate.getOwner().getId())) {
        throw new DuplicatedPetNameException();
    } else
        petRepository.save(petToUpdate);
}

```

```

@Transactional(readOnly = true)
@Cacheable("specialties")
public Collection<Specialty> findSpecialties() {
    return vetRepository.findSpecialties();
}

@Transactional
@CacheEvict(cacheNames = "vetById", allEntries = true)
public void saveVet(Vet vet) throws DataAccessException {
    //creating vet
    vetRepository.save(vet);
    //creating user
    userService.saveUser(vet.getUser());
    //creating authorities
    authoritiesService.saveAuthorities(vet.getUser().getUsername(), "veterinarian");
}

@Transactional(readOnly=true)
@Cacheable("appointmentsByVetAndDate")
public List<Appointment> getAppointmentsByVetAndDate(Integer vetId, LocalDate date) {
    return this.appointmentRepository.getAppointmentsByVetAndDate(vetId, date);
}

@Transactional(readOnly=true)
@Cacheable("nextAppointmentsByVetAndDate")
public List<Appointment> getNextAppointmentsByVetId(Integer vetId, LocalDate date) {
    return this.appointmentRepository.getNextAppointmentsByVetId(vetId, date);
}

@Transactional
@CacheEvict(cacheNames = {"appointmentsByVetAndDate", "nextAppointmentsByVetAndDate"}, allEntries = true)
public void saveAppointment(final Appointment appointment, Integer vetId) throws VeterinarianNotAvailableException {
    if (!isPossibleAppointment(appointment, vetId)) {
        throw new VeterinarianNotAvailableException();
    } else {
        Vet vet = this.vetRepository.findById(vetId);
        LocalDate requestDate = LocalDate.now();
        appointment.setVet(vet);
        appointment.setAppointmentRequestDate(requestDate);
        this.appointmentRepository.save(appointment);
    }
}

@Transactional
@CacheEvict(cacheNames = {"appointmentsByVetAndDate", "nextAppointmentsByVetAndDate"}, allEntries = true)
public void deleteAppointment(final Appointment appointment) {
    this.appointmentRepository.delete(appointment);
}

@Transactional
@CacheEvict(cacheNames = {"appointmentsByVetAndDate", "nextAppointmentsByVetAndDate"}, allEntries = true)
public void editAppointment(final Appointment appointment) throws VeterinarianNotAvailableException {
    int vetId = appointment.getVet().getId();
    Appointment appointmentToUpdate = this.appointmentRepository.findById(appointment.getId());
    LocalDate newDate = appointment.getAppointmentDate();
    if (!isPossibleAppointment(appointment, vetId) && !appointmentToUpdate.getAppointmentDate()
        .equals(appointment.getAppointmentDate())) {
        throw new VeterinarianNotAvailableException();
    } else {
        appointmentToUpdate.setAppointmentDate(appointment.getAppointmentDate());
        LocalDate date = LocalDate.now();
        appointment.setAppointmentDate(newDate);
        appointment.setAppointmentRequestDate(date);
        this.appointmentRepository.save(appointment);
    }
}

```

Una vez hecho esto, creamos cada una de las cachés en el fichero ehcache3.xml:

```
<cache alias="medicalTests" uses-template="default">
    <key-type>org.springframework.cache.interceptor.SimpleKey</key-type>
    <value-type>java.util.Collection</value-type>
</cache>

<cache alias="vetById" uses-template="default">
    <key-type>java.lang.Integer</key-type>
    <value-type>org.springframework.samples.petclinic.model.Vet</value-type>
</cache>

<cache alias="petById" uses-template="default">
    <key-type>java.lang.Integer</key-type>
    <value-type>org.springframework.samples.petclinic.model.Pet</value-type>
</cache>

