G2-8 Design & Testing II

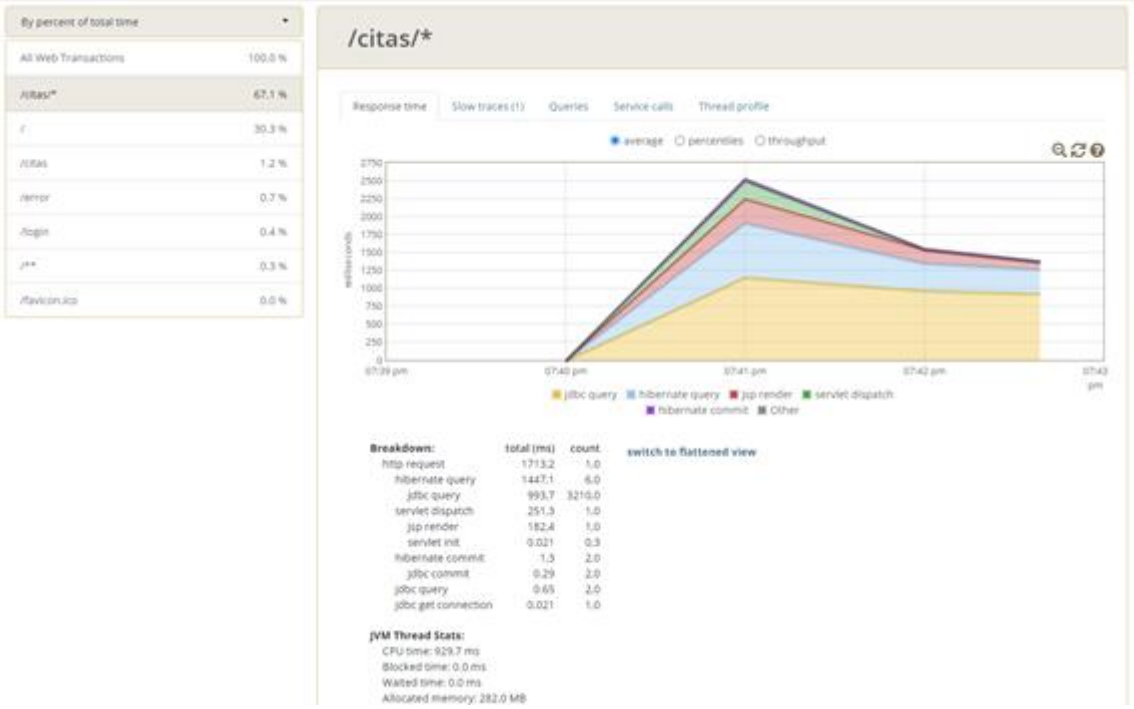
Documentación profiling realizadas:

Profiling #1: Listado y búsqueda de Citas

Uno de los casos que nos llamó la atención al realizar performance fue la búsqueda y listado de citas. Al no tener paginación en nuestro sistema, la query para mostrar todas las citas que un médico ha creado tenía un coste de transacción bastante alto, lo que se traducía en una experiencia no demasiado buena para nuestros usuarios. En la siguiente imagen podemos ver un trozo del report Crear Citas en el que observamos como esto ralentiza nuestro sistema.

ListPers...tasError	4900	4900	0	0%	16.013	0	4	7	1998	4626	10766	364	945
ListPers...tedCitas	4900	4900	0	0%	16.013	0	8	16	1955	5010	10941	377	990
ListPers...direct 1	762	762	0	0%	2.49	0	1	1	2	37	2868	8	131
ListPers...direct 2	8	8	0	0%	0.026	1243	2758	2764	3022	3133	3161	2619	537

Para confirmar esta sospecha hemos utilizado la herramienta Glowroot, la cual nos muestra el tiempo que tarda nuestra aplicación en realizar esta petición. En la imagen podemos ver como hay un pico inicial en la consulta. Esto es debido a que la primera vez que se accede al sistema el tiempo de respuesta es mayor. Al realizar la misma petición una segunda vez podemos ver como el tiempo de respuesta no mejora mucho en cuanto al pico inicial.



Esto se debe a la consulta que recoge todas las citas de nuestro sistema. Podemos observar cómo, de todas las consultas realizadas para este listado, la query que nos devuelve las citas para un determinado médico es la que consumen un mayor tiempo.

Esta consulta nos devuelve todo lo necesario para el listado, pero podemos observar cómo nos devuelve una gran cantidad de datos y requiere de varias relaciones entre tablas.

