Poster

Memoria del proceso de desarrollo de la práctica

Herramientas de Desarrollo
Grado en Ingeniería Informática

Alejandro Buján Pampín	alejandro.bujan.pampin@udc.es
Alfredo Javier Freire Bouzas	javier.freire.bouzas@udc.es
Martín Regueiro Golán	martin.regueiro1@udc.es
Sergio Goyanes Legazpi	sergio.legazpi@udc.es
Patricia Mato Miragaya	patricia.mato.miragaya@udc.es
Daniel Jueguen Pérez	d.jueguen@udc.es

Diciembre 2023

Introducción

El presente documento constituye la memoria técnica del proyecto de desarrollo de Poster, una aplicación basada en Spring Boot y React, orientada a la difusión de ofertas y cupones.

Este informe detalla la arquitectura, el proceso de desarrollo y las herramientas clave utilizadas durante todas las etapas del proyecto. Se ofrece una visión general del entorno de desarrollo, así como el impacto y la contribución de las herramientas empleadas en la consecución de los objetivos establecidos.

Tecnologías utilizadas

Para el backend, se ha empleado Spring Boot. Este framework de Java proporciona una base robusta para el desarrollo de aplicaciones, permitiendo crear servicios web de manera rápida, eficiente, escalable y segura, que además ha demostrado amplia compatibilidad y está respaldado por una amplia gama de bibliotecas. Para el acceso a bases de datos relacionales, se usa JPA (Java Persistence API).

Por otro lado, para el frontend se ha desarrollado bajo React, (respaldada por Facebook), siendo su principal ventaja su modularidad y su enfoque en componentes reutilizables. Este framework lo hemos combinado con Redux para la gestión del estado.

Arquitectura

Nuestro sistema se ha ajustado a una arquitectura en capas típica de una SPA (Single Page Application):