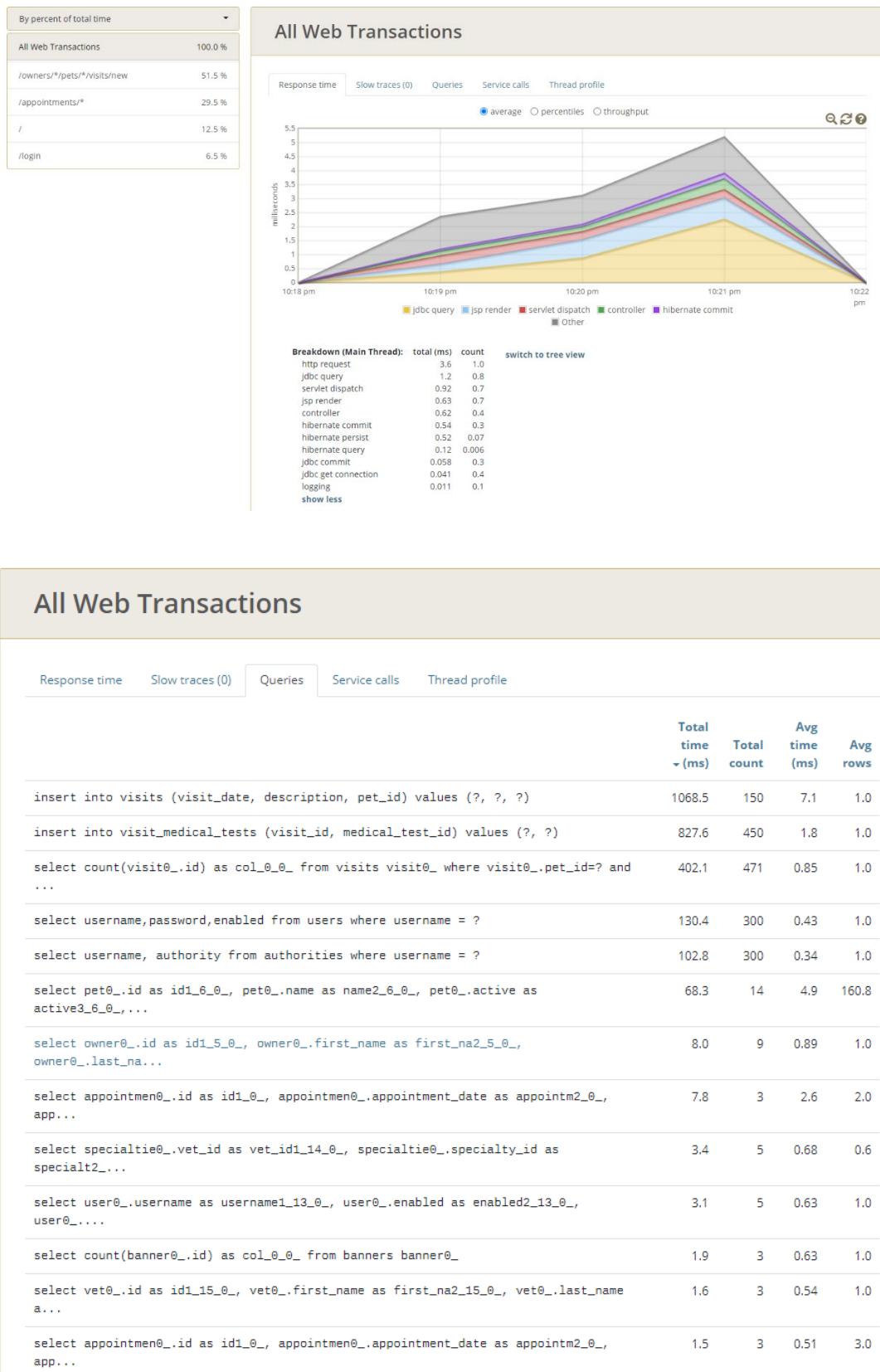
<cache alias="appointmentsByVetAndDate" uses-template="default">
    <key-type>org.springframework.cache.interceptor.SimpleKey</key-type>
    <value-type>java.util.ArrayList</value-type>
</cache>