Para solucionar este problema creemos que la mejor solución es añadir una cache, la cual nos almacene durante un tiempo el resultado de esta consulta y no requiera de esta llamada a base de datos cada vez que se realiza este listado.

```
SELECT informe0_.id AS id1_3_4_,
       informe0_.cita_id AS cita_id4_3_4_,
       informe0_.diagnostico AS diagnost2_3_4_,
       informe0_.historia_clinica_id AS historia5_3_4_,
       informe0_.motivo_consulta AS motivo_c3_3_4_,
       cita1_.id AS id1_1_0_,
       cita1_.name AS name2_1_0_,
       cita1_.fecha AS fecha3_1_0_,
       cita1_.lugar AS lugar4_1_0_,
       cita1_.paciente_id AS paciente5_1_0_,
       paciente2_.id AS id1_6_1_,
       paciente2_.dni AS dni2_6_1_,
       paciente2_.apellidos AS apellido3_6_1_,
       paciente2_.domicilio AS domicili4_6_1_,
       paciente2_.email AS email5_6_1_,
       paciente2_.f_alta AS f_alta6_6_1_,
       paciente2_.f_nacimiento AS f_nacimi7_6_1_,
       paciente2_.medico_id AS medico_10_6_1_,
       paciente2_.n_telefono AS n_telefo8_6_1_,
       paciente2_.nombre AS nombre9_6_1_,
       historiacl3_.id AS id1_2_2_,
       historiacl3_.descripcion AS descripc2_2_2_,
       historiacl3_.paciente_id AS paciente3_2_2_,
       paciente4_.id AS id1_6_3_,
       paciente4_.dni AS dni2_6_3_,
       paciente4_.apellidos AS apellido3_6_3_,
       paciente4_.domicilio AS domicili4_6_3_,
       paciente4_.email AS email5_6_3_,
       paciente4_.f_alta AS f_alta6_6_3_,
       paciente4_.f_nacimiento AS f_nacimi7_6_3_,
       paciente4_.medico_id AS medico_10_6_3_,
       paciente4_.n_telefono AS n_telefo8_6_3_,
       paciente4_.nombre AS nombre9_6_3_
  FROM informe informe0_
       INNER JOIN cita cita1_ ON informe0_.cita_id = cita1_.id
       LEFT OUTER JOIN paciente paciente2_ ON cita1_.paciente_id = paciente2_.id
       LEFT OUTER JOIN historiaclinica historiacl3_ ON informe0_.historia_clinica_id = historiacl3_.id
       LEFT OUTER JOIN paciente paciente4_ ON historiacl3_.paciente_id = paciente4_.id
 WHERE informe0_.cita_id = ?
```

Para realizar esta tarea, lo primero que debemos hacer es identificar el método que realiza esta consulta y añadirle el nombre de la cache que lo almacenara.

```java
@Transactional(readOnly = true)
@Cacheable("findCitasPersonales")
public Collection<Cita> findCitasByMedicoId(final int medicoId) throws DataAccessException {
    Collection<Paciente> pacientes = new ArrayList<>();
    pacientes.addAll(this.citaRepo.findPacientesByMedicoId(medicoId));
    Collection<Cita> citas = new ArrayList<>();
    for (Paciente p : pacientes) {
        citas.addAll(this.citaRepo.findCitasByPacienteId(p.getId()));
    }
    return citas;
}
```

Tendremos que tener en cuenta que nuestra cache puede ser incoherente si se añade o elimina una de estas citas, por lo que debemos de recargarla cuando se realizan estas dos operaciones.

```java
@Transactional
@CacheEvict(cacheNames="findCitasPersonales", allEntries = true)
public Cita save(final Cita cita) throws InvalidAttributeValueException {
    if (cita.getFecha().isBefore(LocalDate.now())) {
        throw new InvalidAttributeValueException("No se puede poner una fecha en pasado");
    } else {
        return this.citaRepo.save(cita);
    }
}

@Transactional
@CacheEvict(cacheNames="findCitasPersonales", allEntries = true)
public void delete(final Cita cita) {
    this.citaRepo.delete(cita);
}
```

También debemos de añadir esta nueva cache al archivo de configuración previamente creado, asignándole el alias utilizado y la plantilla que queremos utilizar en este caso.

```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="findCitasPersonales" uses-template="default">
        <key-type>java.lang.Integer</key-type>
        <value-type>java.util.Collection</value-type>
    </cache>
</config>
```

Una vez implementado esto, realizaremos la misma consulta con Glowroot para comprobar si el sistema ha mejorado. Podemos observar el pico inicial el cual realiza cada vez que se accede por primera vez. Una vez realizado esto, la cache se llena con los datos de la consulta. Con esto mejoramos el rendimiento considerablemente, como se puede observar en la imagen inferior, dado que solo se realizará esta consulta cada 120 segundo, indicados en la configuración de la cache-

Si nos fijamos en las consultas realizadas por nuestro sistema, la primera vez que accedemos se realiza una consulta completa de las citas. Posterior mente, y por 2 minutos, esta consulta no es realizada a no ser que se haga un cambio en los datos, mejorando considerablemente el tiempo de respuesta.