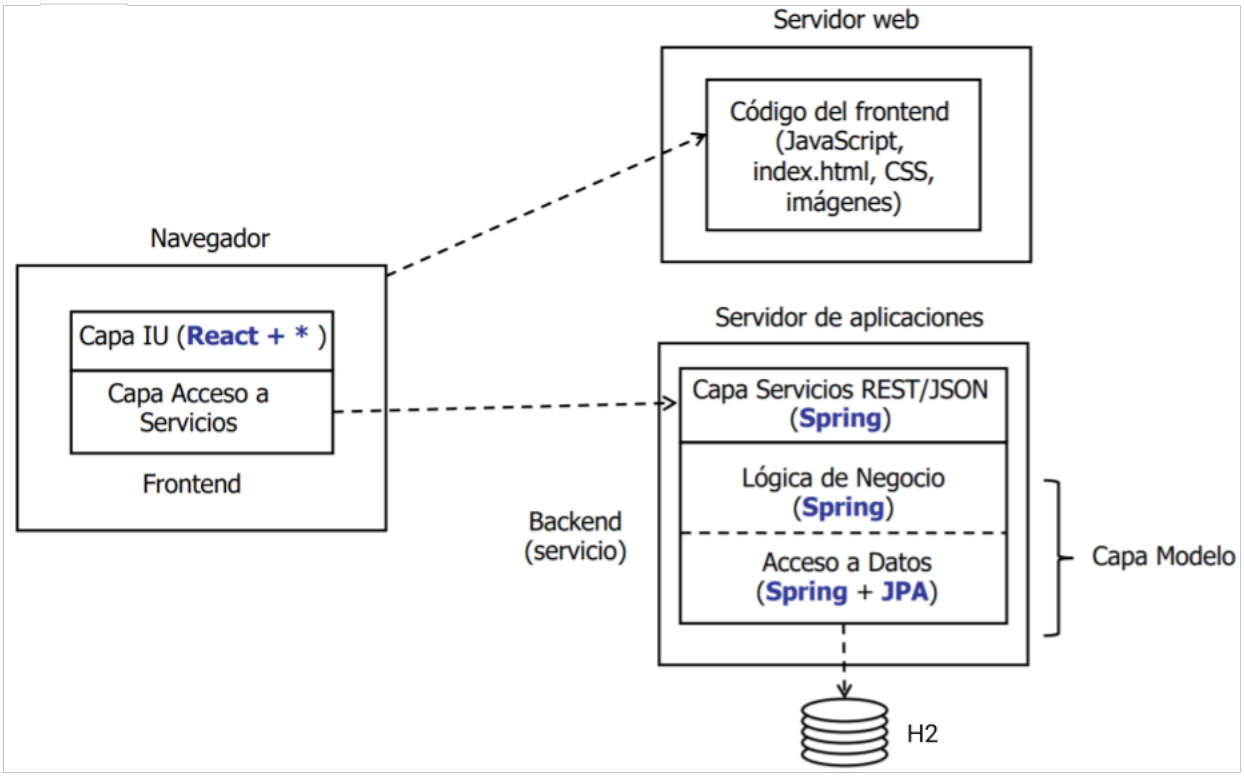

Figura 1: Arquitectura del sistema Poster

Esta arquitectura consta de:

- Un frontend JavaScript que se ejecuta en el navegador.
- Un backend (servicios REST/JSON + capa Modelo) al que accede el frontend.

La IU del frontend reacciona a los eventos causados por el usuario:

- Realiza una petición al backend a través de la capa de Acceso a Servicios (e.g. búsqueda de posts).
- Manipula el árbol DOM de la página que muestra el navegador usando los datos devueltos por el servicio (e.g. construye una tabla con los resultados de la búsqueda y la inserta en el árbol DOM).

Modelo de datos

Diseño conceptual

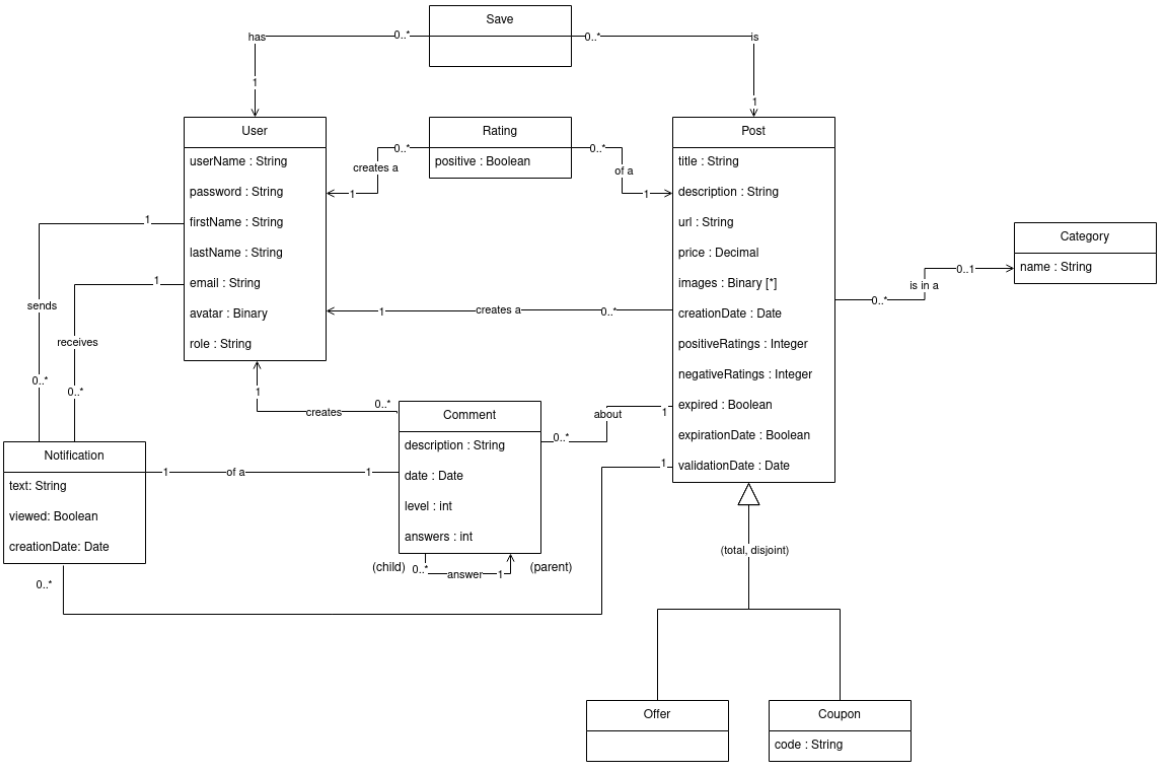

Figura 2: Modelado de datos UML del sistema Poster

Diseño lógico (Modelo relacional)