<cache alias="nextAppointmentsByVetAndDate" uses-template="default">
    <key-type>org.springframework.cache.interceptor.SimpleKey</key-type>
    <value-type>java.util.ArrayList</value-type>
</cache>

<cache alias="countVisitsByPetAndDate" uses-template="default">
    <key-type>org.springframework.cache.interceptor.SimpleKey</key-type>
    <value-type>java.lang.Integer</value-type>
</cache>

<cache alias="specialties" uses-template="default">
    <key-type>org.springframework.cache.interceptor.SimpleKey</key-type>
    <value-type>java.util.Collection</value-type>
</cache>
```

Resultados Post-Optimización



select specialty0_.id as id1_7_, specialty0_.name as name2_7_ from specialties speci...	1.1	3	0.38	7.0
select medicaltes0_.id as id1_3_, medicaltes0_.name as name2_3_, medicaltes0_.descri...	0.87	2	0.44	6.0
select banner0_.id as id1_2_, banner0_.end_colab_date as end_colab2_2_, banner0_.init...	0.82	2	0.41	1.0
select banner0_.id as id1_2_, banner0_.end_colab_date as end_colab2_2_, banner0_.init...	0.36	1	0.36	1.0

Como podemos observar, al comparar los resultados obtenidos antes y después de la optimización, las diferencias son muy notables. Gracias a las cachés hemos podido reducir en gran cantidad las veces que se realizan las transacciones.

Uno de los ejemplos más significativos de ello es el correspondiente a la consulta de seleccionar una mascota por el id, que se reduce de 2400 veces a tan solo 14 veces. Además el tiempo total que ocupan las transacciones de dicha consulta ha pasado de 6368,5 ms a 68,3 ms. Si nos fijamos bien, por cada caché que hemos implementado existe al menos un ejemplo similar al recién comentado.

Por ello, podemos decir que la optimización realizada cumple con creces con nuestras expectativas.

Resultados Gatling

Tras completar todas las optimizaciones a cada una de las historias de usuario, se ha realizado un análisis de Gatling para comprobar la mejora.

- Antes de realizar Optimización con 150 usuarios:



- Tras realizar Optimización con 1200 usuarios:



Pets

Las principales historias de usuarios que hacen uso completo de la entidad Pet son H25. Solicitar el registro de una nueva mascota y H21. Aceptar o rechazar un registro de mascota, vamos a centrarnos en ellas para la realización del Profiling y optimización.

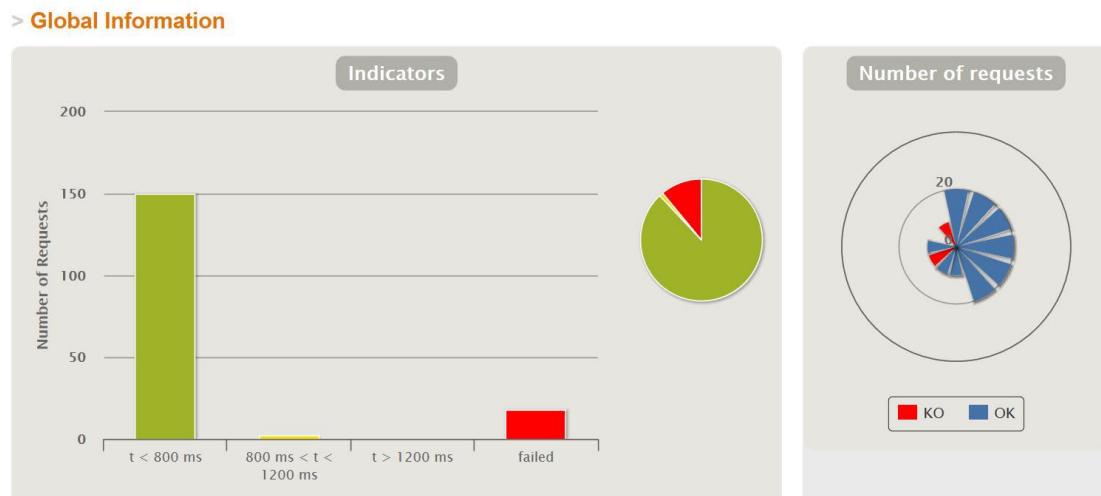
H21. Aceptar o rechazar un registro de mascota