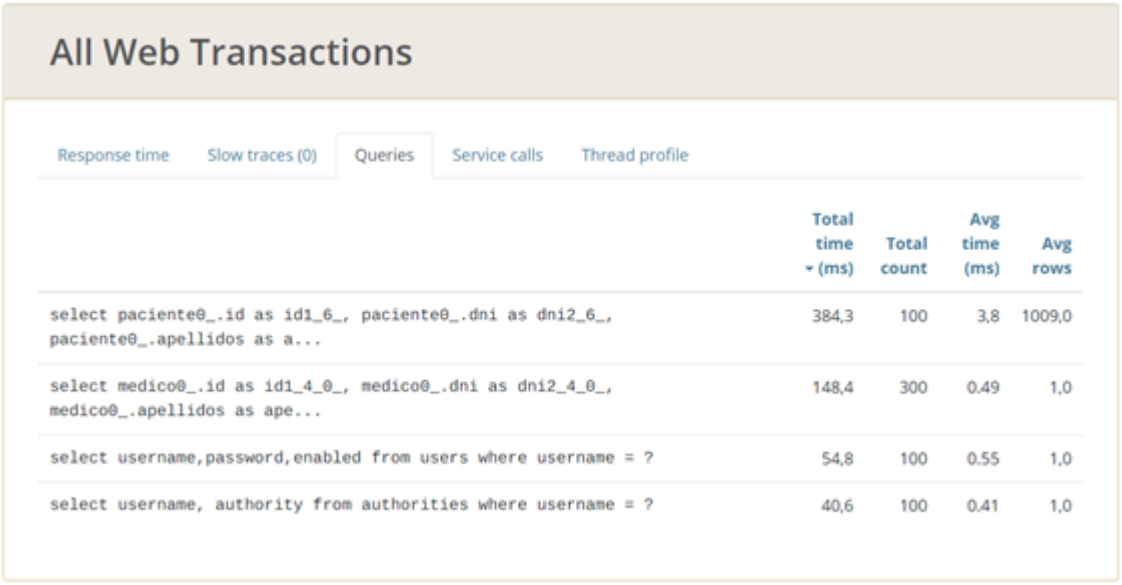
```
Hibernate:
    select
        informe0_.id as id1_3_4_,
        informe0_.cita_id as cita_id4_3_4_,
        informe0_.diagnostico as diagnost2_3_4_,
        informe0_.historia_clinica_id as historia5_3_4_,
        informe0_.motivo_consulta as motivo_c3_3_4_,
        cita1_.id as id1_1_0_,
        cita1_.name as name2_1_0_,
        cita1_.fecha as fecha3_1_0_,
        cita1_.lugar as lugar4_1_0_,
        cita1_.paciente_id as paciente5_1_0_,
        paciente2_.id as id1_6_1_,
        paciente2_.dni as dni2_6_1_,
        paciente2_.apellidos as apellido3_6_1_,
        paciente2_.domicilio as domicili4_6_1_,
        paciente2_.email as email5_6_1_,
        paciente2_.f_alta as f_alta6_6_1_,
        paciente2_.f_nacimiento as f_nacimi7_6_1_,
        paciente2_.medico_id as medico_10_6_1_,
        paciente2_.n_telefono as n_telefo8_6_1_,
        paciente2_.nombre as nombre9_6_1_,
        historiacl3_.id as id1_2_2_,
        historiacl3_.descripcion as descripc2_2_2_,
        historiacl3_.paciente_id as paciente3_2_2_,
        paciente4_.id as id1_6_3_,
        paciente4_.dni as dni2_6_3_,
        paciente4_.apellidos as apellido3_6_3_,
        paciente4_.domicilio as domicili4_6_3_,
        paciente4_.email as email5_6_3_,
        paciente4_.f_alta as f_alta6_6_3_,
        paciente4_.f_nacimiento as f_nacimi7_6_3_,
        paciente4_.medico_id as medico_10_6_3_,
        paciente4_.n_telefono as n_telefo8_6_3_,
        paciente4_.nombre as nombre9_6_3_
    from
        informe informe0_
    inner join
        cita cita1_
            on informe0_.cita_id=cita1_.id
    left outer join
        paciente paciente2_
            on cita1_.paciente_id=paciente2_.id
    left outer join
        historiaclinica historiacl3_
            on informe0_.historia_clinica_id=historiacl3_.id
    left outer join
        paciente paciente4_
            on historiacl3_.paciente_id=paciente4_.id
    where
        informe0_.cita_id=?
```

```
Hibernate:
    select
        user0_.username as username1_11_0_,
        user0_.enabled as enabled2_11_0_,
        user0_.password as password3_11_0_
    from
        users user0_
    where
        user0_.username=?
Hibernate:
    select
        medico0_.id as id1_4_,
        medico0_.dni as dni2_4_,
        medico0_.apellidos as apellido3_4_,
        medico0_.domicilio as domicili4_4_,
        medico0_.n_telefono as n_telefo5_4_,
        medico0_.nombre as nombre6_4_,
        medico0_.username as username7_4_
    from
        medico medico0_
    where
        medico0_.username=?
Hibernate:
    select
        user0_.username as username1_11_0_,
        user0_.enabled as enabled2_11_0_,
        user0_.password as password3_11_0_
    from
        users user0_
    where
        user0_.username=?
Hibernate:
    select
        medico0_.id as id1_4_,
        medico0_.dni as dni2_4_,
        medico0_.apellidos as apellido3_4_,
        medico0_.domicilio as domicili4_4_,
        medico0_.n_telefono as n_telefo5_4_,
        medico0_.nombre as nombre6_4_,
        medico0_.username as username7_4_
    from
        medico medico0_
    where
        medico0_.username=?
```