- **User**(**id**, userName, password, firstName, lastName, email, avatar, role)
- **Category**(**id**, name)
- **Post**(**id**, title, description, url, price, creationDate, positiveRatings, negativeRatings, expirationDate, validationDate, userId, categoryId)
  - **userId** referencia **User**(**id**) ON UPDATE CASCADE ON DELETE CASCADE
  - **categoryId** referencia **Category**(**id**) ON UPDATE CASCADE ON DELETE SET NULL

  Nota: Para la gestión de herencia, hemos escogido la estrategia JOINED (aunque no sea la opción más eficiente) como una opción a largo plazo debido a su mejor mantenibilidad en relación a la evolución de los requisitos, ya que este proyecto se ha llevado a cabo en un entorno de cambios continuos.

- **Offer**(**id**)
  - **id** referencia **Post**(**id**) ON UPDATE CASCADE ON DELETE CASCADE
- **Coupon**(**id**, code)
  - **id** referencia **Post**(**id**) ON UPDATE CASCADE ON DELETE CASCADE
- **Image**(**id**, data, postId)
  - **postId** referencia **Post**(**id**) ON UPDATE CASCADE ON DELETE CASCADE
- **Rating**(**id**, positive, userId, postId)
  - **userId** referencia **User**(**id**) ON UPDATE CASCADE ON DELETE CASCADE
  - **postId** referencia **Post**(**id**) ON UPDATE CASCADE ON DELETE CASCADE
- **Comment**(**id**, description, date, level, answers, userId, postId, parentId)
  - **userId** referencia **User**(**id**) ON UPDATE CASCADE ON DELETE CASCADE
  - **postId** referencia **Post**(**id**) ON UPDATE CASCADE ON DELETE CASCADE
  - **parentId** referencia **Comment**(**id**) ON UPDATE CASCADE ON DELETE CASCADE
- **Notification**(**id**, text, viewed, creationDate, notifierUserId, notifiedUserId, commentId, postId)
  - **notifierUserId** referencia **User**(**id**) ON UPDATE CASCADE ON DELETE CASCADE
  - **notifiedUserId** referencia **User**(**id**) ON UPDATE CASCADE ON DELETE CASCADE
  - **postId** referencia **Post**(**id**) ON UPDATE CASCADE ON DELETE SET NULL
  - **commentId** referencia **Comment**(**id**) ON UPDATE CASCADE ON DELETE SET NULL
- **Save**(**id**, postId, userId)
  - **postId** referencia **Post**(**id**) ON UPDATE CASCADE ON DELETE SET NULL
  - **userId** referencia **User**(**id**) ON UPDATE CASCADE ON DELETE CASCADE

Patrones de diseño

Durante el desarrollo del proyecto, se ha tenido muy en cuenta el uso de buenas prácticas y patrones de diseño, de forma que el código sea altamente mantenible (debido a la naturaleza de requisitos cambiantes que tenía la práctica).

En general, se ha procurado aprovechar las capacidades del lenguaje Java, como el polimorfismo y la ligadura dinámica, y la ocasión perfecta para hacerlo fue durante la gestión de las operaciones con herencia.