Con el fin de comprender ¿Qué se ha hecho? ¿Por qué se ha hecho? y ¿Cómo se ha hecho? debemos comenzar explicando los resultados obtenidos con Gatling.

Esta historia de usuario trata de: Como auxiliar de clínica quiero poder aceptar o rechazar una solicitud de registro de mascota que ha realizado un cliente para poder controlar el número de mascotas con acceso a la clínica.

Primera implementación: El auxiliar de clínica puede visualizar el listado de solicitudes de mascotas en estado pendiente y desde este acceder a un formulario en el que puede aceptar o rechazar una mascota. Una vez aceptada o rechazada, el auxiliar de clínica no puede acceder de nuevo al formulario.

Primer análisis con Gatling: Como podemos observar en las siguientes capturas, hay 18 peticiones fallidas solo con 10 usuarios recurrentes.



▶ ASSERTIONS											
Assertion ↴											Status ↴
Global: max of response time is less than 5000.0											OK
Global: mean of response time is less than 1000.0											OK
Global: percentage of successful events is greater than 95.0											KO

▶ STATISTICS													
Requests ↴	⌚ Executions					⌚ Response Time (ms)							
	Total ↴	OK ↴	KO ↴	% KO ↴	Cnt/s ↴	Min ↴	50th pct ↴	75th pct ↴	95th pct ↴	99th pct ↴	Max ↴	Mean ↴	Std Dev ↴
Global Information	170	152	18	11%	1.197	2	15	30	64	417	872	33	93
Home	20	20	0	0%	0.141	8	15	26	43	43	43	19	10
Login	20	20	0	0%	0.141	2	6	8	10	10	10	6	2
request_2	20	20	0	0%	0.141	2	4	5	6	8	8	4	2
Logged	20	20	0	0%	0.141	4	10	14	16	17	17	10	5
Logged Redirect 1	20	20	0	0%	0.141	12	18	21	24	25	25	18	4
PetRequests	20	20	0	0%	0.141	37	52	61	99	99	99	56	17
ErrorRej...uestForm	10	10	0	0%	0.07	19	30	50	517	801	872	120	251
Accepted...uestForm	10	10	0	0%	0.07	12	21	45	475	752	821	106	239
AcceptedPetRequest	10	1	9	90%	0.07	3	5	7	100	161	176	22	51
ErrorRej...tRequest	10	10	0	0%	0.07	30	39	45	59	59	59	41	10
Accepted...direct 1	1	1	0	0%	0.007	37	37	37	37	37	37	37	0
Accepted..direct 1	9	0	9	100%	0.063	8	16	22	155	219	235	41	69

▶ ERRORS				
Error ↴			Count ↴	Percentage ↴
status.find.in(200,201,202,203,204,205,206,207,208,209,304), found 403			9	50 %
css((input[name=_csrf],Some(value))).find.exists, found nothing			9	50 %

Como se muestra en la captura anterior (errors), esto se debe a que se está tratando de buscar un “_csrf” que solo existe una vez. En concreto, la primera vez que se trata de aceptar o rechazar una mascota.