Profiling#2: Listado de pacientes

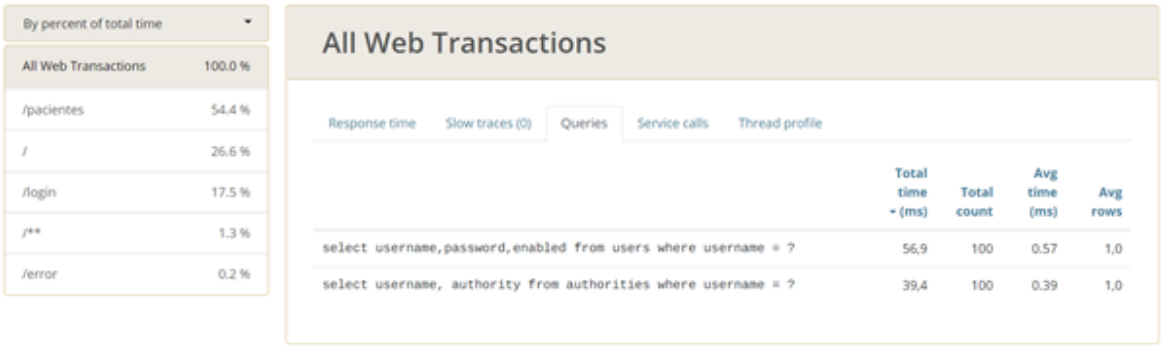
A continuación, se detallará el profiling realizado a la Historia de Usuario 4, correspondiente a listar todos los pacientes del sistema. Esta funcionalidad parece tener problemas de rendimiento cuando existen un número elevado de pacientes dentro del sistema, el cual, dado la naturaleza de este, no parará de crecer día tras día. Siendo esta una funcionalidad elemental en el proyecto y que será usada por todos los médicos registrados, resulta importante realizar un profiling para optimizar su rendimiento.

Así pues, se procedió a realizar un análisis de dicha funcionalidad con Glowroot insertando previamente 1.000 pacientes en el sistema con Gatling, un número que llegará a tener probablemente en no demasiado tiempo un sistema cuyo propósito principal es el almacenamiento de pacientes y datos relacionados con ellos. De este modo obtuvimos el siguiente informe de parte de Glowroot:

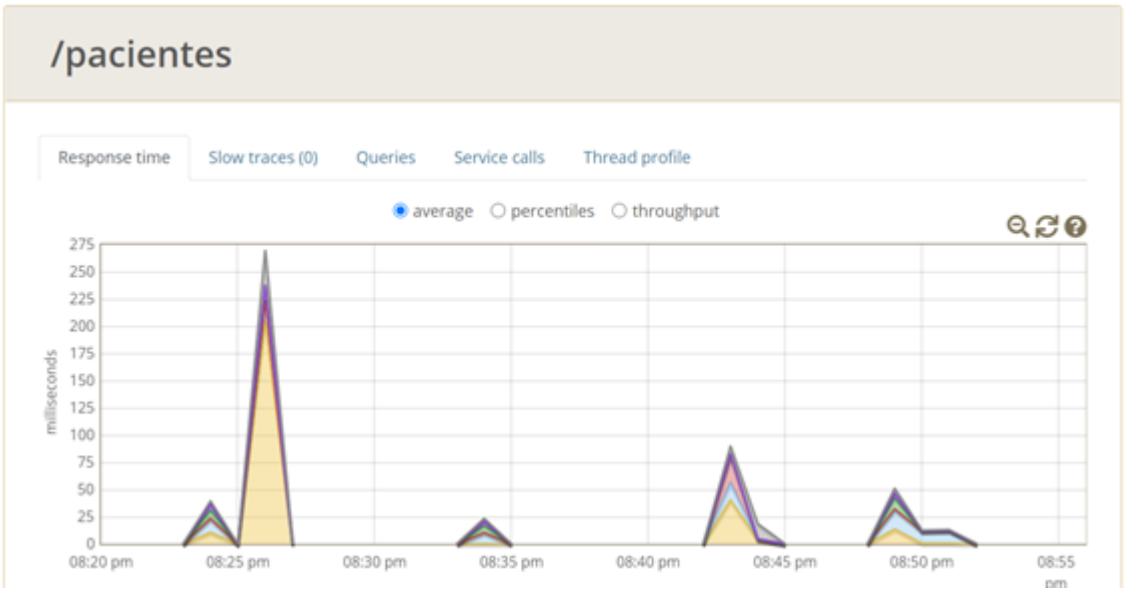


En esta imagen podemos observar las querys realizadas en el sistema de parte de 100 médicos que desean obtener la lista de pacientes registrados. Para empezar, ignoraremos las dos últimas querys pues pertenecen al logueo de los médicos del sistema. Así pues, observamos que las dos primeras querys pertenecen a la historia de usuario que queremos analizar. Podemos observar que se realizan dos querys: una correspondiente a los pacientes y otra correspondiente a los médicos. Además, observamos un problema significativo: se realizan el triple de consultas correspondiente a médicos que a pacientes. Esto tiene aún menos sentido si tenemos en cuenta que el listado general de pacientes ni siquiera muestra el médico asociado a cada uno de ellos. Por otra parte, observamos que el tiempo total que conlleva la consulta respectiva a los médicos es casi un tercio del tiempo total de consultas realizadas por esta funcionalidad. En definitiva, concluímos que podríamos mejorar el rendimiento de manera significativa si eliminábamos la consulta respectiva a los médicos que, en este caso, no tenía ningún propósito. Para ello, añadimos en el modelo de la clase paciente junto a *@ManyToOne* lo siguiente: *(fetch = FetchType.LAZY)*. De este modo conseguiremos que no se realice una consulta en base al médico del paciente cuando esta no sea necesaria. Con ello, volvimos a ejecutar el test y obtuvimos la siguiente información de Glowroot:

Podemos observar que las consultas respectivas a los médicos han desaparecido, reduciendo el tiempo total casi un tercio del total que obtuvimos previamente. Aún así en una funcionalidad tan importante y que irá aumentando tanto su tamaño según pase el tiempo, pues se irán añadiendo pacientes diariamente, resulta conveniente que su rendimiento sea el mejor posible. Por esta razón, pensamos que la implementación de caché para mejorar aún más su optimización resultaba una decisión beneficiosa para el sistema. De este modo, una vez realizada la implementación de caché para esta funcionalidad se volvieron a ejecutar los mismos tests y el resultado proporcionado por Glowroot fue el siguiente:



En esta imagen observamos que gracias a la caché, las querys han desaparecido. En la parte izquierda de la imagen podemos confirmar que se han realizado las consultas en cuestión, pues se ha accedido a la dirección correcta (*/pacientes*). Además se ha optado por introducir 120 segundos de tiempo de caché, dado que mantener la lista de pacientes actualizada es importante en nuestro sistema, pero este tiempo no causará inconvenientes a los usuarios que lo usen. De este modo concluímos el profiling de esta historia de usuario, habiendo alcanzado un rendimiento mucho mejor que el inicial. En la siguiente imagen podemos observar la evolución del rendimiento en base a los cambios realizados:



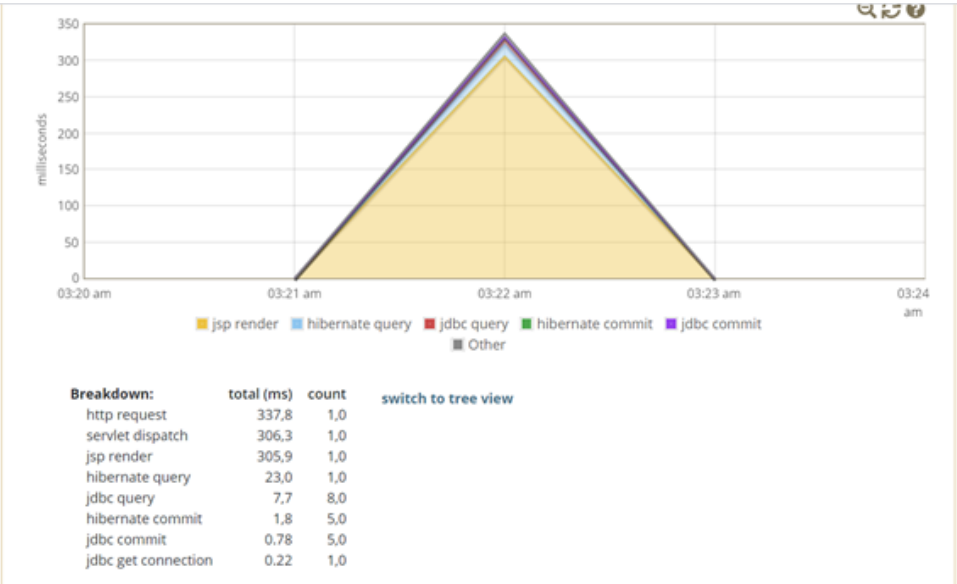
Si ignoramos los picos más pequeños de esta imagen (los cuales se corresponden a pruebas individuales para comprobar el correcto funcionamiento del sistema), podemos observar claramente tres picos significativos en la línea temporal. Cada uno de ellos se corresponden al código inicial y a los dos cambios realizados respectivamente. Así pues podemos concluir que cada uno de los cambios implantados ha conllevado un mejor rendimiento del sistema, finalizando en un rendimiento gratamente superior al que poseíamos inicialmente.

Profiling#3: Mostrar tratamientos de informe

Al realizar las pruebas de performance nos dimos cuenta de que uno de los casos más llamativos se producía al mostrar un informe que contenía muchos tratamientos pues la query devolvía todos los que un médico ha creado en el informe, así como el render de la vista debía mostrar un gran número de tratamientos en la vista de Informe. Esto se traduce en un coste computacional demasiado alto y provoca que la experiencia de usuario se viera afectada.

Estos problemas fueron detectados durante la realización del test de rendimiento correspondiente a la Historia de Usuario 20: Añadir Tratamiento. Dado que cuando añades un tratamiento eres redirigido a la vista de Informe, cada usuario al añadir un informe tarda un poco más que el anterior en acceder a la vista de Informe.

Para confirmar esta idea hemos utilizado un script de Gatling para poblar la base de datos con 5000 tratamientos en un Informe, otro script para realizar accesos por parte de 50 usuarios a este informe y finalmente hemos consultado los tiempos de respuesta en una herramienta de profiling (Glowroot).



Se ha de notar, que el tiempo que consume el render del jsp es muy alto y por tanto la vista debe mostrar todos los tratamientos asociados al Informe. La consulta también recoge todos los tratamientos del informe y consume una buena cantidad de tiempo. Podemos observar cómo, de todas las consultas realizadas para este listado, la query que no devuelve los tratamientos asociados a un respectivo informe es la que consume un mayor tiempo en ejecutarse.

	Total time (ms)	Total count	Avg time (ms)	Avg rows
select tratamient0_.id as id1_9_, tratamient0_.name as name2_9_, tratamient0_.dosis ...	239,0	50	4,8	5005,0
select informe0_.id as id1_3_0_, informe0_.cita_id as cita_id4_3_0_, informe0_.diagn...	55,6	100	0.56	1,0
select informe0_.id as id1_3_4_, informe0_.cita_id as cita_id4_3_4_, informe0_.diagn...	24,6	50	0.49	1,0
select cita0_.id as id1_1_0_, cita0_.name as name2_1_0_, cita0_.fecha as fecha3_1_0_...	20,8	50	0.42	1,0
select medico0_.id as id1_4_, medico0_.dni as dni2_4_, medico0_.apellidos as apellid...	15,5	50	0.31	1,0
select medico0_.id as id1_4_0_, medico0_.dni as dni2_4_0_, medico0_.apellidos as ape...	14,9	50	0.30	1,0
select user0_.username as username1_11_0_, user0_.enabled as enabled2_11_0_, user0_....	14,1	50	0.28	1,0