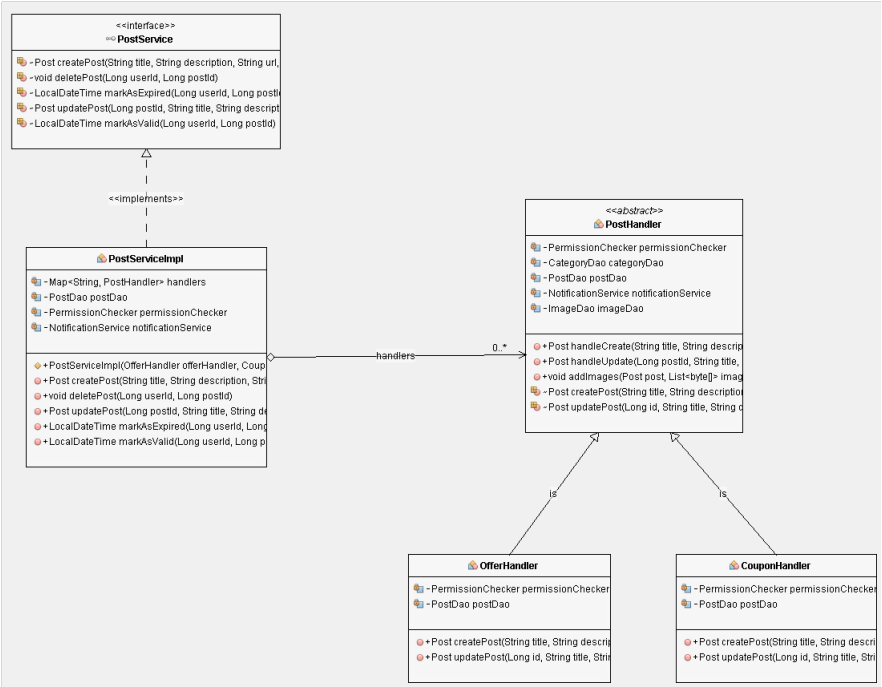

Figura 3: Diseño UML de la gestión de las operaciones específicas de herencia

El patrón estrategia se ha utilizado para manejar la creación y actualización de diferentes tipos de Posts (Coupon y Offer) mediante la implementación de la interfaz PostHandler y sus clases concretas CouponHandler y OfferHandler. Esto permite que la implementación del servicio PostService invoque el método create o update de la estrategia apropiada según el tipo de Post que se esté creando, sin preocuparse por la implementación específica.

Además, el uso de un mapa con clave String y valor PostHandler handler) para almacenar las diferentes estrategias basadas en su tipo (en este caso, Çouponz .ºffer") y luego recuperar la estrategia correcta según el tipo de Post a crear, agrega flexibilidad al diseño, permitiendo manejar diferentes estrategias de manera dinámica y eficiente.

Además, el enfoque se ha reforzado mediante el patrón de Método Plantilla (Template Method), para evitar duplicaciones de código en ambas subclases de PostHandler ya que las operaciones tienen una parte claramente generalizable.

En la clase abstracta PostHandler, los métodos abstractos createPost y updatePost actúa como métodos plantilla. Este método contiene la lógica común para la manipulación de la creación y actualización de posts, pero delegan las operaciones de tipo específico (Offer o Coupon) a las clases hijas implementando el método abstracto createPost y updatePost.

Esto proporciona una estructura flexible donde la lógica común está en la superclase, pero las subclases tienen la responsabilidad de proporcionar la implementación específica según su tipo, como se ve en la figura 3.

Seguridad

En el proceso de desarrollo de Poster, se han implementado ciertas medidas de seguridad para prevenir posibles vulnerabilidades.

Dos aspectos clave abordados en este proyecto son el manejo de sesiones mediante tokens JWT (JSON Web Tokens) y la prevención de la inyección SQL en la capa de acceso a datos, donde ha implementado el uso de consultas parametrizadas y la validación estricta de entrada de datos para prevenir la ejecución de comandos maliciosos en las consultas SQL.

Pruebas funcionales

Hemos estructurado nuestras pruebas en cuatro grupos fundamentales:

- **Tests de la capa modelo (backend)**: Implementados con JUnit y apoyándose eventualmente en Mockito, donde se prueba la lógica de negocio.
- **Tests de la capa de servicios REST/JSON (backend)**: Implementados con JUnit y MockMvc, probando las solicitudes y respuestas HTTP correspondientes.
- **Tests de la capa de acceso a servicios (frontend)**: Implementados en Jest, donde se prueba que la gestión de las solicitudes externas es la adecuada.
- **Tests de la interfaz de usuario y componentes (frontend)**: Implementados con Jest y React Testing Library, basándose en el renderizado condicional de componentes y apoyándose en el uso de snapshots y mocks del estado global.

Entorno de desarrollo integrado

Durante el desarrollo del proyecto, el equipo se ha apoyado en **Eclipse Enterprise Edition**, con el que hemos experimentado una serie de ventajas y desventajas en su uso que comentaremos a continuación.