```

<div class="form-group">
    <div class="col-sm-offset-2 col-sm-10">
        <button class="btn btn-default" type="submit">
            Answer Request</button>
    </div>
</div>
<div>
<input type="hidden" name="_csrf" value="327ccb07-9150-424e-86b8-a402d8b16144" />
</div></form>

```

Tras estos resultados, se consideró necesaria la realización de Profiling y Optimización. ¿Eran excesivos los tiempos de consultas? ¿Podríamos mejorar el rendimiento con optimizaciones como N+1 o caché?

Profiling y optimización

Tras realizar el análisis con Glowroot, podemos observar que se consultan appointments, visits y stays cada vez que se solicita una mascota, a esto se denomina como el problema de N+1 queries.



Aun así, esto solo va a mejorar el tiempo de ejecución de las peticiones, pero no va a mejorar el número de usuarios concurrentes que pueden ejecutar satisfactoriamente la petición aceptar mascota.

Por tanto, el grupo decidió realizar una optimización del código mejorando el número de usuarios recurrentes y solventar el problema de las N+1 queries en la historia H25 mencionada anteriormente.

Con el fin de solventar el problema, se permitió al auxiliar de clínica visualizar la solicitud de la mascota en modo lectura, consiguiendo así que no fallaran los post de aceptarla o rechazarla si se intenta acceder más de una vez.

Pet Request

Name	Birth Date	Type	Status	Owner
George	2010/01/20	snake	ACCEPTED	George Franklin

Status: ACCEPTED

Justification: Estamos encantados de poder atender a su mascota.

[Return](#)

Resultados Gatling

La refactorización y las optimizaciones realizadas han permitido aumentar los usuarios concurrentes de 10 a 3500.

En la siguiente gráfica podemos observar que cada una de las 34000 peticiones fueron satisfactorias y en un tiempo menor a 800 milisegundos. Asimismo, se cumplen todas las condiciones propuestas.



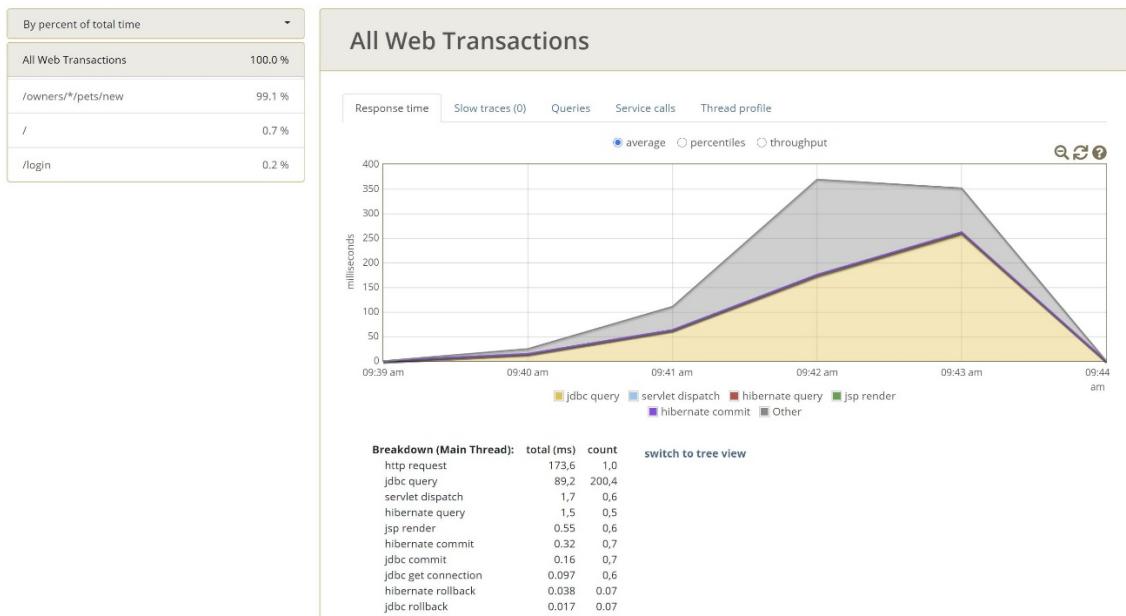
H25. Solicitar el registro de una nueva mascota

Se seleccionó esta historia para hacer Profiling porque el rendimiento óptimo era de 50 usuarios concurrentes y además por tener una relación directa con pet que, como anteriormente verificamos, tiene el problema de N+1 queries que afecta en varias partes del sistema.

Esta historia de usuario trata de: Como cliente quiero poder realizar una solicitud de registro de mascota para que pueda gozar de los servicios de la clínica.

Resultados Pre-Optimización

Al realizar el análisis con Glowroot observamos que existen muchas consultas innecesarias y con un tiempo de ejecución elevado, para mejorar el rendimiento se van a utilizar dos técnicas N+1 queries y caché.



All Web Transactions

	Total time ▾ (ms)	Total count	Avg time (ms)	Avg rows
select appointmen0_.pet_id as pet_id6_0_0_, appointmen0_.id as id1_0_0_, appointm... select visits0_.pet_id as pet_id4_17_0_, visits0_.id as id1_17_0_, visits0_.id as... select stays0_.pet_id as pet_id5_8_0_, stays0_.id as id1_8_0_, stays0_.id as id1_... select owner0_.id as id1_5_0_, pets1_.id as id1_6_1_, owner0_.first_name as first... select pet0_.id as id1_6_, pet0_.name as name2_6_, pet0_.active as active3_6_, pe... select medicaltes0_.visit_id as visit_id1_16_0_, medicaltes0_.medical_test_id as ... select specialtie0_.vet_id as vet_id1_14_0_, specialtie0_.specialty_id as special... select pettype0_.id as id1_12_, pettype0_.name as name2_12_ from types pettype0_ ... insert into pets (name, active, birth_date, justification, owner_id, status, type... select user0_.username as username1_13_0_, user0_.enabled as enabled2_13_0_, user... select username, password, enabled from users where username = ? select username, authority from authorities where username = ? select count(banner0_.id) as col_0_0_ from banners banner0_	29.136,0 15.796,9 15.233,7 1094,0 228,5 190,2 181,2 177,8 170,6 98,4 55,5 40,7 1,1 0,83 0,39	46,147 46,147 46,147 200 50 400 400 300 50 200 100 100 3 2 1	0.63 0.34 0.33 5,5 4,6 0.48 0.45 0.59 3,4 0.49 0.56 0.41 0.36 0.42 0.39	0.03 0.009 0.004 230,7 231,5 1,0 0,5 7,0 1,0 1,0 1,0 1,0 1,0 1,0 1,0 1,0 1,0

Como podemos observar en la captura anterior el número de consultas a visits, appointments y stays es muy elevado. En este caso, se está produciendo el problema de N+1 queries.

Cambios realizados:

En la clase de la entidad Pet se ha eliminado el fetch = FetchType.EAGER y no se ha añadido el LAZY pues por defecto el oneToMany es LAZY.