Esta consulta nos devuelve todo lo necesario para el listado, pero podemos observar cómo nos devuelve una gran cantidad de datos y requiere de varias relaciones entre tablas.

```
SELECT tratamient0_.id AS id1_9_,
       tratamient0_.name AS name2_9_,
       tratamient0_.dosis AS dosis3_9_,
       tratamient0_.f_fin_tratamiento AS f_fin_tr4_9_,
       tratamient0_.f_inicio_tratamiento AS f_inicio5_9_,
       tratamient0_.informe_id AS informe_7_9_,
       tratamient0_.medicamento AS medicame6_9_
  FROM tratamiento tratamient0_
```

Para solucionar este problema creemos que la mejor solución es añadir paginación.

Para añadir paginación hemos añadido un nuevo query con una respuesta Page (Tipo presente en las librerías de Sprint-Boot):

```
@Query("SELECT ALL t from Tratamiento t where t.informe.id = :id")
Page<Tratamiento> findTratamientoByInforme(int id, Pageable pageable) throws DataAccessException;
```

Un método de servicio que hace una llamada a este query:

```
@Transactional(readOnly = true)
public Page<Tratamiento> findTrata (int informeId, Pageable pageable){
    Page<Tratamiento> page = tratamientoRepo.findTratamientoByInforme(informeId, pageable);
    return page;
}       You, 25 minutes ago • paginacion
```

Hemos modificado la sección del método del controlador que se encarga de mostrar los informes para que añada al modelo los datos que requerimos (El contenido del Page y el número total de páginas) así como la cabecera para establecer los valores por defecto de la paginación.

```
@GetMapping(value = "/informes/{informeId}")
public ModelAndView showInforme(@PathVariable("informeId") final int informeId,
@PageableDefault(value = 5, page= 0) Pageable pageable){
```

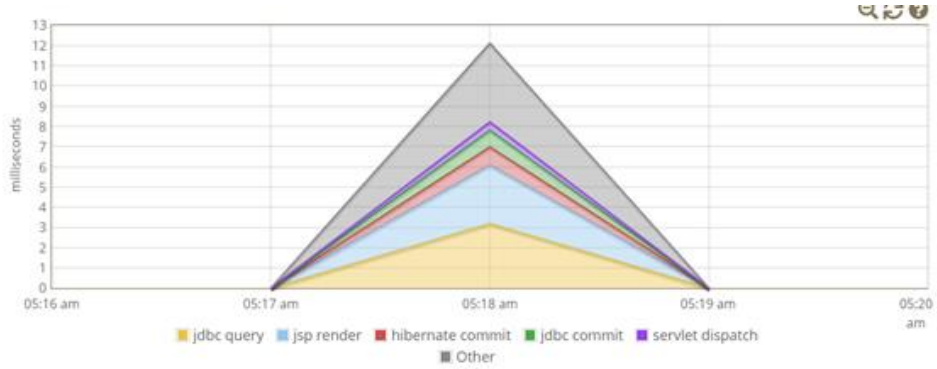
```
if(informe.getTratamientos() != null){
Page<Tratamiento> tratamientos = this.tratamientoService.findTrata(informeId, pageable);

mav.getModel().put("editTratamientoOk", cita.getFecha().equals(LocalDate.now()));
mav.getModel().put("tratamientos", tratamientos.getContent());
mav.getModel().put("tratapages", tratamientos.getTotalPages()-1);
    }       You, 2 days ago • paginationFix
```

Y finalmente hemos añadido el bucle que se encarga de añadir los botones para las páginas en la vista jsp:

```
<c:forEach begin="0" end="${tratapages}" var="current">
    <spring:url value="../informes/{informeId}?page=${current}" var="pageUrl">
    <spring:param name="informeId" value="${informe.id}" />
    </spring:url>
<a href="${fn:escapeXml(pageUrl)}" class="btn btn-default">Pagina ${current+1}</a>
</c:forEach>
```