Eclipse tiene una amplia gama de ventajas que facilitan la codificación rápida y eficaz del proyecto desarrollado. En primer lugar, destacan todas las posibilidades que ofrece a la hora de depurar el código, ya que dota de un gran control al programador para establecer las condiciones de parada y de observación de las variables. Otro de los aspectos destacables de este entorno de desarrollo es la enorme variedad de plugins que permiten ampliar la funcionalidad base del mismo, como por ejemplo el plugin **EGit**, que nos permite el cambio y creación de ramas, realización sencilla de commits y operaciones como merge, visualización clara del grafo de commits... También fueron empleadas **Mylyn** y **Timekeeper**, que nos permiten gestionar las tareas a realizar y su seguimiento de los tiempos. Algo muy usado por el equipo de desarrollo ha sido el formateo de las líneas de código y la adición o eliminación de imports automáticamente. Algo muy destacable de esta herramienta es la organización de la interfaz mediante perspectivas y vistas, que muestra todos los elementos relacionados, agrupados para cada situación específica que puede ocurrir codificando. Por ejemplo, algunas de las vistas más útiles para el equipo han sido la vista de errores y warnings y la de javadoc.

A pesar de las ventajas mencionadas anteriormente, también hemos encontrado múltiples desventajas, siendo la más importante el amplio consumo de recursos, no escalando bien ante un mayor número de plugins instalados, dando lugar a que en ocasiones su funcionamiento fuese contraproducente para el desarrollo del proyecto. Por otro lado, al inicio del proyecto el equipo tuvo que emplear una cantidad importante de tiempo para aprender a usar correctamente el IDE, puesto que tiene una curva de aprendizaje más alta en comparación con otros IDEs.

Eventualmente, ha habido ocasiones en la que, como se recomendaba en la documentación del arquetipo dado, el equipo también se ha apoyado en **VSCode** para el desarrollo del frontend

Control de versiones y seguimiento de issues

En este apartado, desde la segunda iteración (Redmine sólo desde la última semana de ésta) aplicamos un flujo de trabajo basado en GitFlow, donde se encapsulaban las funcionalidades o issues en una rama específica en la fase de desarrollo, para luego dar lugar a ramas de release y finalmente main (siendo los errores encontrados en producción tratados en ramas hotfix).

En Redmine, se registraron como issues las propias funcionalidades, que se dividirían en subissues que harían referencia a pequeñas partes de trabajo con entidad propia dentro de la funcionalidad, y si era el caso, se establecían las precedencias adecuadas cuando el desarrollo de ciertas tareas limitaba o bloqueaba el comienzo o finalización de otra de ellas.

La rama en la que se trataba la resolución de las issues se establecía en base a la siguiente tabla:

<table>
<tr><th>Tipo de Issue</th><th>Descripción</th><th>Afecta a</th><th>Prioridad</th><th>Nombre de rama</th></tr>
<tr><td rowspan="2">Bug</td><td rowspan="2">Tarea que implica la solución de un error encontrado en la aplicación</td><td>Cualquiera (en producción)</td><td>Inmediata</td><td>hotfix/xxxx</td></tr>
<tr><td>Varias funcionalidades (en desarrollo)</td><td rowspan="3">Variable</td><td>bug/xxxx</td></tr>
<tr><td>Feature</td><td>Tarea que implica el desarrollo o ampliación de una funcionalidad específica</td><td>Una funcionalidad en específico (en desarrollo)</td><td>feature/feature-name</td></tr>
<tr><td>Support</td><td>Tarea que da soporte al sistema globalmente</td><td>La aplicación en su conjunto</td><td>support/xxxx</td></tr>
</table>

Figura 4: Correspondencia issue-ramas (siendo *xxxx* el id de issue, y *feature-name* el nombre de la funcionalidad)

Como norma general, solamente un miembro del equipo de trabajo era el responsable del desarrollo de una issue, y su merge request sería revisada por otro o otros miembros distintos al desarrollador.

Durante el desarrollo de cada una de estas tareas, se referenciaba la issue asociada en cada commit, así como el tiempo empleado según el trackeo del plugin de TimeKeeper de Eclipse.

Integración continua e inspección continua

Hemos usado Jenkins para llevar a cabo la integración continua del proyecto. Para ello, se creó un job asociado a nuestro proyecto. En dicho job, se configuró una serie de parámetros para su correcto funcionamiento. Entre dichos parámetros cabe destacar: la integración con GitLab a través de un webhook, que permita la ejecución de una build cuando se produzcan push events sobre la rama main, develop o release(s) o cuando se produzcan eventos sobre una merge request abierta o la integración con Redmine. También se configuró la integración con Kubernetes, como se explicará en su sección correspondiente de esta memoria.

También hemos usado Sonar para medir la calidad de un código dado a partir de una serie de criterios establecidos tales como duplicaciones, cobertura,... y se configuró conjuntamente con Jenkins con el objetivo de analizar el código en cada build dada bajo ciertas condiciones (ramas main, develop y release, además de merge requests abiertas).

Para ello, tuvimos que confgurar Sonar a través de la configuración del Job de Jenkins asociado a nuestro proyecto. Por una parte le indicamos a Jenkins que hiciera checkout a las ramas locales de forma que, posteriormente, se pueda informar a Sonar de qué rama o merge request se buscaba analizar. También definimos la siguiente configuración específica de Sonar:

```
# Project properties
sonar.projectKey=es.udc.fi.dc.fd:poster
sonar.projectName=poster
sonar.projectVersion=0.1-SNAPSHOT

# Source configuration
sonar.sourceEncoding=UTF-8
sonar.language=java, js
sonar.inclusions=src/main/java/**/*.java, frontend/src/**/*.js
sonar.exclusions= frontend/src/tests/**/*.test.js, frontend/src/
    registerServiceWorker.js, frontend/src/tests/**/*.mock.js,
sonar.java.binaries=target/classes/
sonar.java.libraries=

# Test configuration
sonar.test.inclusions=src/test/java/**/*.java, frontend/src/tests/**/*.test.js

#Test coverage
sonar.java.coveragePlugin=jacoco
sonar.javascript.lcov.reportPaths=frontend/coverage/lcov.info
```

Las tres primeras líneas configuran, respectivamente, la clave, el nombre y la versión del proyecto.

Luego, se configura la codificación de los caracteres, los lenguajes de programación (java y js), los directorios a analizar y a excluir por parte de Sonar, dónde se encuentran los binarios java y las librerías asociadas.

Por último, se indican los directorios donde están los tests, la herramienta usada para la cobertura en el backend (Jacoco en nuestro caso, no siendo necesario aclarar la ubicación de los reports ya que Sonar la busca automáticamente) y también se aclaran dónde están los reports de cobertura del frontend.

Es importante aclarar que nunca se ha mergeado una rama que no cumpliese los requisitos de calidad establecidos, y en el conjunto global del proyecto (overall code) estos requisitos también se han garantizado.

Finalmente, cabe mencionar que durante el desarrollo del proyecto se ha integrado dicha herramienta con Eclipse a través del plugin de SonarLint.

Build automation

La configuración de Maven se ha llevado a cabo, como es normal, en el pom.xml. Los aspectos más relevantes (no todos) se introducen aquí:

Al proyecto se ha añadido un plugin que nos permite trabajar con WebSockets, que abren una conexión persistente y son usados en el proyecto para notificar a un usuario que se encuentra en el feed, en caso de que se añada un nuevo post.

Con el objetivo de obtener un informe con la cobertura de los test cada vez que se ejecute la aplicación, se ha decidido incluír en el proyecto el plugin de JaCoco, una herramienta utilizada para medir la cobertura de código Java.

Por otro lado, existen en el proyecto una serie de dependencias relacionadas con Json Web Tocken, como son jjwt-api, jjwt-impl o jjwt-jackson.

Como base de datos, se ha decidido utilizar H2 debido a sus características y sencillez. En nuestro caso, la base de datos está en fichero, lo que implica que cada vez que se lanza la aplicación, los datos no se pierden. Por el contrario, para la base de datos usada en los tests, se crea una instancia en memoria, ya que nos es indiferente la persistencia de los mismos, puesto que los datos creados en cada test son eliminados al finalizar para evitar conflictos con los demás.

Por otro lado, ha sido necesario utilizar ciertas dependencias para validación, como jakarta.validation-api e hibernate-validator. Con el objetivo de facilitar la codificación y el encapsulamiento, y reducir el número de líneas de código, se ha decidido hacer uso del Proyecto Lombok, incluyendo la dependencia correspondiente.

En lo que respecta a la localización de los recursos del perfil test, esta se ha establecido en la ruta src/test/resources.

Respecto al plugin de frontend de Maven, utilizará Node.js y Yarn, que, tras instalarse, ejecutarán inicialmente yarn test (mostrando la cobertura) y, a continuación, yarn build.

Para realizar el despliegue, se incluye el plugin de Kubernetes.

Gran parte de estas herramientas se verán involucradas y detalladas en otras explicaciones a lo largo de la presente memoria.

Análisis de rendimiento

En este punto se utilizó la herramienta Java Mission Control (JMC), que permite ver fugas de procesamiento o memoria, junto con JMeter para así obtener métricas más realistas. Durante la realización de dichas pruebas, se empleó el mismo equipo que el utilizado para las pruebas de rendimiento. Lo realizado consistió en crear un flight recording, que consiste en una grabación temporal del consumo de recursos y eventos que suceden en nuestra aplicación para su posterior análisis, en base a la siguiente configuración:

Característica	**Valor**
Duración	30 segundos
Garbage collector	Normal
Allocation profiling	Medium
Compiler	Detailed
Method profiling	Maximum
Thread dump	Every 60s
Exceptions	Errors only
Memory Leak Detection	Object Types + Allocation Stack traces
Locking / File I/O / Socket I/O	10 ms

Cuadro 1: Configuración de parámetros

En cuanto a los datos mostrados en las figuras posteriores, podemos observar que a pesar de que en esos 30 segundos se estan lanzando peticiones constantemente y sin ningun retraso entre ellas, observamos que: los parámetros del Garbage Collector no presentan ningún error grave puesto que, por ejemplo ante tantas peticiones, solo se hace 1 pausa y ninguna limpieza debido a las inspecciones de la pila; los parámetros de la Memoria eran todos correctos y no se observan errores medios ni graves y los parámetros de CPU muestran que hay competencia entre procesos por la CPU, pero suponemos que esto es debido a la gran cantidad de peticiones que se hacen al mismo tiempo a nuestra aplicación.

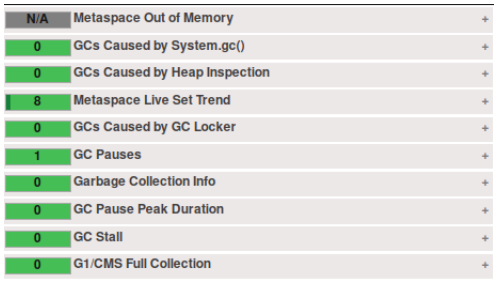

Figura 5: Análisis del Garbage Collector

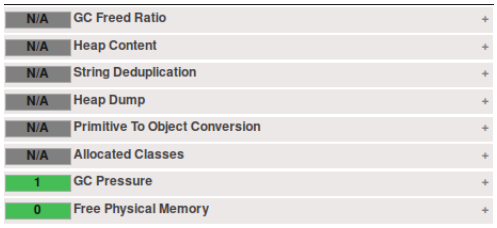

Figura 6: Análisis de memoria

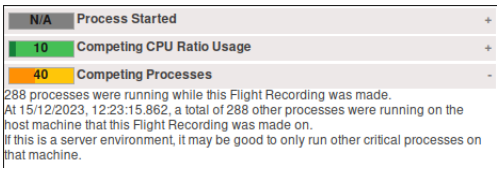

Figura 7: Análisis de procesamiento

Pruebas de rendimiento

Una vez finalizadas las funcionalidades de nuestra aplicación, ha sido necesaria la realización de pruebas de rendimiento para determinar cómo se comporta ante distintas situaciones de carga. Para ello, la herramienta utilizada ha sido **JMeter**, en la que se estableció la siguiente configuración:

En primer lugar creamos un Thread Group donde establecemos los hilos que se van a ejecutar y el número de veces que se va a repetir cada hilo. A continuación establecemos la dirección IP y el puerto a nivel de Thread Group mediante una HTTP Request Defaults.

Además, creamos tres peticiones, que son *Create user* (que permite crear a un número determinado de usuarios), *Create post*, (para la creación de una serie de posts) y *Feed* (que lanza peticiones contra el endpoint de feed de nuestra aplicación).

Todas las solicitudes mencionadas comparten dos archivos de configuración, conocidos como CSV Data Set Config y HTTP Header Manager. El primero ha sido utilizado para establecer una serie de variables a tener en cuenta en las peticiones a partir de un fichero .csv, y el segundo permite añadir una serie de cabeceras, como por ejemplo Content-Type, necesaria para indicar que los datos se envían en formato JSON, y Authorization, necesaria para las peticiones que requieren autenticación del usuario a través de un token. En la solicitud de crear usuarios, a mayores necesitamos un archivo llamado JSON Extractor, que nos permite extraer el token de autenticación mencionado anteriormente y poder usarlo para la creación de los distintos posts.

El equipo utilizado para la obtención de los tiempos presenta las siguientes características:

- **SO**: Manjaro Linux / 64 bit
- **Procesador**: AMD Ryzen™ 7 4700U with Radeon™ Graphics ×8
- **Memoria**: 16GB
- **Disco**: 512 GB SSD

A continuación se expondrán los distintos tiempos de respuesta ante las diferentes situaciones de carga:

Nº de posts, usuarios y consultas	**Tiempo medio de respuesta del Feed (ms)**
100	29.25
1000	9.49
10000	32.49

Cuadro 2: Medias de tiempos de respuesta para diferentes cargas frente al endpoint de feed.

Cabe destacar que, para la realización de estos tests, los usuarios son todos diferentes al igual que los posts. En cuanto a la búsqueda contra el endpoint de feed, se crearon 10 peticiones con distintos filtros que se van repitiendo: se varían las "keywords", se filtra por categoría, se muestran posts expirados, se alterna entre el criterio de ordenación...

En cuanto a los datos obtenidos en las pruebas de rendimiento, se ha observado que los datos no siguen la lógica que inicialmente se esperaba (menos tiempo cuanto menor es el número de peticiones), puesto que tardan menos 1000 peticiones a feed que 100. Esto puede ser debido a que en 100 peticiones, el tamaño de la muestra no fuese lo suficientemente grande para que los datos obtenidos sean significativos, o que los sistemas de optimización internos de la JVM se aprovechen más cuando se pasa un cierto umbral de carga.

Contenerización y despliegue

La última fase de nuestro pipeline de DevOps es la contenerización y el despliegue de nuestra aplicación. En la que, en base a nuestro Dockerfile, se genera una imagen de nuestro sistema aprovechando las ventajas de los contenedores:

```
FROM eclipse-temurin:17-jre

# Copy the packaged jar file into our docker image
COPY maven/*.jar /app.jar

EXPOSE 8080

# Set the startup command to execute the jar
CMD ["java", "-jar", "/app.jar"]
```

La imagen base para este contenedor Docker se obtendrá de una imagen llamada eclipse-temurin con la versión 17-jre.

Esto proporcionará un entorno de ejecución de Java 17 en el contenedor. Después, se copia el .jar generado por maven en la imagen y expone el puerto 8080 (el que emplea nuestra aplicación en modo producción), justo antes de ejecutar la aplicación en nuestro contenedor. Se ha configurado que durante cada build de Jenkins, será creada y pusheada una imagen al container registry de nuestra instancia de GitLab (puerto 5050), justo antes de ser desplegada a Kubernetes. Si ya hubiese una instancia ejecutándose, se anularía antes de desplegar la nueva.

Nuestra aplicación está visible en la url deploy.fic.udc.es/poster.

Trabajo futuro

Como parte de la retrospectiva, se explican aquí las posibles mejoras que nos hubiese gustado mejorar, debido al alcance actual del proyecto, y los límites que se esperan en su evaluación, no se han priorizado:

- Sustituír la estrategia de la gestión de las notificaciones, por una alternativa que encaje mejor con los eventos en vivo, como WebSockets (como sí hemos empleado en las sugerencias de recarga) o SSE (Server-Sent Events).
- Aprovechar el uso de Service Workers, de forma que mejore la experiencia del usuario al aumentar el rendimiento y se reducir la carga e interacción con el servidor.
- Realizar tests end-to-end (e2e) con Selenium, de forma que podamos, durante nuestro proceso de build, probar el sistema en su conjunto de forma automatizada. También se podría haber aumentado la presencia de Mockito a nivel de pruebas unitarias.

Conclusiones

En conclusión, la aplicación del conjunto de las herramientas recogidas en la memoria han facilitado el proceso de desarrollo del proyecto. A pesar de que inicialmente la adaptación al uso de dichas herramientas no ha sido fácil, ya que difieren mucho en su curva de aprendizaje, a medida que se ha aprendido a usarlas, su ayuda ha sido notable, simplificando todo el proceso y ayudando a la gestión del equipo y su trabajo.