```

@OneToOne(cascade = CascadeType.ALL, mappedBy = "pet")
private Set<Visit> visits;

@OneToMany(cascade = CascadeType.ALL, mappedBy = "pet")
private Set<Appointment> appointments;

@OneToMany(cascade = CascadeType.ALL, mappedBy = "pet")
private Set<Stay> stays;

```

En el repositorio se han añadido o modificado las siguientes consultas:

```
@Query("SELECT DISTINCT p FROM Pet p LEFT JOIN FETCH p.visits v LEFT JOIN FETCH p.appointments a LEFT JOIN FETCH p.stays s WHERE p.id=:petId")
Pet findPetByIdWithVisitsAppointmentsStays(@Param("petId") Integer petId);

@Query("SELECT DISTINCT p FROM Pet p LEFT JOIN FETCH p.visits v LEFT JOIN FETCH p.appointments a LEFT JOIN FETCH p.stays s"
+ " WHERE p.status=:accepted AND p.active=:active AND p.owner.id=:ownerId")
List<Pet> findMyPetsAcceptedByActive(@Param("accepted") PetRegistrationStatus accepted, @Param("active") boolean active, @Param("ownerId") Integer ownerId);
```

Respecto al servicio, únicamente ha sido necesario modificar `findPetById`, pues se ha realizado una nueva consulta.