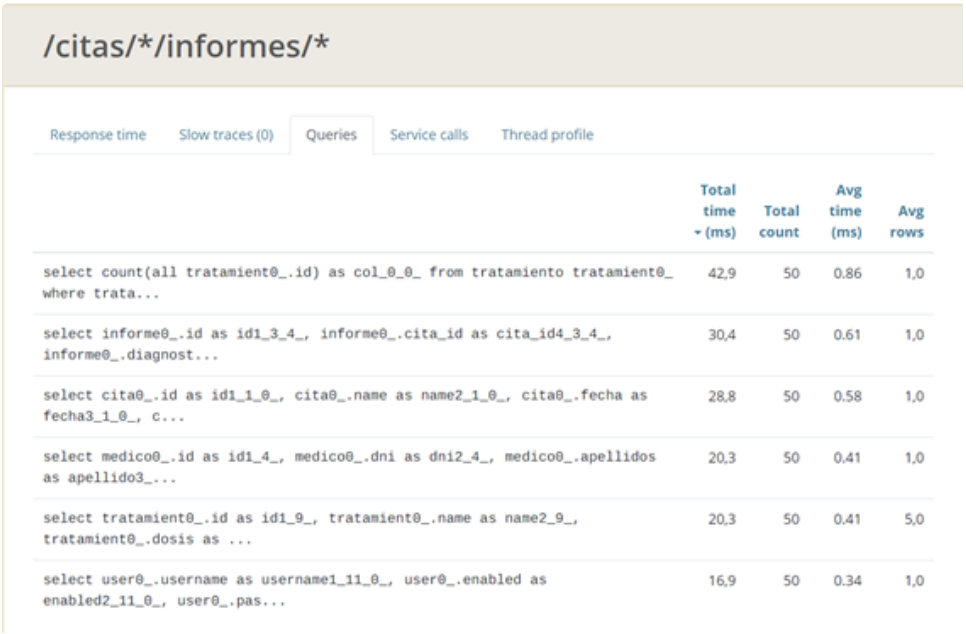
Tras realizar estos cambios, hemos poblado nuevamente la base de datos con 5000 tratamientos mediante un script de Gatling y posteriormente hemos usado otro script para realizar accesos por parte de 50 usuarios. Así, recibimos estos nuevos datos de la herramienta Glowroot:



Breakdown:	total (ms)	count
http request	12,1	1,0
servlet dispatch	3,3	1,0
jdbc query	3,2	6,0
jsp render	2,9	1,0
hibernate commit	1,7	5,0
hibernate query	1,5	2,0
jdbc commit	0.83	5,0
jdbc get connection	0.19	1,0

switch to tree view

Como podemos observar, tras realizar los cambios y añadir paginación, los tiempos de respuesta se reducen drásticamente. De esta forma conseguimos que mejore la experiencia del usuario al usar el sistema. En la imagen podemos ver que el tiempo total de ‘’jdbc query’’ ha bajado hasta los 3,2ms y su count; mientras que el tiempo total que consume el render jsp ha sido enormemente reducido hasta 2,9 ms.



/citas/*/informes/*

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

	Total time (ms)	Total count	Avg time (ms)	Avg rows
select count(all tratamient0_.id) as col_0_0_ from tratamiento tratamient0_ where trata...	42,9	50	0.86	1,0
select informe0_.id as id1_3_4_, informe0_.cita_id as cita_id4_3_4_, informe0_.diagnost...	30,4	50	0.61	1,0
select cita0_.id as id1_1_0_, cita0_.name as name2_1_0_, cita0_.fecha as fecha3_1_0_, c...	28,8	50	0.58	1,0
select medico0_.id as id1_4_, medico0_.dni as dni2_4_, medico0_.apellidos as apellido3_...	20,3	50	0.41	1,0
select tratamient0_.id as id1_9_, tratamient0_.name as name2_9_, tratamient0_.dosis as ...	20,3	50	0.41	5,0
select user0_.username as username1_11_0_, user0_.enabled as enabled2_11_0_, user0_.pas...	16,9	50	0.34	1,0

En esta imagen podemos observar que el tiempo de ejecución de dicha query también se ha reducido hasta un tiempo medio de 0.41ms.

```
SELECT tratamient0_.id AS id1_9_,
       tratamient0_.name AS name2_9_,
       tratamient0_.dosis AS dosis3_9_,
       tratamient0_.f_fin_tratamiento AS f_fin_tr4_9_,
       tratamient0_.f_inicio_tratamiento AS f_inicio5_9_,
       tratamient0_.informe_id AS informe_7_9_,
       tratamient0_.medicamento AS medicame6_9_
  FROM tratamiento tratamient0_
 WHERE tratamient0_.informe_id = ?
 LIMIT ?
```