```
@Transactional(readOnly = true)
@Cacheable("petById")
public Pet findPetById(Integer petId) throws DataAccessException {
    return petRepository.findPetByIdWithVisitsAppointmentsStays(petId);
}
```

Por otro lado, para disminuir el número de transacciones con la base de datos, se han implementado dos cachés. En primer lugar, la correspondiente a la llamada de `petTypes` y, en segundo lugar, la referida a los clientes por el id.

La caché para los `petTypes` se realiza al solicitar todos los tipos de mascotas que se encuentran en la aplicación, en el servicio de `Pet` y se elimina en el caso de que se añada o edite un `petType`.

```
@Transactional(readOnly = true)
@Cacheable("petById")
public Pet findPetById(Integer petId) throws DataAccessException {
    return petRepository.findPetByIdWithVisitsAppointmentsStays(petId);
}

@CacheEvict(cacheNames = "petTypes", allEntries = true)
public void addPetType(PetType petType) throws DuplicatedPetNameException{
    if(!typeNameDontExists(petType.getName())) {
        throw new DuplicatedPetNameException();
    } else {
        petTypeRepository.save(petType);
    }
}
```

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

La caché para la solicitud del owner por id se realiza en el servicio del owner y se elimina en el caso de que se añada o edite un nuevo owner.

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

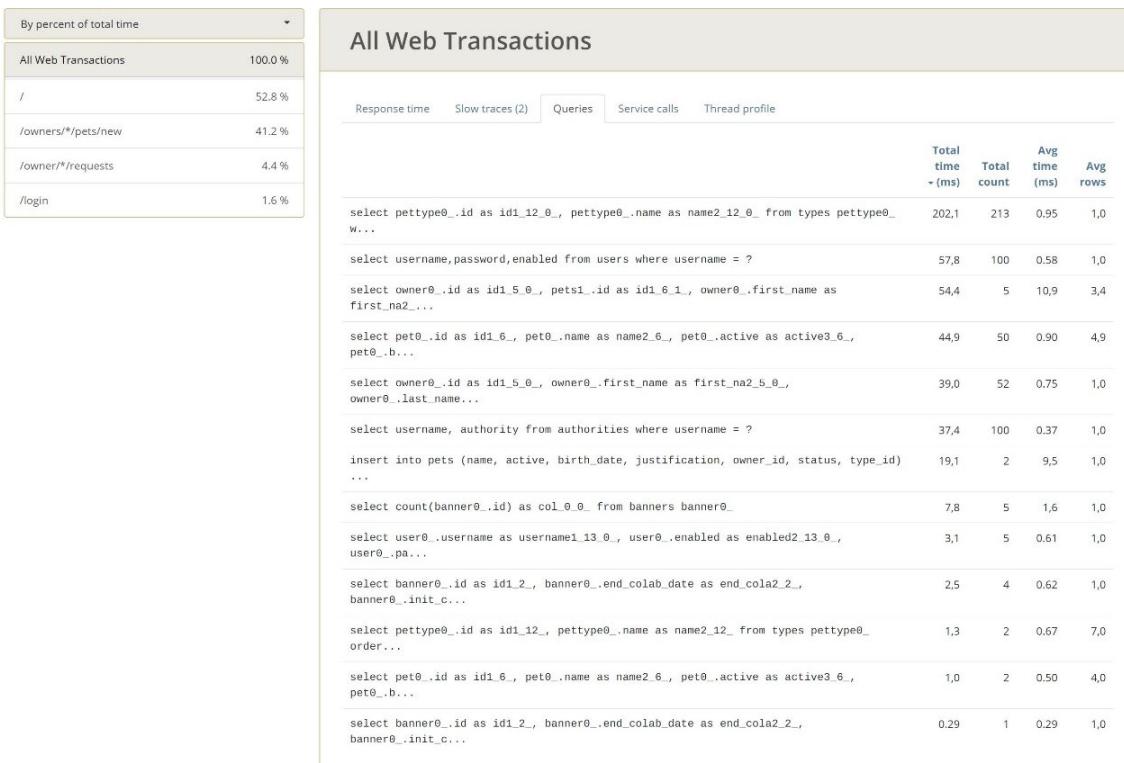
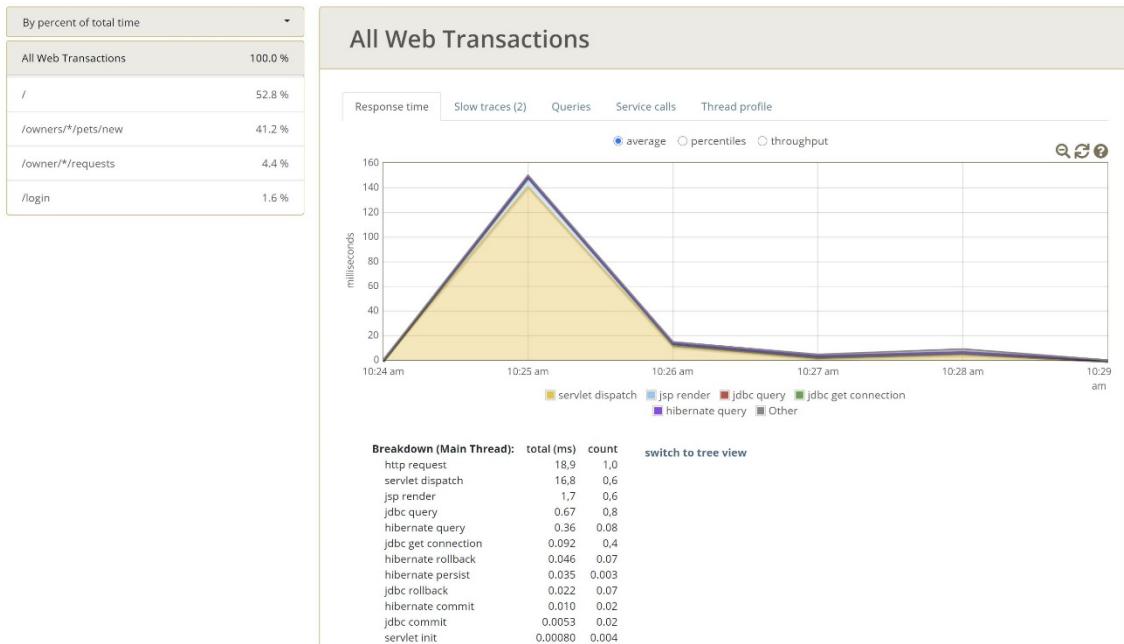
@Transactional
@CacheEvict(cacheNames = "ownerById", allEntries = true)
public void saveOwner(Owner owner) throws DataAccessException {
    //creating owner
    ownerRepository.save(owner);
    //creating user
    userService.saveUser(owner.getUser());
    //creating authorities
    authoritiesService.saveAuthorities(owner.getUser().getUsername(),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



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

Resultados Post-Optimización

Todas estas optimizaciones han mejorado el rendimiento de las historias que tiene relación con pet. En las siguientes capturas podemos observar la mejora de los tiempos de ejecución con Glowroot.



Podemos percibir una reducción en los tiempos notable. Por un lado, las consultas de appointments, visits y stays, no se realizan. Por otro lado, owner ha disminuido de 1094 a 55 milisegundos. Por último, las consultas a los petTypes han disminuido de 300 a 2 y con ello el tiempo de ejecución 178 de 1,3 milisegundo.

Resultados Gatling

Tras completar todas las optimizaciones a cada una de las historias de usuario, se ha realizado un análisis de Gatling para comprobar la mejora.

- Antes de realizar Optimización con 50 usuarios:



- Tras realizar Optimización con 5000 usuarios:

