TecnoCampus
Escola Superior Politècnica

Centre adscrit a la
upf. Universitat Pompeu Fabra Barcelona

Grado en Ingeniería Informática de Gestión y Sistemas de Información

Estudio de Reactive Processing

Memoria

Andrés Sabater Parietti

TUTOR: Josep Roure Alcobé

Curso 2021-2022

TecnoCampus
Mataró-Maresme

Agradecimientos

A mi familia, mi tutor y a mis amigos.

Abstract

This project carries out a study on reactive programming, and deepen into the world of access to databases. To do this, two applications are created, one using imperative programming, and the other using reactive programming. As support for the study, various load and stress tests are carried out on the applications to study their behavior. These results show a great speed advantage of the imperative solution over the reactive one. But the reactive shows greater consistency and robustness over the imperative, in addition to making better use of system resources.

Resum

Aquest projecte realitza un estudi sobre la programació reactiva, i s'endinsa al món de l'accés a les bases de dades. Per això es creen dues aplicacions, una usant la programació imperativa, i una altra usant la programació reactiva. Com a suport a l'estudi, es fan diverses proves de càrrega i estrès sobre les aplicacions per estudiar-ne el comportament. Aquests resultats mostren un gran avantatge de velocitat de la solució imperativa sobre la reactiva. Però la reactiva mostra més consistència i robustesa sobre la imperativa, a més d'aprofitar millor els recursos del sistema.

Resumen

Este proyecto realiza un estudio sobre la programación reactiva, y se adentra en el mundo del acceso a las bases de datos. Para ello se crean dos aplicaciones, una usando la programación imperativa, y otra usando la programación reactiva. Como soporte al estudio, se realizan diversas pruebas de carga y estrés sobre las aplicaciones para estudiar el comportamiento de estas. Dichos resultados muestran una gran ventaja de velocidad de la solución imperativa sobre la reactiva. Pero la reactiva muestra una mayor consistencia y robustez sobre la imperativa, además de aprovechar mejor los recursos del sistema.

Índice

Índice de figuras

Glosario de términos

API Application Programming Interface.

CRUD Es el acrónimo de "Crear, Leer, Actualizar y Borrar".

Framework Entorno o marco de trabajo a partir del cual se desarrolla un proyecto.

JDK **J**ava **D**evelopment **K**it.

JSON **J**ava**S**cript **O**bject **N**otation. Formato de texto utilizado para el intercambio de estructuras de datos.

SPI **S**ervice **P**rovider **I**nterface. Es un mecanismo de descubrimiento de servicios integrado en JDK.

Warm-Up El calentamiento en un sistema o una aplicación se refiere al hecho de que acaba de ser encendido y no ha realizado demasiados procesos, lo que puede provocar un mayor tiempo de respuesta al realizar dicho proceso.

1. Introducción

El objetivo de este proyecto es realizar un estudio sobre el paradigma del reactive processing, utilizando las herramientas de programación reactiva de Spring, analizando sus características, y comparándolo con la solución tradicional. Además, se busca profundizar en cómo esta tecnología influye en el acceso a Bases de Datos Relacionales utilizando soluciones reactivas e imperativas.

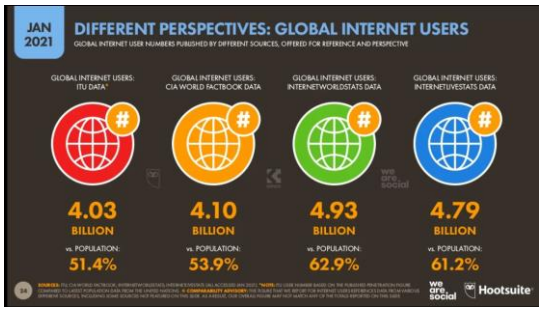
Dicho estudio se corroborará mediante la realización de dos aplicaciones a modo de test. Una usando la solución clásica de Spring, y otra utilizando Spring Reactive. Estas pruebas nos servirán para comparar la dificultad y eficiencia de ambas soluciones.

2. Marco Teórico y análisis de referentes

2.1 Contexto

Hoy en día las empresas cada vez están apostando más por el uso de la arquitectura de micro servicios, debido a su gran escalabilidad, entre muchas otras cosas.

Se puede ver como en los últimos años la cantidad de usuarios que utilizan internet ha aumentado mucho, y seguirá creciendo.



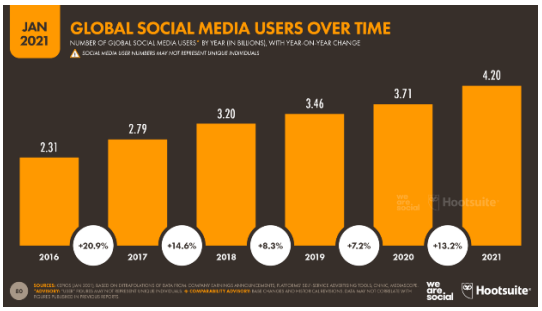


Fig. 2.1.1 y 2.1.2 DIGITAL 2021: GLOBAL OVERVIEW REPORT [1]

Por esto mismo, el paradigma del procesamiento reactivo también está en auge ya que cada vez las aplicaciones tienen más usuarios concurrentes, lo que provoca que se necesiten gestionar más datos sin entorpecer la experiencia del usuario. De esta forma este proyecto está enfocado a disminuir la carga que supone el aumento de los usuarios.

Lo que proporciona principalmente este paradigma es aumentar la eficiencia de las aplicaciones, hacerlas más escalables y reducir al máximo los costes.

Un ejemplo muy claro es el número de suscriptores que ha ido creciendo en la plataforma de Netflix, donde se puede observar no solo la cantidad de usuarios que pueden acceder a la vez, sino la cantidad de información que solicitan. [2]

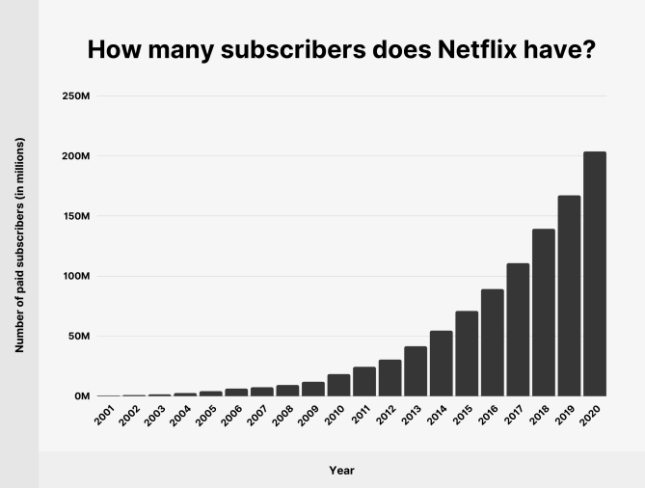


Fig. 2.1.3 Incremento de suscriptores de Netflix (2001/2020). Fuente: https://www.affde.com/

2.2 Antecedentes

Los orígenes de la Programación Reactiva probablemente se remontan a la década de 1970 o incluso antes. Como primer paso en la dirección de la programación reactiva, Microsoft creó la biblioteca Reactive Extensions (Rx) en el ecosistema .NET. Rx fue portado a varios lenguajes y plataformas, incluidos JavaScript, Python, C ++, Swift y Java, por supuesto, llamado RxJava. [3]

Algunas de las empresas que ya utilizan alguna solución reactiva son Netflix, Microsoft, GitHub y Trello. Netflix, uno de los servicios de mayor escala del planeta (con multitud de servicios construidos en Spring), adoptó la programación reactiva hace casi una década. Alibaba (con muchos servicios construidos en Spring) también han adoptado la programación reactiva.

2.3 Información previa

El término "*reactive*" se refiere a los modelos de programación que se crean para reaccionar al cambio: componentes de red que reaccionan a eventos de I/O, controladores de interfaz de usuario que reaccionan a eventos del mouse y otros. En ese sentido, el no bloqueo es reactivo, porque, en lugar de estar bloqueados, ahora está en un modo de reaccionar a las notificaciones a medida que se completan las operaciones o que los datos están disponibles. [4]

Dado que las API de acceso a base de datos son bloqueantes por naturaleza, crear solo la aplicación usando programación reactiva no puede aportar mucho cambio. Por eso, con el

paso del tiempo se han ido desarrollando especificaciones de APIs. Uno de estos casos es R2DBC, la cual es una API reactiva que permite la concurrencia en una aplicación no bloqueante, que a su vez permite usar un numero de *threads* menor, y facilitar la escalabilidad con un menor número de recursos hardware.

[5] Las características del Reactive Manifesto son las siguientes:

- **Responsivos:** El sistema responde a tiempo en la medida de lo posible. La responsividad significa que los problemas pueden ser detectados rápidamente y tratados efectivamente. Los sistemas responsivos se enfocan en proveer tiempos de respuesta rápidos y consistentes, estableciendo límites superiores confiables para así proporcionar una calidad de servicio consistente. Este comportamiento consistente, a su vez, simplifica el tratamiento de errores, aporta seguridad al usuario final y fomenta una mayor interacción.

- **Resiliente:** El sistema permanece responsivo frente a fallos. Cualquier sistema que no sea resiliente dejará de ser responsivo después de un fallo. La resiliencia es alcanzada con replicación, contención, aislamiento y delegación. Los fallos son manejados dentro de cada componente, aislando cada componente de los demás, y asegurando así que cualquier parte del sistema pueda fallar y recuperarse sin comprometer el sistema como un todo.

- **Flexible:** El sistema se mantiene responsivo bajo variaciones en la carga de trabajo. Los Sistemas Reactivos pueden reaccionar a cambios en la frecuencia de peticiones incrementando o reduciendo los recursos asignados para servir dichas peticiones. Esto implica diseños que no tengan puntos de contención o cuellos de botella centralizados, resultando en la capacidad de dividir o replicar componentes y distribuir las peticiones entre ellos.

- **Orientados a mensajes:** Los Sistemas Reactivos confían en el intercambio de mensajes asíncrono para establecer fronteras entre componentes, lo que asegura bajo acoplamiento, aislamiento y transparencia de ubicación. Estas fronteras también proporcionan los medios para delegar fallos como mensajes. El uso del intercambio de mensajes explícito posibilita la gestión de la carga, la elasticidad, y el control de flujo.

3. Objetivos y alcance

3.1 Objetivos

3.1.1 Objetivos del Proyecto

- Realizar un estudio sobre el paradigma de procesamiento reactivo en Spring Reactive.
- Mostrar sus principales características, los beneficios y desventajas, y la diferencia con las soluciones tradicionales.
- Estudiar el acceso a BBDD relacionales con drivers reactivos.

Para comprobar lo estudiado el objetivo es crear una pequeña aplicación utilizando Spring Reactive a modo de test.

3.1.2 Objetivos del Producto

- Estudiar la programación reactiva de forma práctica.
- Comparar la dificultad y la eficiencia de desarrollo entre la versión reactiva y la tradicional.
- Comparar el rendimiento en los accesos a BBDD relacionales.

Se puede pensar como público potencial toda empresa de desarrollo software que tenga altas posibilidades de sufrir incrementos elevados en el número de usuarios concurrentes como puede ser una plataforma de Streaming o las redes sociales.

El proyecto acaba en el momento en el que finaliza el estudio, se realizan las pruebas, y se sacan las conclusiones sobre dichas pruebas.

3.2 Planificación inicial

3.2.1 Lista de actividades

ID	Dependencia	Tareas
A0	-	Realizar Anteproyecto
A1	A0	Estudiar Reactive Processing.
A2	A1	Realizar estudio Reactive Spring.
A3	A2	Estudiar acceso a BBDD relacionales con drivers reactivos.
A4	A3	Crear Modelo de Base de Datos.
A5	A3	Instalar y configurar Software.
A6	A3	Configurar Base de Datos(PostgreSQL).
A7	A5	Instalar dependencias y drivers.
A8	A4	Crear Script de Creación de Base de Datos.
A9	-	Estudiar funcionamiento de JMeter
A10	A8	Desarrollar **primera** funcionalidad de forma imperativa.
A11	A10	Desarrollar **primera** funcionalidad usando *reactive*.
A12	A10; A11	Testear la **primera** funcionalidad (Postman).
A13	A12; A9	Testear **primera** funcionalidad usando JMeter.
A14	A13	Escribir conclusiones de los resultados del **test número uno**.
A15	A1…A14	Realizar y entregar Memoria Intermedia.

A16	A15; A8	Desarrollar **segunda** funcionalidad de forma imperativa.
A17	A16	Desarrollar **segunda** funcionalidad usando *reactive*.
A18	A16; A17	Testear la **segunda** funcionalidad (Postman).
A19	A18; A9	Testear **segunda** funcionalidad usando JMeter.
A20	A19	Escribir conclusiones de los resultados del **test número dos**.
A21	A20; A8	Desarrollar **tercera** funcionalidad de forma imperativa.
A22	A21	Desarrollar **tercera** funcionalidad usando *reactive*.
A23	A21; A22	Testear la **tercera** funcionalidad (Postman).
A24	A23; A9	Testear **tercera** funcionalidad usando JMeter.
A25	A24	Escribir conclusiones de los resultados del **test número tres**.
A26	A25	Testear toda la aplicación en conjunto (Postman).
A27	A14-A20-A25	Juntar todas las conclusiones de los test realizados.
A28	A27	Realizar comparativa entre reactive y la forma tradicional.
A29	A1…A28	Realizar Estudio de Viabilidad.
A30	A1…A28	Escribir la memoria del proyecto.

Tabla 1. Lista de Actividades del proyecto

3.2.2 Diagrama de Gantt

Para calcular los tiempos de ejecución y poder ver los posibles puntos críticos se realiza un Diagrama de Gantt, el cual nos ayuda a situar las tareas en el tiempo y ver cuáles son las más importantes y de las que depende más el proyecto.

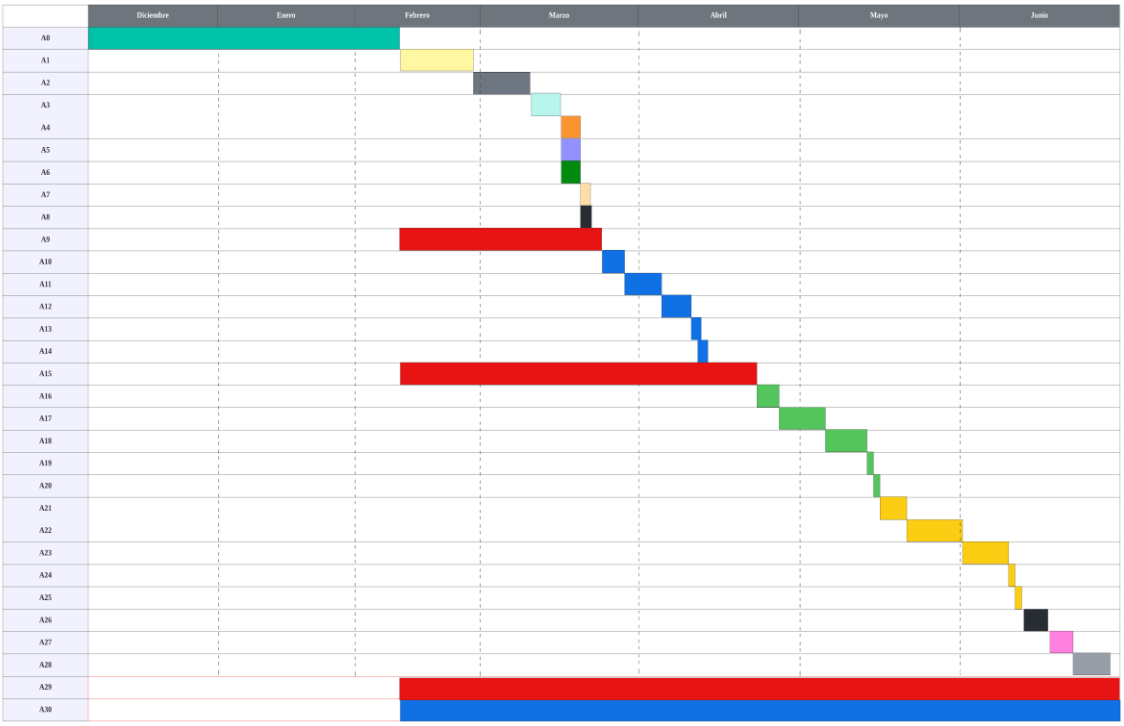


Fig. 3.2.2 Diagrama de Gantt

3.2.3 Actividades críticas

De este diagrama y junto a la lista de actividades se puede deducir que el camino critico pasa por las tareas iniciales **A1, A2, A3** (referentes al estudio), pero también en las actividades de desarrollo de funcionalidades, **A10…A14**; **A16…A20**; **A21…A25.**

Esta estimación nos muestra que realmente hay poco margen de error, por lo que debe trabajarse para cumplir los plazos acordados. Por suerte, gracias a la metodología que se ha escogido para realizar el proyecto, nos va a ayuda a cumplir con los plazos en la medida de lo posible.

4. Metodología

4.1 Búsqueda de Información

Para este proyecto se va a optar por buscar en libros y artículos de gente reconocida en el ámbito de la programación reactiva o en Spring, además de documentaciones oficiales. La información se obtiene principalmente buscando en Google Books o Google Académico, y en caso de al leer un documento no encontrar lo esperado, buscando en libros/artículos relacionados o en Google de forma normal.

Las **palabras claves** son:

Spring Reactive, Reactive Processing, Reactive Programming, R2DBC, Bases de Datos, Microservicios, Spring.

En caso de que alguna información sea muy ambigua o poco clara, se descarta o se tiene como base para nuevas búsquedas. Por lo contrario, para aceptar el uso de determinada información se busca la misma información en diferentes fuentes para comprobar la validez y la calidad de la información.

4.2 Documentación y Control de Versiones

En cuanto a la gestión de la documentación y al control de versiones, se realizan copias de Seguridad de todos los documentos del proyecto en Drive cada día que se modifiquen, para intentar minimizar las pérdidas posibles si el ordenador se estropease.

También se va a hacer un control de versiones utilizando Github y Github Desktop tanto para la documentación del Anteproyecto, Memoria y estudio de viabilidad, como para el software de prueba desarrollado, teniendo una rama diferente para cada funcionalidad, que después se van a unir en la rama principal a medida que se finalicen.

4.3 Desarrollo de Software

Fig. 4.3 Ventajas y desventajas metodología Scrum. Fuente: https://blog.wearedrew.co [6]

En cuanto al método para desarrollar la solución de software, se utiliza la metodología Agile de **SCRUM**, ya que este tipo de metodología se basa en la rapidez y adaptabilidad, y trabaja de forma cíclica.

4.3.1 Perfiles SCRUM

Ya que el equipo es de una única persona, esta misma representa a los diferentes perfiles que contempla, como son:

- **Product Owner**: Representa al cliente, y es quien define y prioriza los objetivos del proyecto.

- **Equipo de desarrollo:** Es el equipo que se encarga de desarrollar el producto y entregarlo.

- **Stakeholder:** Es el conjunto de personas interesadas en el proyecto (Directores, dueños, comerciales…).

Faltaría un perfil, que es el de **Scrum Master**, que puede atribuirse al tutor del proyecto ya que es el que se encarga de guiar y ayudar al equipo de desarrollo.

4.3.2 Etapas SCRUM

SCRM cuenta con 4 etapas principales.

- **Planificación de Sprint**: Un sprint es un mini-proyecto con un objetivo concreto. En la primera reunión del equipo se definen aspectos como la funcionalidad, objetivos, riesgos del *sprint*, plazos de entrega, entre otros.

- **Etapa de Desarrollo**: Es donde el equipo se encarga de realizar las tareas asignadas a ese sprint.

- **Revisión del sprint**: Se analiza el trabajo realizado durante el sprint, tanto los resultados como problemas surgidos. En esta etapa participan los equipos, supervisores, jefes y dueños de productos.

- **Retroalimentación:** Los resultados se entregan para recibir un *feedback* no solo por parte de los profesionales dentro del proyecto, sino también de las personas que utilizan directamente el producto, los clientes potenciales.

En la etapa de Desarrollo se va a utilizar Waterfall, una práctica de desarrollo de software que consiste en primero establecer los requisitos, desarrollar y testear, sucesivamente se van desarrollando las funcionalidades.

En el caso de este proyecto los test y las funcionalidades están sobretodo orientadas a analizar y comparar el funcionamiento de la programación reactiva, pero encaminado al acceso a Bases de Datos.

5. Definición Requerimientos

5.1 Requerimientos tecnológicos

BBDD: PostgreSQL.

Back-End: Java.

IDE: IntelliJ.

Frameworks: Spring MVC, Project Reactor, Spring WebFlux

DB Drivers: JDBC, R2DBC (para Postgres)

Testing externo: ~~LoadView~~ JMeter

Test para el desarrollo: Postman

Monitorización de recursos del PC: VisualVM para IntelliJ

Previamente se había elegido LoadView ya que era de las mejores aplicaciones que había para hacer testing. Pero al no ser gratuito y tener un coste por cada test realizado se ha optado por utilizar JMeter, el cual es Open-Source. Además, al ser unos test muy básicos no se necesita una herramienta tan completa.

5.2 Requerimientos funcionales

- El usuario debe poder interactuar con el servicio a través de alguna herramienta de testing de API Rest (Ej. Postman).
- Primera funcionalidad:
    - El usuario debe poder registrarse en un formulario web.
    - El usuario debe poder identificarse.
    - El usuario debe poder modificar sus datos.
    - El usuario debe poder eliminar su perfil.
- Segunda funcionalidad (Estando el usuario identificado):
    - El usuario debe tener una lista donde agregar series o películas manualmente.
    - El usuario debe poder añadir una serie o una película.
    - El usuario debe poder modificar una película o serie.
    - El usuario debe poder eliminar una película o serie.
    - El usuario debe poder visualizar su lista de series y películas.

- Tercera funcionalidad:
    - El usuario podrá crear una valoración para una serie o película.
    - El usuario podrá modificar una valoración para una serie o película.
    - El usuario podrá visualizar la valoración de una serie o película.
    - El usuario podrá eliminar una valoración para una serie o película.
- La aplicación debe guardar los datos en la base de datos.
- La aplicación debe poder obtener los datos de la base de datos.

5.3 Modelo de Base de Datos

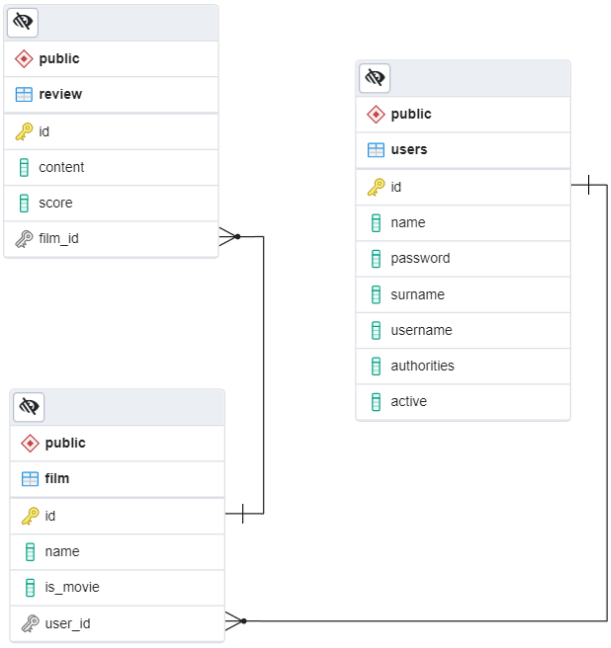


Fig. 5.3 Modelo de Base de Datos.

Este es el modelo de base de datos utilizado en este proyecto. Como se puede observar, hay tres tablas principales que se mencionan en el anterior punto. La tabla “User” guarda los usuarios; “Film” es una serie o pelicula (“isMovie” dice el tipo), y un usuario tiene una lista de peliculas o series; Finalmente “Review” guarda la valoracion del usuario sobre una pelicula, el cual es un texto, y una puntuacion opcional. La clave foranea de review hacia film tiene una restricción de único de forma que una film solo pueda tener una review por cada usuario.

Se usan dos bases de datos que siguen este mismo modelo, una para la aplicación reactiva y otra para la aplicación imperativa, a diferencia que en esta hay dos tablas mas para la relacion roles-usuario.

6. Estudio sobre Reactive Spring

Para realizar el proyecto se deben entender también los principios básicos y el funcionamiento de la programación reactiva. También es importante conocer el contexto donde se aplicará, así como los elementos relacionados con dicho paradigma.

6.1 Programación Reactiva

Como ya se ha comentado en el punto 2, según la documentación de Spring sobre Spring Web [4], el término "reactive" se refiere a los modelos de programación que se crean para reaccionar al cambio: componentes de red que reaccionan a eventos de I/O, controladores de interfaz de usuario que reaccionan a eventos del mouse y otros.

Por otro lado, en su libro “Reactive Spring” Josh Long la define como un enfoque para escribir software que abarca la entrada y salida (IO) asíncronas. [7]

Junto a estas definiciones encontramos algunos términos que se deberán explicar un más ampliamente, como los *Reactive Streams, Backpressure o Event-Driven.* Pero las características más importantes que tiene este paradigma es ser **Asíncrono** y **No-Bloqueante,** referente a las entradas y salidas de datos de un sistema.

En este proyecto se va a trabajar con la librería Reactor que es la base de los *frameworks* reactivos de Spring, y se utilizara el *framework* de Spring WebFlux. La programación reactiva también se puede aplicar al front-end, pero este caso se va a centrar en back-end y el acceso a base de datos.

6.1.1 *Backpressure*

En el mundo del software, la "contrapresión" es una analogía tomada de la dinámica de fluidos. En el contexto del software, la definición podría modificarse para referirse al flujo de datos dentro del software:

“Resistencia o fuerza que se opone al flujo de datos deseado a través del software.”

Como se ha comentado en el anterior punto, la programación reactiva está pensada para trabajar con entradas y salidas asíncronas. Es decir, tomar datos de entrada y convertirlos en datos de salida deseados. Esos datos de salida pueden ser JSON de una API, pueden ser HTML para una página web o los píxeles que se muestran en su monitor.

[8] La contrapresión se da cuando el progreso de convertir esa entrada en salida se resiste de alguna manera. En la mayoría de los casos, esa resistencia es la velocidad computacional (problemas para calcular la salida tan rápido como entra la entrada), por lo que esa es, con mucho, la forma más fácil de verlo. Pero también pueden ocurrir otras formas de contrapresión: por ejemplo, si su software tiene que esperar a que el usuario realice alguna acción.

Es posible escuchar a alguien usar la palabra "contrapresión" para dar a entender que algo tiene la capacidad de controlar o manejar la contrapresión. [8]

6.1.2 Reactive Streams

La programación reactiva se basa en el patrón de diseño Observer, donde se tiene un Publisher y uno o más Suscribers que reciben notificaciones cuando el Publisher emite nuevos datos.

Reactive Streams es una pequeña especificación que define la interacción entre componentes asincrónicos con contrapresión. [9] El objetivo principal de Reactive Streams es permitir que el suscriptor controle la rapidez o la lentitud con la que el editor produce datos.

[9] Por ejemplo, un repositorio de datos (que actúa como publicador) puede generar datos que un servidor HTTP (que actúa como suscriptor) puede escribir en la respuesta.

Los Reactive Streams tienen cuatro componentes principales:

- El **Publisher<T>** es un productor que emiten el flujo de datos de tipo T. Un **Publisher** produce valores de tipo T para un Subscriber<T>. Un publicador puede servir a múltiples suscriptores suscritos dinámicamente en varios momentos.

```
package org.reactivestreams;

public interface Publisher<T> {

    void subscribe(Subscriber<? super T> s);
}
```

Fig 6.1.2.1 Publisher Interface. Fuente: *Reactive Spring*, Josh Long

- Un **Subscriber<T>** representa a la consumidora de datos, que se suscribe a Publisher<T>. Son a los que se les notifican los cambios en el flujo de datos que emite el Publisher, y lo hace a través del método onNext(T).

```
package org.reactivestreams;

public interface Subscriber<T> {

    public void onSubscribe(Subscription s);

    public void onNext(T t);

    public void onError(Throwable t);

    public void onComplete();
}
```

Fig 6.1.2.2 Subscriber Interface. Fuente: *Reactive Spring*, Josh Long

Si hay algún error, se llama a su método onError(Throwable t). Cuando el procesamiento se ha completado normalmente, se llama al método onComplete del suscriptor.

- Cuando un Suscriptor y Publicador se conectan, se le otorga una suscripción en el método Subscriber onSubscribe(). La suscripción es la parte más importante de toda la especificación ya que permite la contrapresión. El Suscriptor utiliza el método de Subscription request() para solicitar más datos o el método Subscription cancel() para detener el procesamiento.

```
package org.reactivestreams;

public interface Subscription {

    public void request(long n);

    public void cancel();
}
```

Fig 6.1.2.3 Interfaz Suscripción. Fuente: *Reactive Spring*, Josh Long

- Un procesador<T, R> que simplemente amplía tanto el suscriptor<T> como el publicador<R>. Además, produce y consume los datos.

```
package org.reactivestreams;

public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
}
```

Fig 6.1.2.4 Interfaz Procesador. Fuente: *Reactive Spring*, Josh Long

Aquí se puede ver un como es el proceso de conexión entre el Publisher y el Subscriber:

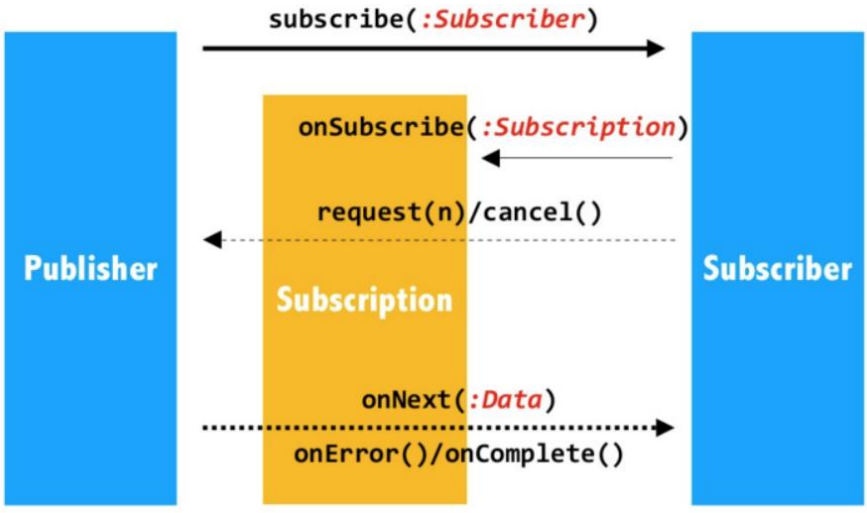


Fig 6.1.3.1 Conexión Publisher-Suscriber. [10]

6.1.3 Project Reactor

Aun así, solo con los reactive streams no son suficientes, y se necesitaría una implementación superior que soporte operaciones de filtrado o transformación. Project Reactor es la solución, ya que se basa en los reactive streams y sirve para dar soporte a las tareas de procesamiento.

Reactor proporciona dos especializaciones de Publisher<T>.

El primero, **Flux<T>**, es un publicador que produce cero o más valores ilimitados.

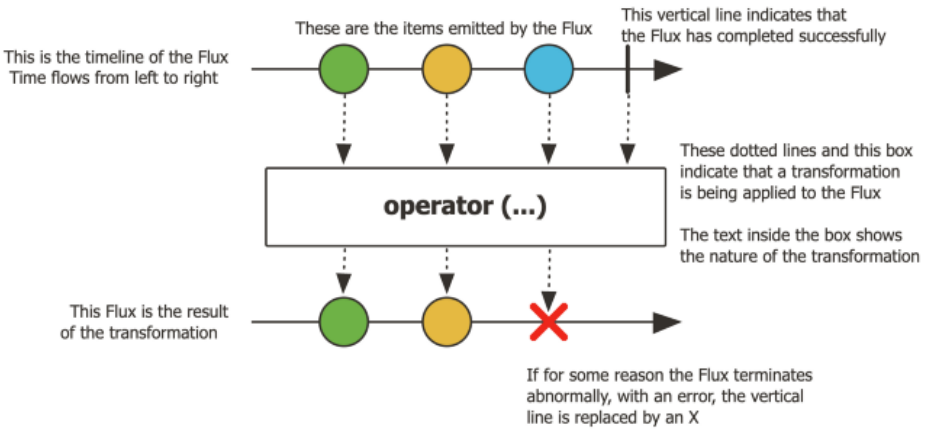


Fig 6.1.3.2 Flujo de Datos con Flux. [10]

El segundo, **Mono<T>**, es un Publisher que emite cero o un valor.

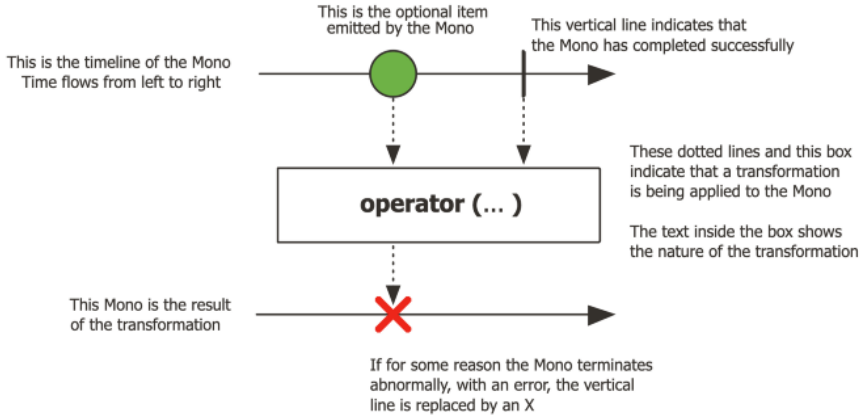


Fig 6.1.3.3 Flujo de Datos con Mono. [10]

6.1.4 Programación Síncrona vs Asíncrona

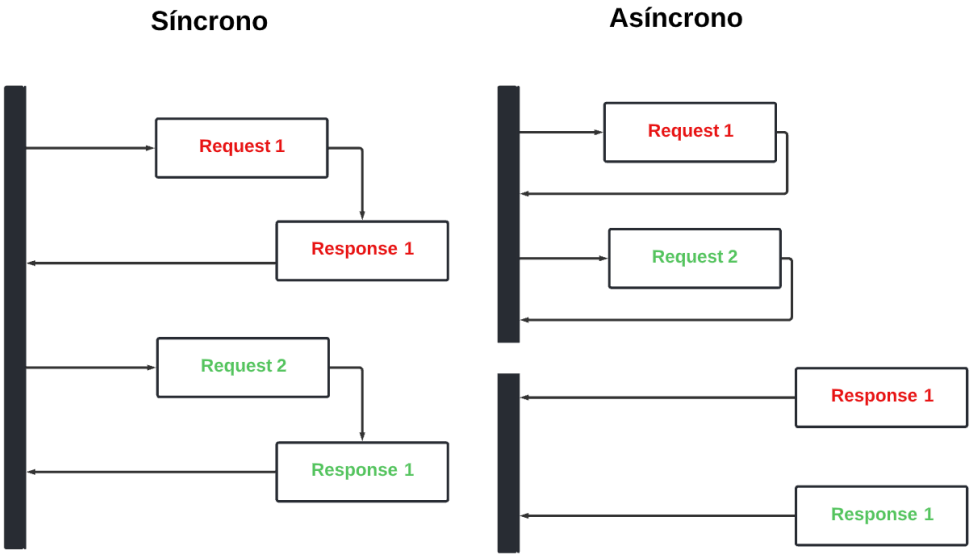


Fig 6.1.4 Proceso Síncrono vs Asíncrono

6.1.4.1 Síncrono Bloqueante

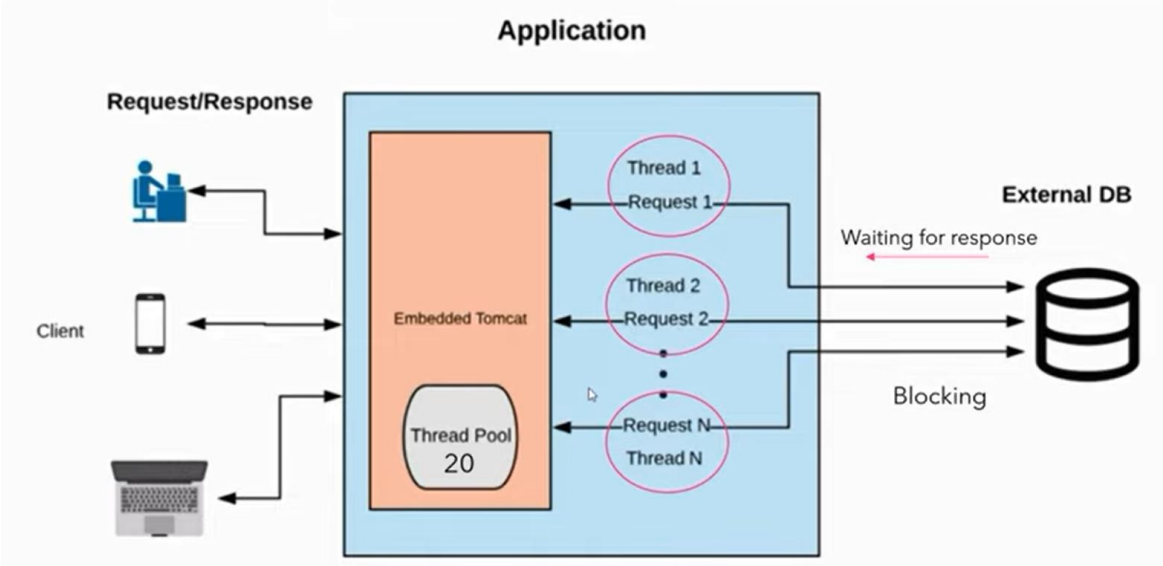


Fig 6.1.4.1 Flujo Síncrono [11]

La programación síncrona es el tipo de programación que más común; Como se ve en la imagen, llega una petición, crea un *thread* que conecta con la base de datos, y espera a la respuesta. Tan pronto como se obtiene una respuesta, pasa a la siguiente petición y espera su respuesta, así sucesivamente. La aplicación hace de Subscriber y la DB de Publisher.

En un sistema síncrono, la peticion bloquea el *thread* hasta que todos los elementos seleccionados se recuperan de la base de datos y se devuelven al consumidor. Mientras se realiza la consulta, el subproceso se bloquea y esto provoca una pérdida de recursos. Eventualmente, cuando los datos se devuelven al consumidor, el subproceso se vuelve a colocar en el grupo y queda disponible para manejar otra solicitud.

6.1.4.2 Asíncrono No Bloqueante

La programación asincrónica es un medio de programación en paralelo. Una unidad de trabajo se ejecuta por separado del subproceso de la aplicación principal y notifica al subproceso de llamada de su finalización, falla o progreso.

En un sistema asíncrono, la lectura de datos de la base de datos no bloquea el *thread*. Cada vez que se obtiene un registro, se publica un evento. Cualquier subproceso puede manejar el evento y enviar el registro al consumidor sin tener que esperar a que se obtengan los otros registros.

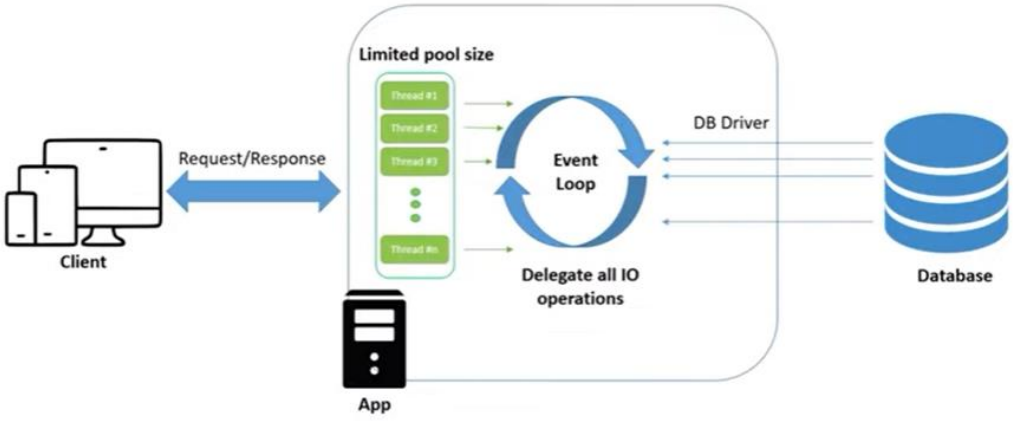


Fig 6.1.4.2 Flujo Asíncrono [11]

6.1.4.3 Ejemplo con Analogia

Ahora, mediante una analogía se pretende ejemplificar estos conceptos:

El contexto se sitúa en un restaurante, en el cual se distinguen, clientes, camareros y la cocina. Los clientes son las peticiones que llegan a una aplicación, los camareros los diferentes *threads* (Suscribers), y la cocina seria la Base de Datos, por ejemplo, (Publisher).

En un sistema síncrono hay que imaginar que le llega un pedido a cada camarero, estos proceden a entregárselo al cocinero. El primer camarero entrega la orden al cocinero, entonces esperara hasta que termine de cocinar y le entregue el pedido. Mientras tanto los otros camareros tendrán que esperar también hasta que finalice el primer pedido. Una vez finalizado el pedido, el siguiente camarero procederá a entregarle la orden al cocinero, y así sucesivamente.

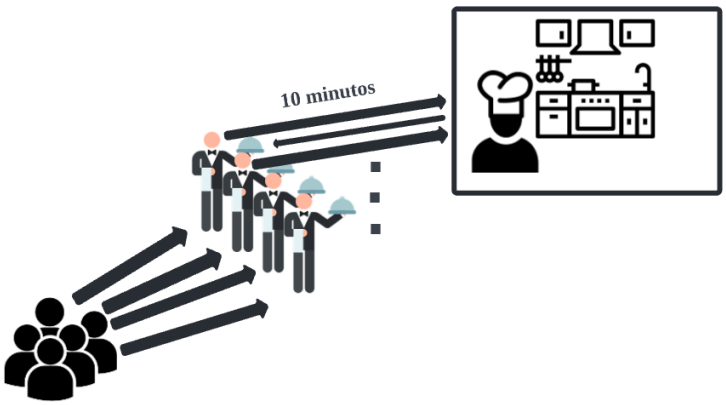


Fig 6.1.4.3.1 Analogía flujo Síncrono

Mientras tanto en un sistema asíncrono los camareros entregan la orden en un “tablón de órdenes” del cual el cocinero ira cogiendo, entonces el camarero quedaría libre para recibir más peticiones de los clientes. Cuando el cocinero termina una orden, lo notifica a los camareros y alguno se hace cargo de entregárselo al cliente. De esta forma los camareros no quedarían esperando a que finalice el primer pedido.

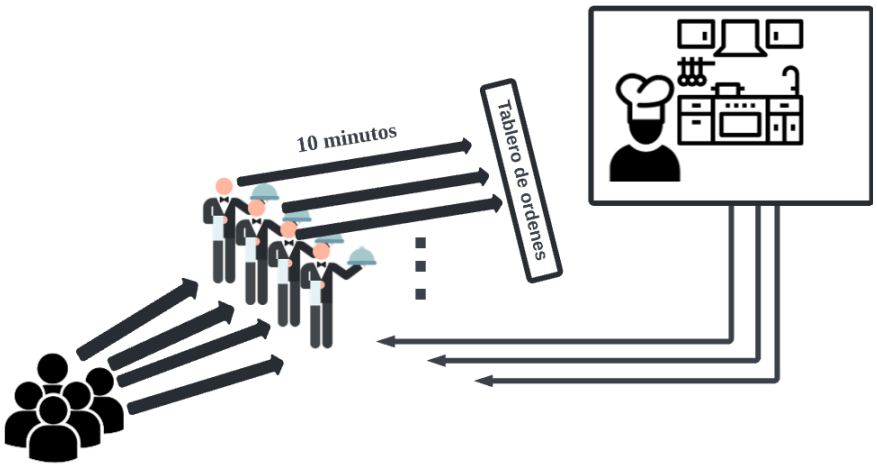


Fig 6.1.4.3.2 Analogía flujo Asíncrono

6.2 R2DBC

R2DBC es una especificación diseñada desde cero para la programación reactiva con bases de datos SQL.

· **Características**:

- Se basa en la especificación de Reactive Streams, que proporciona una API sin bloqueo totalmente reactiva.
- Trabaja con bases de datos relacionales.
- Admite soluciones escalables. Con Reactive Streams, R2DBC le permite pasar del modelo clásico de "un subproceso por conexión" a un enfoque más potente y escalable.
- Admite aplicaciones nativas en la nube que utilizan bases de datos relacionales como PostgreSQL, MySQL y otras.

En Spring existe Spring Data R2DBC, que es familia de Spring Data, que facilita la implementación de repositorios basados en R2DBC. Aplica abstracciones de la familia de Spring y soporte de repositorio para R2DBC. Facilita la creación de aplicaciones basadas en Spring que utilizan tecnologías de acceso a datos relacionales en una *stack* de aplicaciones reactivas.

No ofrece almacenamiento en caché, carga diferida, escritura detrás o muchas otras características de los marcos ORM. Esto hace que Spring Data R2DBC sea un mapeador de objetos simple, limitado y con opiniones.

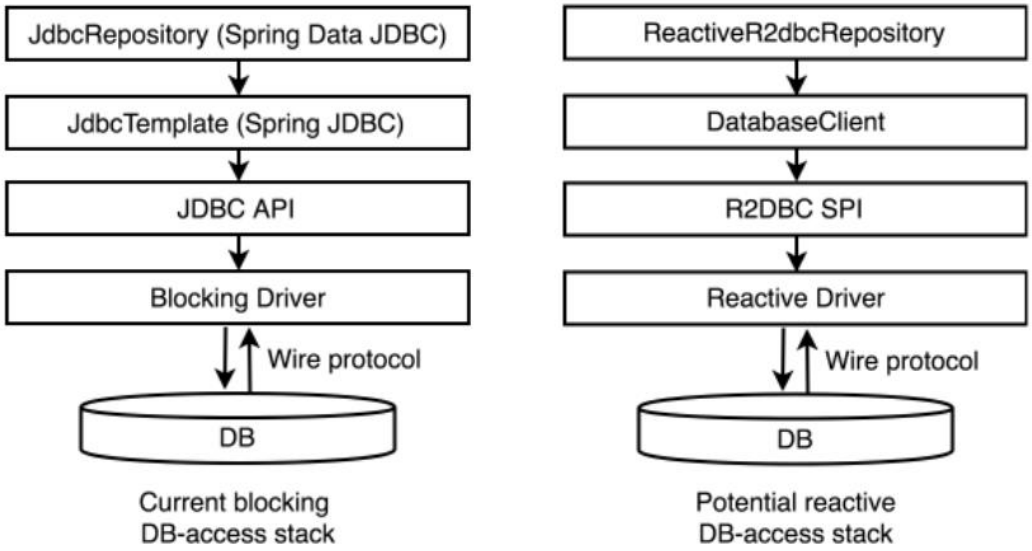


Fig. 6.2 App Reactiva con JDBC vs R2DBC. [12]

Si se tiene una aplicación programada de forma reactiva y se quiere conectar a una base de datos a través de drivers bloqueantes como JDBC, entonces no tiene mucho sentido usar programación reactiva, ya que por mucho que se hagan llamadas asíncronas en la aplicación, este driver bloqueara el *thread* cuando intentan acceder a la base de datos, lo cual no es una solución completamente reactiva.

Gracias al driver R2DBC, eliminar esta limitación es posible, ya que permite acceder a la base de datos sin bloquear los *threads*.

6.2.1 Service-ProviderInterface (SPI)

Estas son las interfaces que nos proporciona la API, en negrita las que son más importantes.

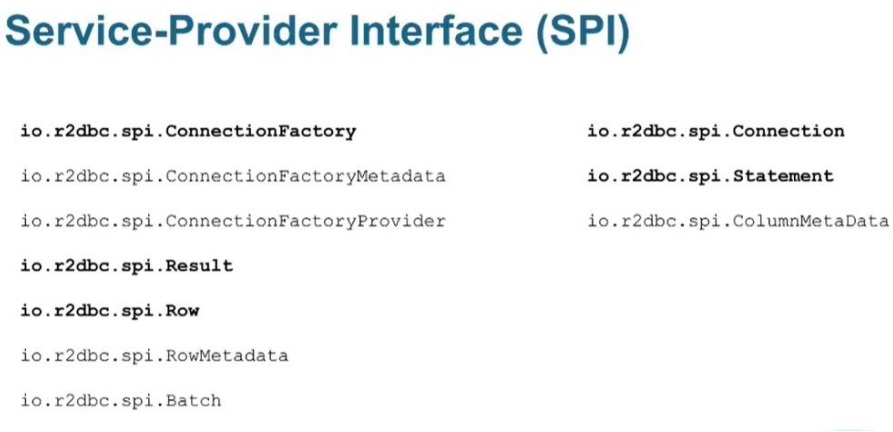


Fig. 6.2.1.1 Interfaces SPI R2DBC. [13]

Esta es la interfaz que se encarga de realizar la conexión a la base de datos:

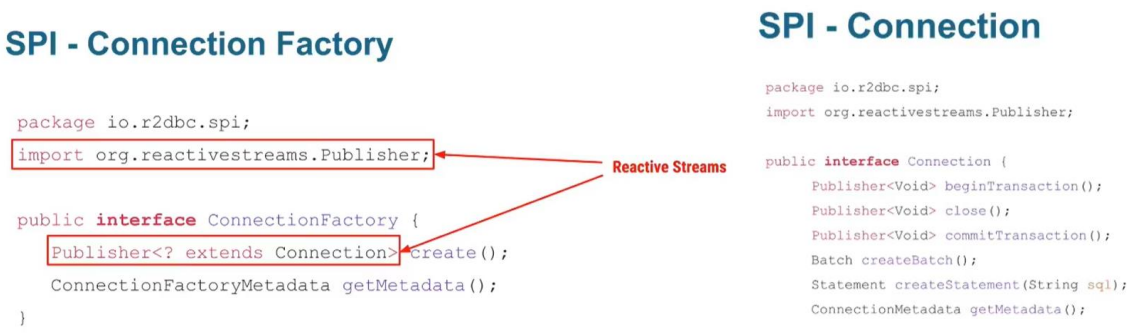


Fig. 6.2.1.2 Interfaz Connection Factory y Connection R2DBC. [13]

Hay que darse cuenta que el método créate no devuelve la conexión a la base de datos ya que si devolviese la conexión directamente entonces estaría bloqueando hasta obtener dicha conexión. De esta forma devuelve un **Publisher**, de Reactive Streams al cual hay que suscribirse para realizar las operaciones con la conexión.

De igual forma los métodos de Connection que interactúan con la base de datos también devuelven Publishers.

Fig. 6.2.1.2 Interfaz Statement y Result R2DBC. [13]

Siguiendo con las anteriores interfaces al hacer consultas su ejecución devuelve un Publisher con un resultado, que este a su vez obtiene las filas de la base de datos a través de otro Publisher, que se pueden mapear al tipo que se quiera.

Más adelante se muestra la forma en que se configura la conexión a base de datos en Spring Boot.

7. Aplicaciones del Proyecto

La utilidad principal que se quiere crear para la aplicación de este proyecto es para testear una aplicación reactiva que accede a base de datos usando un driver reactivo y compararlo a la solución tradicional con drivers bloqueantes.

Se ha creado una API Rest que permite las operaciones básicas de CRUD para la clase Usuario, es decir poder crear un usuario, retornarlo, modificarlo y eliminarlo.

Por otra parte, se ha añadido seguridad a dicha aplicación, de forma que cualquiera puede crear un usuario, siempre que no tenga el mismo nombre de usuario. Por otro lado, el resto de operaciones de *get*, *update* y *remove* solo se podrán hacer sobre el usuario que este autenticado. Para esto se ha añadido el módulo de Spring Security y se ha agregado JWT para la seguridad de las comunicaciones cliente-servidor. Esto se ha realizado para tener un escenario más realista de lo que sería el acceso a una aplicación. Esta seguridad se ha añadido usando la variante reactiva de Spring Security.

Se ha seguido el patrón por capas como la arquitectura de este proyecto, con esta estructura:

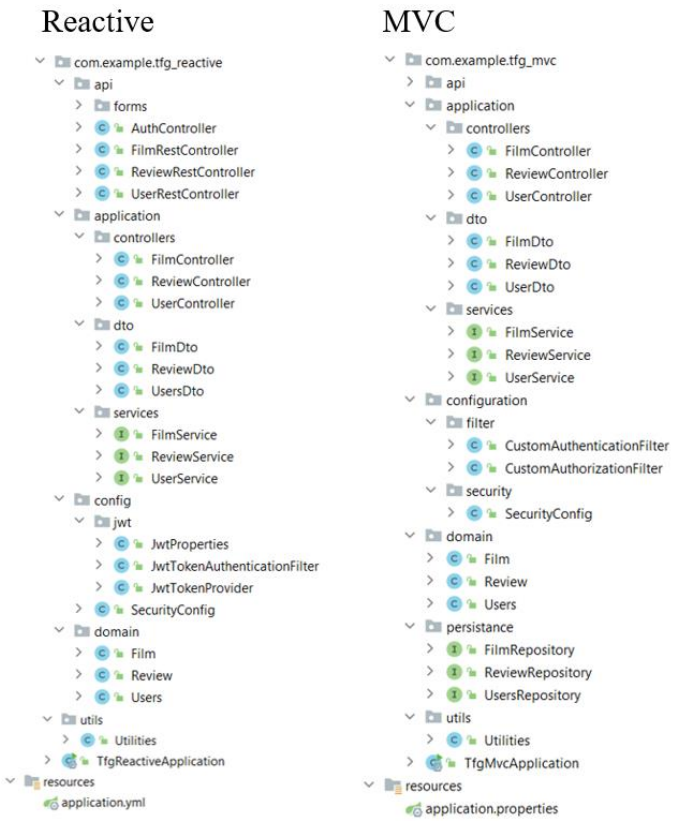


Fig. 7.1 Arquitectura por capas y clases del proyecto.

Las funcionalidades se testean utilizando Postman para validar que las APIs funcionan correctamente, para comprobar que al hacer los test de carga y estrés funcionen correctamente.

Para ambas aplicaciones se utilizan los servlets por defecto. En el caso de la solución reactiva al usar Project Reactor utiliza Netty, el cual es un servlet que soporta el funcionamiento no bloqueante. Por otro lado, la solución tradicional utiliza Tomcat.

Además, para configurar la base de datos en la solución tradicional se ha realizado usando el auto-configurador de Spring Boot y JDBC en el archivo **properties**.

```
spring.datasource.url=jdbc:postgresql://localhost:5432/JDBC
spring.datasource.username=postgres
spring.datasource.password=[redacted]
```

Fig. 7.2 Configuración JDBC

Por otro lado, para configurar la base de datos en la solución tradicional se ha realizado usando el auto-configurador de Spring Boot y r2dbc en el archivo **yaml**.

```
spring:
  r2dbc:
    url: r2dbc:postgresql://localhost:5432/ReactiveDatabase
    username: postgres
    password:[redacted]
```

Fig. 7.3 Configuración R2DBC

7.1 Repositorios

En este proyecto se contiene tres repositorios, correspondientes a las tablas vistas en base de datos y a los requerimientos, siendo repositorios para *Users*, *Films* y *Reviews*. En cada repositorio se utiliza la anotación @Query para marcar que consulta quiere que realice cada función, ya que son consultas personalizadas. Esto se debe a que en r2dbc no hay mapeador de relaciones por lo tanto hay que realizar las consultas manuales para que no de error a causa de que la clase contiene una clase que no está contemplada en la tabla de base de datos.

7.1.1 Repositorios de *Users*

De esta forma se configura el Repositorio de *users* para la aplicación imperativa:

```
@Repository
public interface UsersRepository extends JpaRepository<Users, Long> {
    Users findByUsername(String username);
}
```

Fig. 7.1.1 Repositorio *Users* con JPA

Y esta es la forma de configurar el Repositorio de *users* para la aplicación reactiva:

```
@Repository
public interface UserRepository extends R2dbcRepository<Users, Long> {
        Mono<Users> findByUsername(String username);
}
```

Fig. 7.1.1.2 Repositorio *Users* con R2DBC

7.1.2 Repositorios de *Films*

De esta forma se configura el Repositorio de *films* (películas) para la aplicación imperativa:

```
@Repository
public interface FilmRepository extends JpaRepository<Film, Long> {
    List<Film> findByUserID(Long id);
}
```

Fig. 7.1.2 Repositorio *Films* con JPA

Y esta es la forma de configurar el Repositorio de *films* (películas) para la aplicación reactiva:

```
@Repository
public interface FilmRepository  extends R2dbcRepository<Film, Long>{
    Flux<Film> findByUserID(Long user_id);

    @Query("INSERT INTO film (name, is_movie, user_id) VALUES ($1, $2, $3)")
    Mono<Void> saveFilm(String name, boolean isMovie, Long user_id);

    @Query("UPDATE film set name = $1, is_movie = $2 where id=$3")
    Mono<Film> updateFilm(String name, boolean isMovie, Long id);
}
```

Fig. 7.1.2.2 Repositorio *Films* con R2DBC

7.1.3 Repositorios de *Reviews*

De esta forma se configura el Repositorio de *reviews* para la aplicación imperativa:

```
@Repository
public interface ReviewRepository extends JpaRepository<Review, Long> {
    Review findByFilmID(Long id);

    @Transactional
    @Modifying
    @Query("UPDATE Review set content = ?1, score = ?2 where id=?3")
    void updateReview(String content, int score, Long id);

}
```

Fig. 7.1.3 Repositorio *Reviews* con JPA

Y así se configura el Repositorio de *reviews* para la aplicación reactiva:

```
@Repository
public interface ReviewRepository extends R2dbcRepository<Review, Long> {

    Mono<Review> findByFilmID(Long id);

    @Query("UPDATE review set content = $1, score = $2 where id=$3")
    Mono<Void> updateReview(String content, int score, Long id);
}
```

Fig. 7.1.3.2 Repositorio *Reviews* con R2DBC

Como se puede ver, las operaciones de r2dbc devuelven objetos Mono o Flux ya que las operaciones se empaquetan en un flujo reactivo al igual que Spring WebFlux. De esta forma los métodos realizan las llamadas a los métodos del SPI como, por ejemplo, al llamar a la anotación de @Query realiza la ejecución que devuelve un Publisher.

7.2 Rest APIs

Se ha construido la API Rest para las diferentes funcionalidades, y a partir de aquí solo se muestra el desarrollo de la aplicación reactiva. Se puede ver como igual que el repositorio devuelven objetos Mono y Flux, y además los objetos que reciben como parámetros también lo son ya que se quiere que las operaciones sobre estos objetos no bloqueen y se suscriban a algún Publisher para que sea 100% reactivo. Se recibe el usuario autenticado para saber sobre que usuario hacer las operaciones del CRUD (Excepto crear un usuario).

7.2.1 Rest API de *Users* - Reactive

Para la aplicación imperativa será igual, pero devolviendo una ResponseEntity con el objeto.

```java
@RestController
@CrossOrigin(origins = "*")
@RequiredArgsConstructor
@RequestMapping("/api/users")
public class UserRestController {

    private final UserController userController;

    @ResponseStatus(HttpStatus.OK)
    @GetMapping("/me")
    public Mono<UsersDto> getMe(@AuthenticationPrincipal UserDetails principal){
        return userController.getUserDto(principal); }

    @ResponseStatus(HttpStatus.OK)
    @GetMapping("/all")
    public Flux<Users> getUsers(){ return userController.getUsers(); }

    @ResponseStatus(HttpStatus.NO_CONTENT)
    @DeleteMapping("/delete/me")
    public Mono<Void> deleteMe(@AuthenticationPrincipal UserDetails principal){ return userController.deleteUser(principal); }

    @ResponseStatus(HttpStatus.OK)
    @PutMapping("/update/me")
    public Mono<UsersDto> updateMe(@AuthenticationPrincipal UserDetails principal, @RequestBody Users users){ return userController.updateMe(principal, users); }

    @ResponseStatus(HttpStatus.CREATED)
    @PostMapping("/save")
    public Mono<Void> saveUser(@RequestBody Mono<Users> user) { return userController.saveUser(user); }
}
```

Fig. 7.2.1 Rest API *Users*

7.2.2 Rest API de *Films* - Reactive

Esta es la Rest API creada para la funcionalidad relacionada con los *films*:

```java
@RestController
@RequestMapping("/api/films")
@CrossOrigin(origins = "*")
@RequiredArgsConstructor
public class FilmRestController {

    private final FilmController filmController;

    @ResponseStatus(HttpStatus.OK)
    @GetMapping("/me")
    public Flux<Film> getMyFilmsAll(@AuthenticationPrincipal UserDetails principal){ return filmController.getMyFilms(principal); }

    @ResponseStatus(HttpStatus.OK)
    @GetMapping("/me/random")
    public Mono<FilmDto> getMyRandomFilm(@AuthenticationPrincipal UserDetails principal){ return filmController.getMyFirstFilm(principal); }

    @ResponseStatus(HttpStatus.NO_CONTENT)
    @DeleteMapping("/delete/random")
    public Mono<Void> deleteRandomFilm(@AuthenticationPrincipal UserDetails principal){ return filmController.deleteRandomFilm(principal); }

    @ResponseStatus(HttpStatus.OK)
    @PutMapping("/update/me/random")
    public Mono<FilmDto> updateRandomFilm(@AuthenticationPrincipal UserDetails principal, @RequestBody Film film){ return filmController.updateRandomFilmMe(principal, film); }

    @ResponseStatus(HttpStatus.CREATED)
    @PostMapping("/save")
    public Mono<Void> saveFilm(@AuthenticationPrincipal UserDetails principal, @RequestBody Film film){ return filmController.saveFilm(principal, film); }

}
```

Fig. 7.2.2 Rest API *Films*

7.2.3 Rest API de *Reviews* - Reactive

Esta es la Rest API creada para la funcionalidad relacionada con las *reviews*:

```java
@RestController
@CrossOrigin(origins = "*")
@RequiredArgsConstructor
@RequestMapping("/api/reviews")
public class ReviewRestController {

    private final ReviewController reviewController;

    @ResponseStatus(HttpStatus.OK)
    @GetMapping("/me")
    public Flux<Film> getMeWithFilmsAndReviews(@AuthenticationPrincipal UserDetails principal){ return reviewController.getMyUserWithFilmsAndReview(principal); }

    @ResponseStatus(HttpStatus.OK)
    @GetMapping("/me/random")
    public Mono<ReviewDto> getMyRandomFilmReview(@AuthenticationPrincipal UserDetails principal){ return reviewController.getMyFirstFilmReview(principal);  }

    @ResponseStatus(HttpStatus.NO_CONTENT)
    @DeleteMapping("/delete/random")
    public Mono<Void> deleteRandomFilmReview(@AuthenticationPrincipal UserDetails principal){ return reviewController.deleteRandomFilmReview(principal);  }

    @ResponseStatus(HttpStatus.OK)
    @PutMapping("/update/random")
    public Mono<Void> updateRandomFilmReview(@AuthenticationPrincipal UserDetails principal, @RequestBody Review review){
        return reviewController.updateRandomFilmReview(principal, review);
    }

    @ResponseStatus(HttpStatus.CREATED)
    @PostMapping("/save")
    public Mono<Void> saveReview(@AuthenticationPrincipal UserDetails principal, @RequestBody Review review){return reviewController.saveReviewForAllFilms(principal, review); }
}
```

Fig. 7.2.3 Rest API *Reviews*

7.3 Controladores

Ahora bien, en el controlador de usuarios están los métodos que utilizan el repositorio para lanzar las operaciones a la base de datos y en el modo imperativo se usa de la forma común, llamar a la operación del repositorio y enviar los parámetros necesarios.

Sin embargo, en la programación reactiva se complica ligeramente, ya que para hacer la operación de forma asíncrona debemos suscribirnos a un Publisher y entonces lanzar la operación del repositorio.

```java
@Override
public Mono<Void> saveUser(Mono<Users> user) {
    return user
            .flatMap(us -> {
                us.setPassword(passwordEncoder.encode(us.getPassword()));
                return userRepository.save(us);
            }).then();
}
```

Fig. 7.3 Función reactiva *Create User*

Para hacer la suscripción tenemos la opción de *subscribe()* y también *flatMap()* hace la función de suscribirse, a diferencia de que además devuelve un objeto Mono. Entonces, *flatMap()* debe usarse para operaciones sin bloqueo, en resumen, cualquier cosa que devuelva Mono, Flux. Mientras que *map()* debe usarse cuando desea realizar la transformación de un objeto/datos, es decir operaciones síncronas.

7.3.1 Controlador de *Users* - Reactive

Estas son el resto de operaciones del controlador de *users*:

```
@RequiredArgsConstructor
@Service
public class UserController implements UserService {
    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;


    @Override
    public Mono<UsersDto> getUserDto(UserDetails principal) {return userRepository.findByUsername(principal.getUsername()).map(Utilities::entityToDto);
    }

    @Override
    public Mono<Void> deleteUser(UserDetails principal) {
        return userRepository.findByUsername(principal.getUsername()).flatMap(d -> userRepository.deleteById(d.getId()));
    }

    @Override
    public Mono<UsersDto> updateMe(UserDetails principal, Users users) {
        return userRepository.findByUsername(principal.getUsername())
                .flatMap(u -> {
                    users.setId(u.getId());
                    return userRepository.save(users);
                }).map(Utilities::entityToDto);
    }

    @Override
    public Flux<Users> getUsers() { return userRepository.findAll(); }
```

Fig. 7.3.1 Otros métodos reactivos del controlador de *users*

7.3.2 Controlador de *Films* - Reactive

Estas son el resto de operaciones del controlador de *films*:

```
@RequiredArgsConstructor
@Service
public class FilmController implements FilmService {
    private final UserRepository userRepository;
    private final FilmRepository filmRepository;

    public Flux<Film> getMyFilms(UserDetails principal) {
        return userRepository.findByUsername(principal.getUsername()).flatMapMany(u -> filmRepository.findByUserID(u.getId()));
    }

    @Override
    public Mono<FilmDto> getMyFirstFilm(UserDetails principal) { return this.getMyFilms(principal).elementAt( index: 0).map(Utilities::filmEntityToDto); }

    @Override
    public Mono<Void> deleteRandomFilm(UserDetails principal) {
        return this.getMyFilms(principal).elementAt( index: 0).flatMap( f-> filmRepository.deleteById(f.getId())).then();
    }

    @Override
    public Mono<Void> saveFilm(UserDetails principal, Film film) {
        return userRepository.findByUsername(principal.getUsername()).flatMap(u -> filmRepository.saveFilm(film.getName(), film.isMovie(), u.getId())).then();
    }

    @Override
    public Mono<FilmDto> updateRandomFilmMe(UserDetails principal, Film film) {
        return this.getMyFilms(principal).elementAt( index: 0)
                .flatMap( f-> filmRepository.updateFilm(film.getName(),film.isMovie(),f.getId())).map(f -> Utilities.filmEntityToDto(film));
    }
}
```

Fig. 7.3.2 Métodos reactivos del controlador de *films*

7.3.3 Controlador de *Reviews* - Reactive

Estas son el resto de operaciones del controlador de *reviews*:

```
@RequiredArgsConstructor
@Service
public class ReviewController implements ReviewService{
    private final UserRepository userRepository;
    private final FilmRepository filmRepository;
    private final ReviewRepository reviewRepository;

    @Override
    public Flux<Film> getMyUserWithFilmsAndReview (UserDetails principal){
        return userRepository.findByUsername(principal.getUsername())
                .flatMapMany(u -> filmRepository.findByUserID(u.getId())
                        .flatMap(f-> reviewRepository.findByFilmID(f.getId()).switchIfEmpty(Mono.just(new Review())))
                        .flatMap(r ->{
                            f.setReview(r);
                            return Mono.just(f);
                        })));
    }

    @Override
    public Mono<ReviewDto> getMyFirstFilmReview(UserDetails principal) {
        return this.getMyUserWithFilmsAndReview(principal).elementAt( index: 0).flatMap(f-> Mono.just(f.getReview())).map(Utilities::reviewEntityToDto);
    }

    @Override
    public Mono<Void> deleteRandomFilmReview(UserDetails principal) {
        return this.getMyUserWithFilmsAndReview(principal).elementAt( index: 0)
                .flatMap(f -> reviewRepository.deleteById(f.getReview().getId()));
    }

    @Override
    public Mono<Void> saveReviewForAllFilms(UserDetails principal, Review review) {
        return this.getMyUserWithFilmsAndReview(principal)
                .flatMap(u -> {
                    review.setId(null);
                    review.setFilmID(u.getId());
                    return reviewRepository.save(review);}).then();
    }

    @Override
    public Mono<Void> updateRandomFilmReview(UserDetails principal, Review review) {
        return this.getMyUserWithFilmsAndReview(principal).elementAt( index: 0)
                .flatMap(f -> reviewRepository.updateReview(review.getContent(), review.getScore(), f.getReview().getId()));
    }
}
```

Fig. 7.3.3 Métodos reactivos del controlador de *reviews*

8. Pruebas de Carga y Estrés

En este apartado se realizan algunos test de carga y estrés hacia nuestra aplicación. Para dichos test se utiliza JMeter y en algunos casos un csv que contienen información para hacer las peticiones. Por otro lado, también se usa el plugin de VisualVM en IntelliJ para poder monitorear los recursos del sistema durante algunas de las pruebas, y ver el uso de memoria y de CPU.

Para hacer las pruebas de la forma más equitativa posible se han enviado cien mil peticiones antes de hacer la prueba real, para que la aplicación se haya calentado (warm-up).

8.1 Funcionalidad 1: *Users*

Los test para esta primera funcionalidad únicamente se centran en realizar peticiones relacionadas con los usuarios.

8.1.1 Prueba de Carga: *Create User*

Para este test se utiliza un csv con mil usuarios con nombres de usuarios distintos ya que tendrán clave única y no se podrán repetir (mismo nombre/apellido/contraseña). Además, se quiere evaluar la capacidad de cada aplicación de recibir estas mil peticiones en treinta segundos, aunque no habrá una gran diferencia al no ser una gran cantidad de peticiones. En el test de Estrés es donde realmente podremos comprobar si hay diferencias.

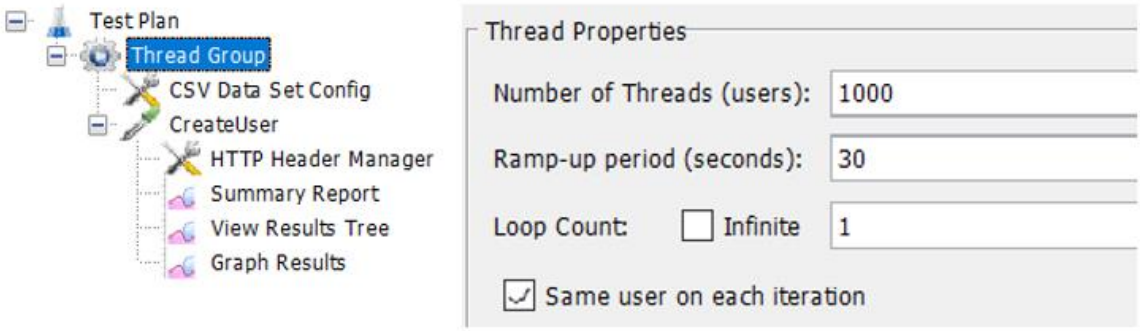


Fig. 8.1.1 Configuración del primer test de JMeter.

8.1.1.2 Resultados Prueba de Carga: *Create User* en MVC

Como se puede observar, el tiempo de respuesta mínimo es de cero segundos, pero es debido a alguno de los fallos, y la máxima de diecisiete segundos. Esos errores no son significativos ya que pueden ser debidos a un fallo en la conexión ajenos a la aplicación, estos errores han aparecido a lo largo del testeo. Además, hay una salida de veintiocho transacciones por segundo.

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	1000	3	0.30%	4144.61	0	17169	4502.00	5812.90	6132.35	10201.77	28.65	20.71	7.50
CreateUser	1000	3	0.30%	4144.61	0	17169	4502.00	5812.90	6132.35	10201.77	28.65	20.71	7.50

Fig. 8.1.1.2.1 Estadísticas básicas resultados de Carga. *Users* - MVC

Aquí se puede ver como incrementa el tiempo de respuesta a través del tiempo, y que a mayor número de request mayor es el tiempo de respuesta, esto se puede ver como un potencial cuello de botella.

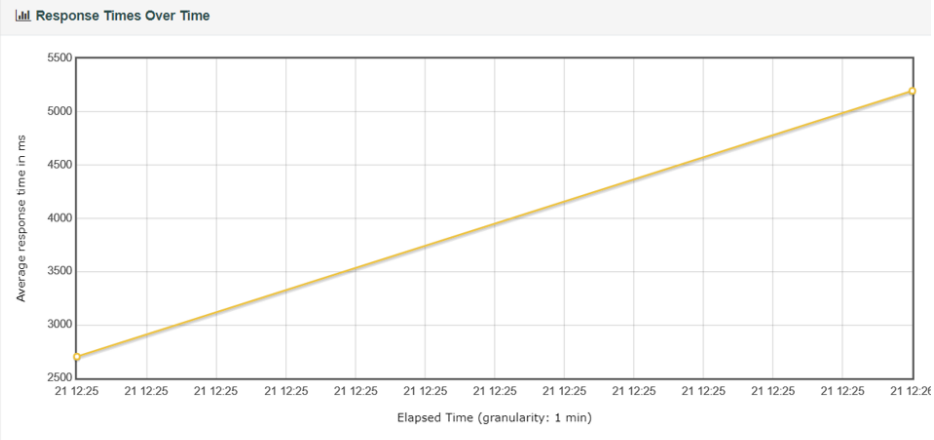


Fig. 8.1.1.2.2 Tiempo de respuesta a través del tiempo. *Users* - MVC

Como se puede apreciar, el uso de *threads* es incremental, lo cual nos muestra que cada vez necesita más de ellos y podría llegar al límite si tardase más en enviarse las peticiones. Hay que recalcar que los *Threads* activos en este caso se refiere a los de JMeter.

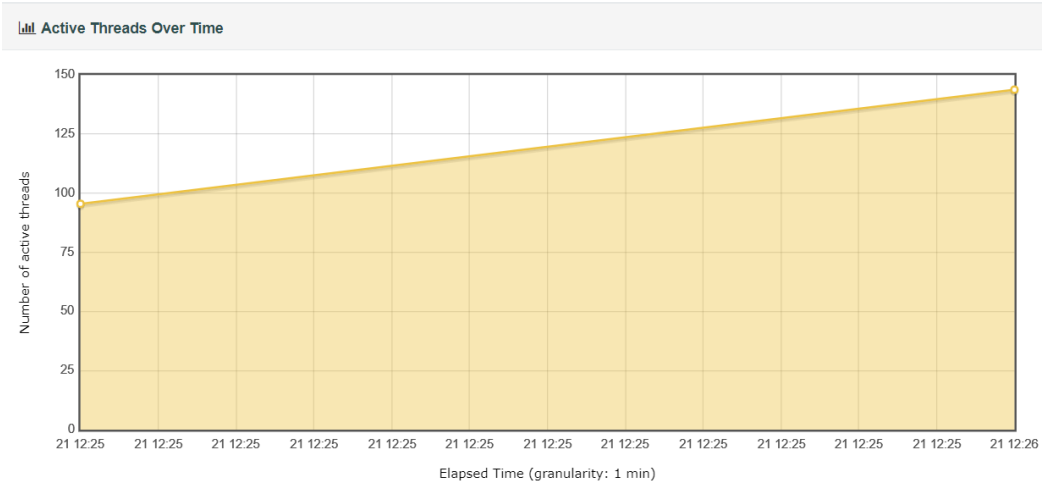


Fig. 8.1.1.2.3 *Threads* activos a través del tiempo. *Users* - MVC

8.1.1.3 Resultados Prueba de Carga: *Create User* en Reactive

Como se puede observar, el tiempo de respuesta mínimo es de un milisegundo, pero es debido a alguno de los fallos, y la máxima de tres con cinco segundos, una gran diferencia respecto a la solución imperativa. Además, hay con una ligera ventaja de treinta transacciones por segundo.

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	1000	5	0.50%	2522.21	1	3624	2668.00	3491.90	3536.95	3566.97	30.58	11.73	7.99
CreateUser	1000	5	0.50%	2522.21	1	3624	2668.00	3491.90	3536.95	3566.97	30.58	11.73	7.99

Fig. 8.1.1.3.1 Estadísticas básicas resultados de Carga. *Users* - Reactive

En este caso también aumenta bastante el tiempo de respuesta cuantas más request llegan, sin embargo, el tiempo de respuesta es tres veces menor. Además, el tiempo de respuesta se conserva más o menos estable sin muchas variaciones.

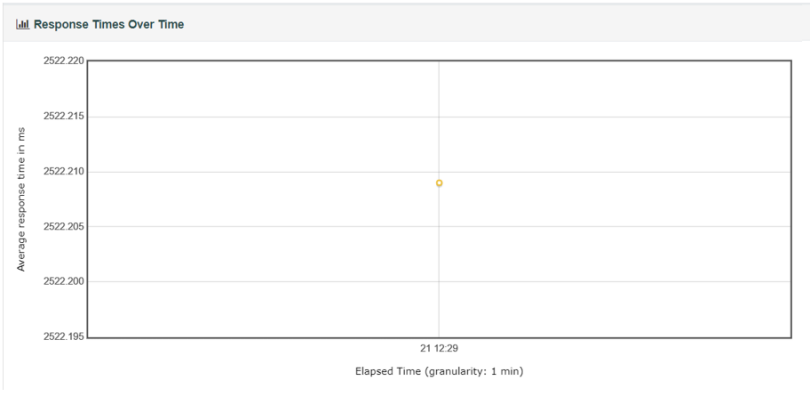


Fig. 8.1.1.3.2 Tiempo de respuesta a través del tiempo. *Users* - Reactive

Aquí se puede ver como el consumo de *threads* es casi siempre el mismo por lo que está aprovechando los recursos de manera más eficiente. Hay que recalcar que los *Threads* activos en este caso se refiere a los de JMeter.

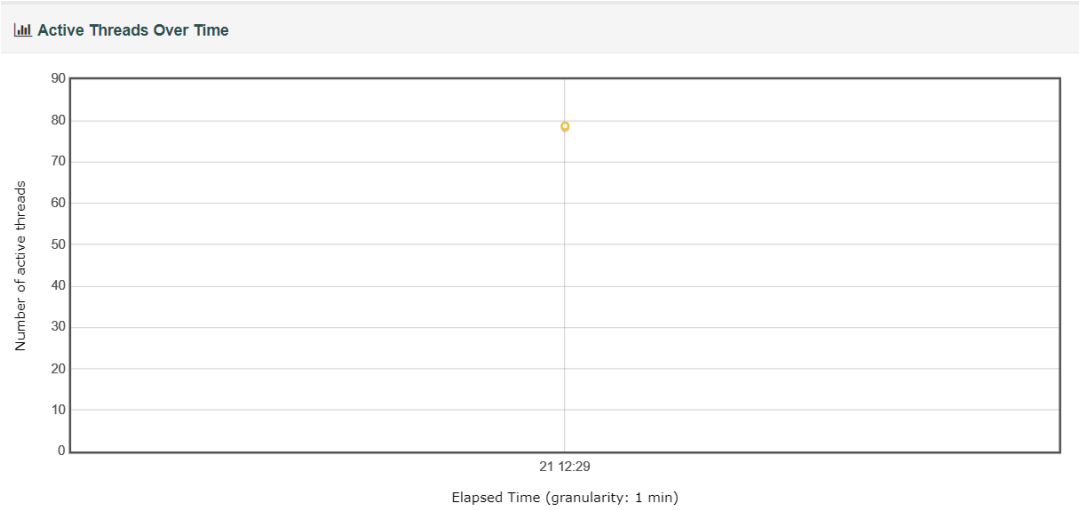


Fig. 8.1.1.3.3 *Threads* Activos a través del tiempo. *Users* - Reactive

8.1.2 Pruebas de Estrés

Para este tipo de test lo que se busca es hacer que la aplicación falle. Por ello se envía una cantidad inicial de quinientas peticiones. Después se incrementa la cantidad añadiendo cada vez más usuarios hasta que en los resultados hay una diferencia de error notoria.

Al ser un test de estrés en el que solo se quiere conocer la capacidad de la aplicación para soportar una determinada carga de usuarios concurrentes y se busca el error del sistema, solo se muestran las estadísticas básicas de la prueba.

8.1.2.1 Prueba de Estrés: *Get One User*

Para esta prueba se llama a la API pidiendo que devuelva el usuario con el nombre de usuario que se envía a través del token, y se ha establecido el límite en mil usuarios concurrentes. La prueba se repite cinco veces para hacer la media de los resultados.

8.1.2.1.1 Resultados Prueba de Estrés: *Get One User* - MVC

En este caso se ven buenos resultados en cuanto a tiempo de respuesta. Sin embargo, el porcentaje de error supera por mucho el de la aplicación reactiva, además de tener una salida de peticiones por segundo bastante inferior, debido a la congestión que sufre la aplicación.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
GetMyUser	5000	773	0	1886	402.75	13.34%	20.2/sec	19.33	6.50	978.8
TOTAL	5000	773	0	1886	402.75	13.34%	20.2/sec	19.33	6.50	978.8

Fig. 8.1.2.1.1 Resultados prueba de Estrés. *Get One User* - MVC

8.1.2.1.2 Resultados Prueba de Estrés: *Get One User* - Reactive

Para la solución reactiva el tiempo de respuesta es ligeramente mayor a la solución tradicional. Sin embargo, al procesar de panera más constante la salida, tiene una salida de peticiones por segundo muy superior a la solución imperativa, ya que en este caso hay muy pocos errores en las peticiones.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
GetMyUser	5000	1619	0	2947	724.73	0.14%	111.7/sec	55.83	36.93	511.8
TOTAL	5000	1619	0	2947	724.73	0.14%	111.7/sec	55.83	36.93	511.8

Fig. 8.1.2.1.2 Resultados prueba de Estrés. *Get One User* - Reactive

8.1.2.1.3 Comparativa Uso de Memoria y CPU: *Get One User*

Como se puede observar en la aplicación reactiva llega a utilizar ochenta por ciento de la CPU y un uso de memoria aumenta hasta ser el mismo que la solución imperativa. Además, se puede ver que en este caso solo ha creado veinte *threads* nuevos, para usar un total de cincuenta. Aquí se puede ver como con pocos *threads* y no demasiados recursos consigue soportar la carga de la prueba.

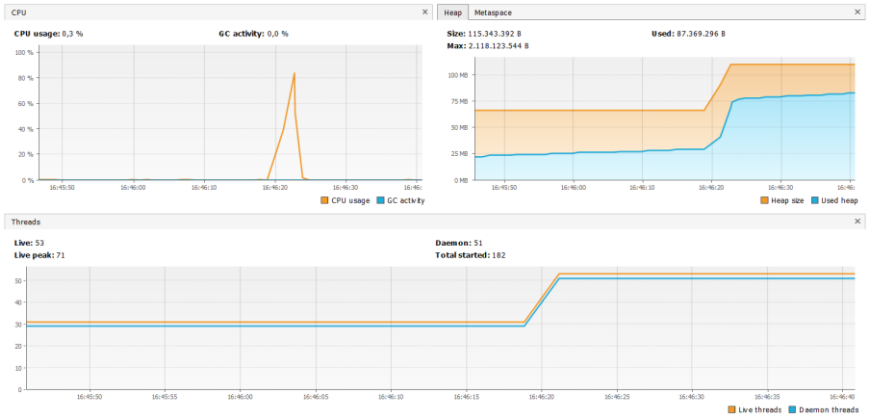


Fig. 8.1.2.1.3.1 Uso de Recursos. *Get One User* - Reactive

En el caso de la aplicación imperativa el uso máximo no llega ni a un veinte por ciento y el uso de memoria se mantiene en una subida constante sin llegar a utilizar una gran cantidad, esto es debido a que al tener tantos fallos y una salida peticiones por segundos baja no llega a utilizar muchos recursos. Por otra parte, esta aplicación utiliza doscientos *threads* nuevos, esto se debe a el servlet que utiliza cada aplicación, en este caso Tomcat, Netty para la reactiva.

Fig. 8.1.2.1.3.2 Uso de Recursos. *Get One User* - MVC

8.1.2.2 Prueba de Estrés: *Get All Users*

Para este test se realiza una petición que devuelve todos los usuarios en la base de datos, actualmente mil usuarios, creados en la prueba de carga. Nuevamente se ha llegado al límite de mil peticiones.

8.1.2.2.1 Resultados Prueba de Estrés: *Get All Users* - MVC

Al realizar esta prueba se ve que el tiempo de respuesta es bastante bajo, no muy superior a la prueba anterior. Se puede ver que cada petición recibe una gran cantidad de datos. Al haber tantos errores hace que la salida de peticiones por segundo parezca más alta. Además, sigue teniendo un gran porcentaje de error siendo casi un tercio del total.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
GetMyUser	5000	1603	0	4989	1350.47	30.16%	27.5/sec	2803.49	7.08	104561.5
TOTAL	5000	1603	0	4989	1350.47	30.16%	27.5/sec	2803.49	7.08	104561.5

Fig. 8.1.2.2.1 Resultados prueba de Estrés. *Get All Users* - MVC

8.1.2.2.2 Resultados Prueba de Estrés: *Get All Users* - Reactive

En este caso la mejora es muy notoria, siendo el porcentaje de error prácticamente cero. Sin embargo, el tiempo de respuesta es mucho mayor a la solución imperativa y al recibir más cantidad de datos la salida de peticiones por segundo disminuye. También cabe destacar que esta prueba recibe mayor cantidad de datos por segundo.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
GetMyUser	5000	17901	0	36303	9210.90	0.04%	18.6/sec	3130.66	6.13	171985.2
TOTAL	5000	17901	0	36303	9210.90	0.04%	18.6/sec	3130.66	6.13	171985.2

Fig. 8.1.2.2.2 Resultados prueba de Estrés. *Get All Users* - Reactive

8.1.2.2.3 Comparativa Uso de Memoria y CPU: *Get All Users*

Al ser una prueba que toma más tiempo en procesar, se puede ver que al principio utiliza hasta un sesenta por ciento de CPU y luego se mantiene constante hasta finalizar. El uso de memoria en este caso va subiendo regularmente hasta llegar a trescientos megabytes. En este caso llega a crear diez nuevos *threads*, llegando a usar cincuenta de forma activa.

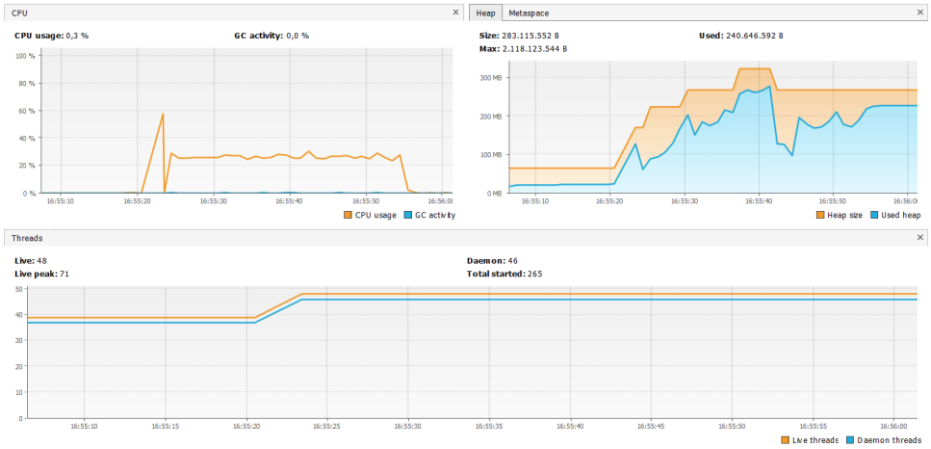


Fig. 8.1.2.2.3.1 Uso de Recursos. *Get All Users* - Reactive

En el caso de la aplicación imperativa llega a la máxima ocupación de CPU de forma drástica y luego en picado, esto se debe a la gran sobrecarga de datos que se crea. El uso de memoria no aumenta demasiado y se mantiene entre cien y ciento cincuenta. Como la prueba anterior utiliza hasta aproximadamente doscientos cincuenta *threads* activos.

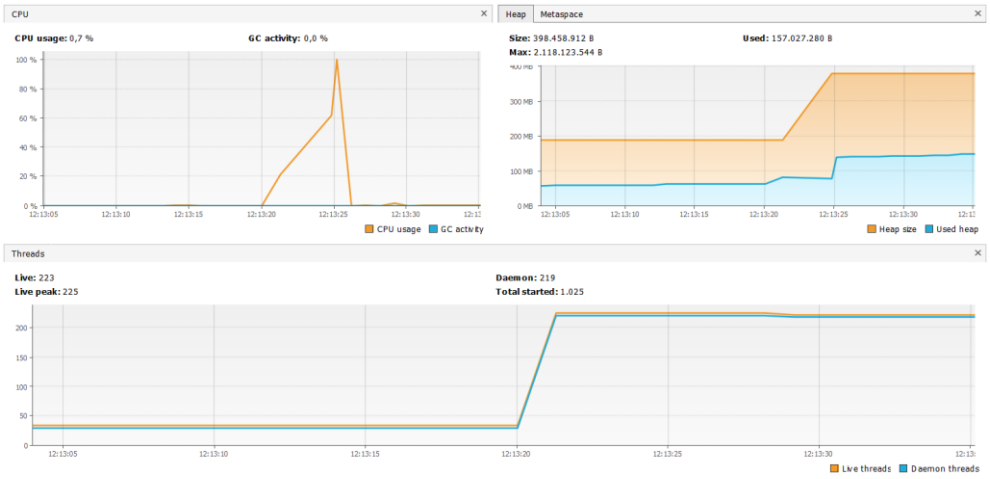


Fig. 8.1.2.2.3.2 Uso de Recursos. *Get All Users* - MVC

8.1.2.3 Prueba de Estrés: *Update User*

En esta prueba se envían mil peticiones a la vez, estas peticiones modifican siempre el mismo usuario que llega a través de un token.

8.1.2.3.1 Resultados Prueba de Estrés: *Update User* - MVC

Al realizar esta prueba se ve que el tiempo de respuesta es bastante alto, en contraste con las anteriores pruebas para esta aplicación. El procesamiento de respuestas por segundo también es menor. Sin embargo, el porcentaje de error sigue siendo, y esto se debe a que tiene más operaciones bloqueantes dentro de la lógica.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
Update Request	5000	7749	0	30923	7917.21	32.46%	18.2/sec	23.82	6.25	1337.9
TOTAL	5000	7749	0	30923	7917.21	32.46%	18.2/sec	23.82	6.25	1337.9

Fig. 8.1.2.3.1 Resultados prueba de Estrés. *Update User* - MVC

8.1.2.3.2 Resultados Prueba de Estrés: *Update User* - Reactive

Para la solución reactiva también hay una gran mejora, ya que no ha aparecido ningún error. El procesamiento de respuestas por segundo es parecido a las otras pruebas, y en este caso tiene una gran mejora respecto al tiempo de respuesta respecto a la solución imperativa.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
Update Request	5000	2381	310	3471	880.91	0.00%	11.2/sec	5.08	5.22	466.0
TOTAL	5000	2381	310	3471	880.91	0.00%	11.2/sec	5.08	5.22	466.0

Fig. 8.1.2.3.2 Resultados prueba de Estrés. *Update User* - Reactive

8.1.2.3.3 Comparativa Uso de Memoria y CPU: *Update User*

En este caso el uso de CPU de la solución reactiva crece hasta llegar un ochenta por ciento, luego sufre una caída y vuelve a subir a veinte. El uso de memoria es bastante regular subiendo hasta el doble en la mitad de la prueba. En este caso crea diez *threads* y se mantiene con un uso de cuarenta de forma activa.

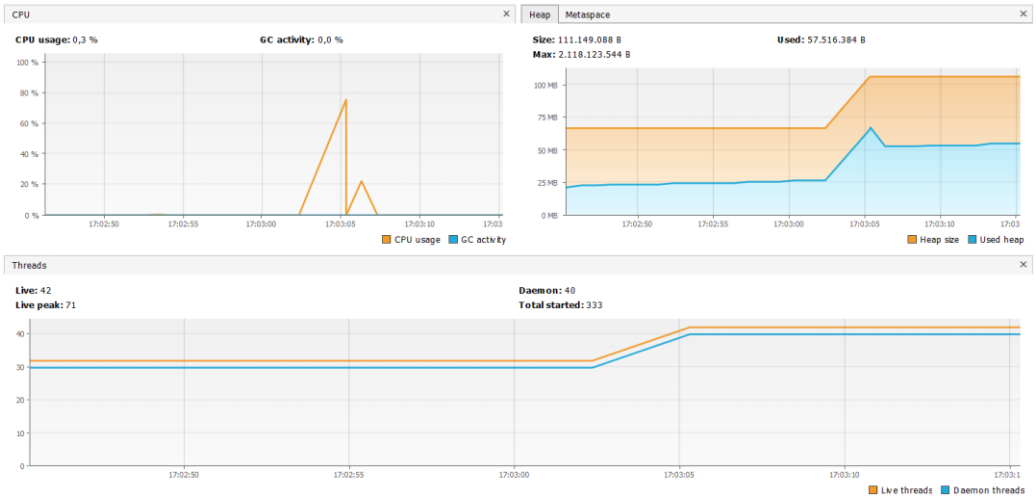


Fig. 8.1.2.3.3.1 Uso de Recursos. *Update User* - Reactive

Para la aplicación imperativa la CPU mantiene un uso del máximo, destacando una caída hasta cero, pero teniendo un uso constante cercano a un cien por ciento. La memoria usada se no aumenta mucho, pero tiene subidas y bajadas irregulares. También mantiene un uso de más de doscientos *threads* activos.

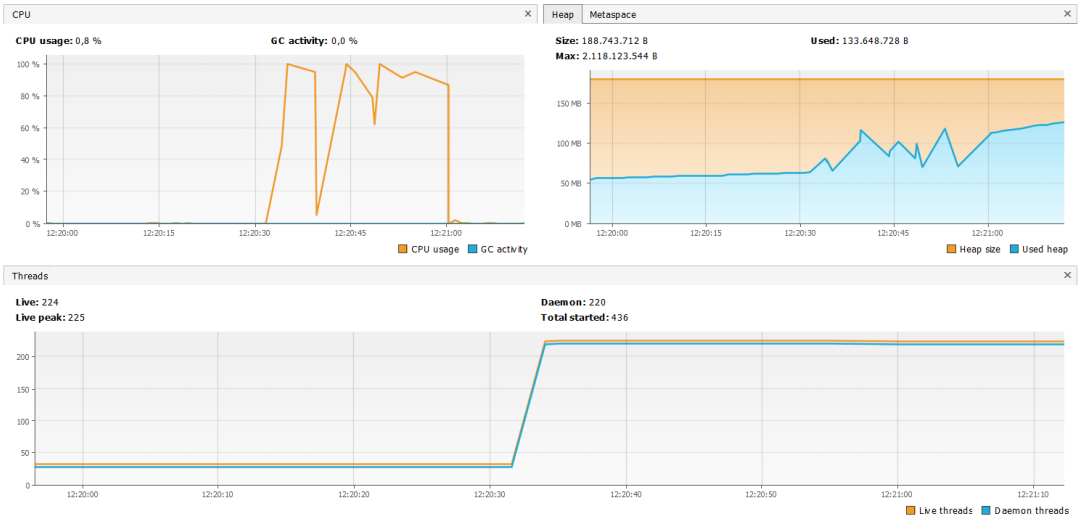


Fig. 8.1.2.3.3.2 Uso de Recursos. *Update User* - MVC

8.2 Funcionalidad 2: *Films*

Los test para esta segunda funcionalidad se centran en realizar peticiones relacionadas con las películas, pero estas peticiones en el back-end hacen llamadas tanto a los repositorios de *users* como de *films*, como se ha mostrado en el apartado número siete.

8.2.1 Prueba de Carga: *Create Film*

Para este test se crean mil *films* para un usuario, que se pasa a través de los tokens de JWT, para ver cómo reaccionan los dos sistemas al lanzar las mil peticiones. La configuración es igual que en la funcionalidad anterior, pero en lugar lanzar las peticiones en treinta segundos lo hacen en diez. Nuevamente se crean los usuarios cogiendo los datos de un csv de mil filas.

8.2.1.2 Resultados Prueba de Carga: *Create Film* - MVC

Como se puede observar, el tiempo de respuesta mínimo es de veinte milisegundos, pero es debido a alguno de los fallos, y el máximo es parecido a la solución reactiva pero la media es bastante inferior. Además, realiza un procesamiento de cien peticiones por segundo, un numero bastante superior a la solución reactiva.

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	1000	0	0.00%	475.17	20	4865	251.50	1232.00	1912.45	3548.57	105.50	61.61	46.05
HTTP Request	1000	0	0.00%	475.17	20	4865	251.50	1232.00	1912.45	3548.57	105.50	61.61	46.05

Fig. 8.2.1.2.1 Estadísticas básicas resultados de Carga. *Films* - MVC

En este caso la mayoría de peticiones tienen un tiempo de respuesta menor a quinientos milisegundos, un setenta por ciento de las peticiones aproximadamente, siendo un gran resultado.

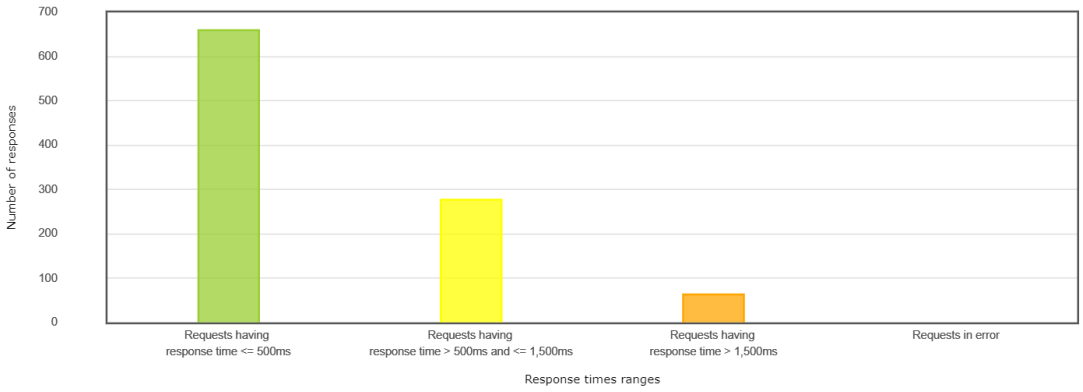


Fig. 8.2.1.2.2 Rangos de tiempo de respuesta. *Films* - MVC

Como se puede apreciar, el número de *threads* activos se mantiene constante alrededor de cincuenta, dado a la alta velocidad de procesamiento de JDBC y de Tomcat cuando no hay una gran sobrecarga.

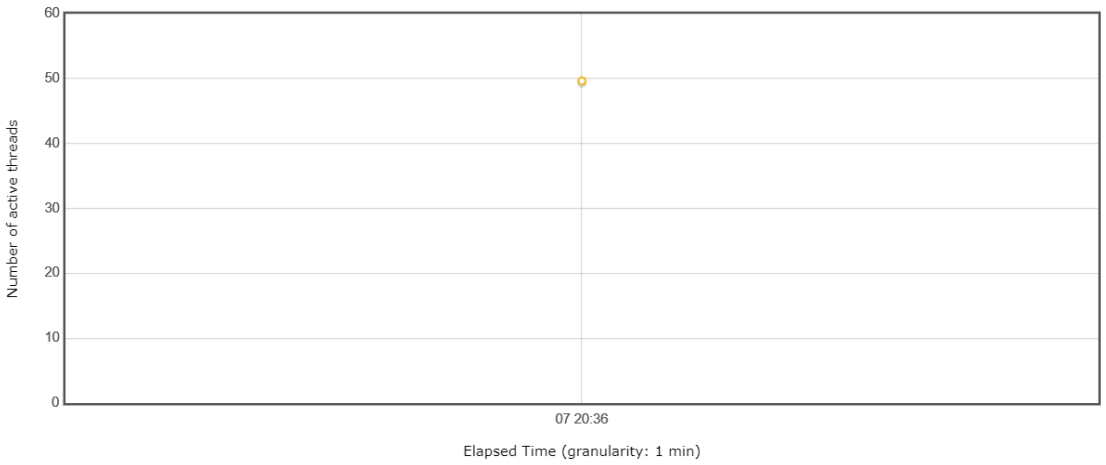


Fig. 8.2.1.2.3 *Threads* activos a través del tiempo. *Films* - MVC

8.2.1.3 Resultados Prueba de Carga: *Create Film* - Reactive

Como se puede observar para la solución reactiva, el tiempo de respuesta mínimo es de un segundo, pero es debido a ese único fallo, y el máximo es parecido a la solución tradicional, siendo la media el triple. Además, realiza menos transacciones por segundo, con una cantidad de setenta. Con este ejemplo podemos ver que al ser una carga de peticiones baja y que permite un mayor margen de respuesta, la solución tradicional es bastante mejor en cuanto a velocidad.

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**1000**	**1**	**0.10%**	**1465.96**	**1**	**4765**	**417.00**	**4205.80**	**4460.60**	**4725.00**	**72.92**	**24.72**	**28.88**
HTTP Request	1000	1	0.10%	1465.96	1	4765	417.00	4205.80	4460.60	4725.00	72.92	24.72	28.88

Fig. 8.2.1.3.1 Estadísticas básicas resultados de Carga. *Films* - Reactive

En este caso también tiene bastantes peticiones con tiempo de respuesta menor a quinientos milisegundos, siendo la mitad. Pero también hay bastantes con tiempo de respuesta mayor a uno con cinco segundos.

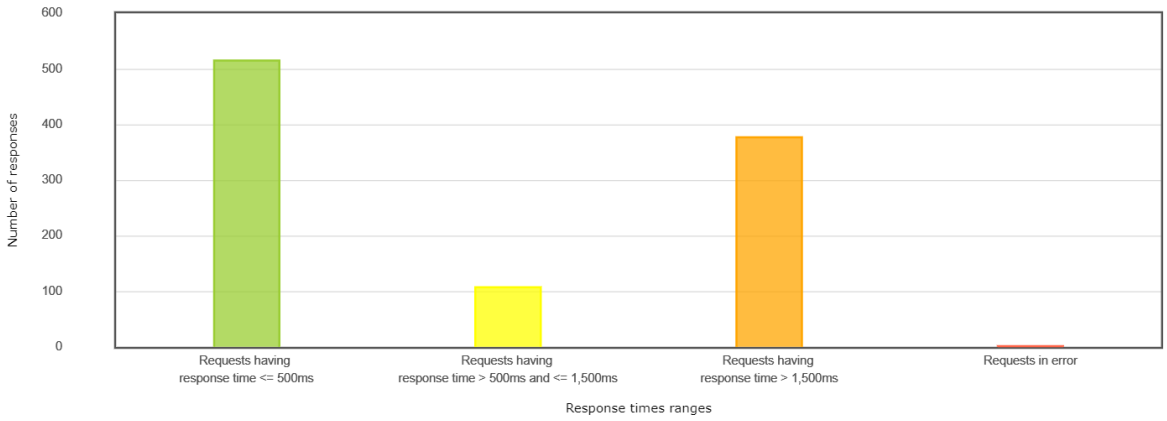


Fig. 8.2.1.3.2 Rangos de tiempo de respuesta. *Films* - Reactive

En este caso, los *threads* activos se incrementan hasta llegar a cien, el doble que la solución tradicional. Con esto se puede reafirmar que al tener poca carga de peticiones el driver de r2dbc y el servlet de Netty no son los más eficientes. En comparación a la creación de usuarios esto se debe a que cada vez que se crea un *film* la siguiente petición también la recupera. Ya que cada vez que se llama a esta función se recogen todos los *films* de este usuario.

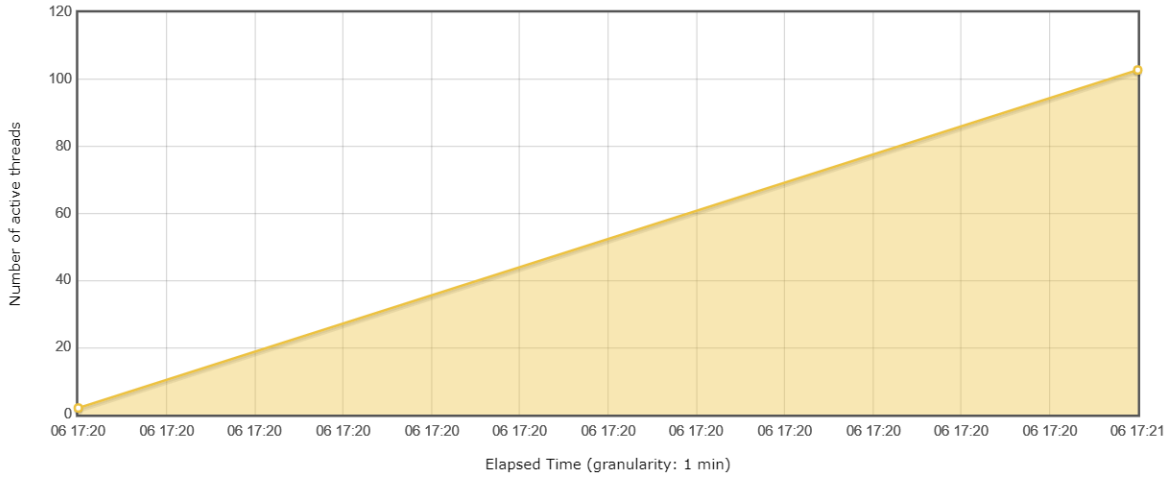


Fig. 8.2.1.3.3 *Threads* Activos a través del tiempo. *Films* - Reactive

8.2.2 Pruebas de Estrés

Para este test lo que se busca es hacer que la aplicación falle. Por ello se envía una cantidad inicial de quinientos usuarios tratando siempre los *films* del mismo usuario. Después se incrementa la cantidad añadiendo más peticiones hasta que en los resultados hay una gran diferencia de porcentaje de error. Las pruebas se repiten cinco veces para hacer la media de todos los resultados.

Al ser un test de estrés en el que solo se quiere conocer la capacidad de la aplicación para soportar una determinada carga de usuarios concurrentes y se busca el error del sistema, solo se muestran las estadísticas básicas de la prueba.

8.2.2.1 Prueba de Estrés: *Get One Film*

En esta prueba las peticiones llaman a una función que devuelve la primera de todos los *films* de un usuario especificado a través de un token. Llega a soportar mil usuarios concurrentes. En el caso de la aplicación reactiva, la función devuelve un usuario aleatorio, ya que al devolver un flujo de datos no garantiza que el primero que se reciba sea el primero guardado en la base de datos, mientras que en la solución imperativa si se recupera el primer *film*.

8.2.2.1.1 Resultados Prueba de Estrés: *Get One Film* - MVC

En este caso se ven buenos resultados en cuanto a tiempo de respuesta y a procesamiento de peticiones por segundo. El porcentaje de error es un cuarto de las peticiones totales.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
HTTP Request	5000	1394	0	4878	1235.91	25.78%	33.2/sec	36.76	9.31	1133.7
TOTAL	5000	1394	0	4878	1235.91	25.78%	33.2/sec	36.76	9.31	1133.7

Fig. 8.2.2.1.1 Resultados prueba de Estrés. *Get One Film* - MVC

8.2.2.1.2 Resultados Prueba de Estrés: *Get One Film* - Reactive

Para la solución reactiva el tiempo de respuesta es mayor a la solución tradicional. Por otro lado, esto puede deberse a que procesa correctamente todas las peticiones. Además, tiene casi la misma salida de peticiones por segundo.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	5000	3782	288	6457	1114.28	0.00%	32.9/sec	15.70	11.14	489.0
TOTAL	5000	3782	288	6457	1114.28	0.00%	32.9/sec	15.70	11.14	489.0

Fig. 8.2.2.1.2 Resultados prueba de Estrés. *Get One Film* - Reactive

8.2.2.1.3 Comparativa Uso de Memoria y CPU: *Get One Film*

En este caso el uso de CPU de la solución reactiva crece hasta llegar a un noventa por ciento, luego sufre una caída y vuelve a subir a veinte. El uso de memoria es bastante regular subiendo hasta ciento cincuenta megabytes. En este caso crea treinta *threads* y se mantiene con un uso de sesenta de forma activa.

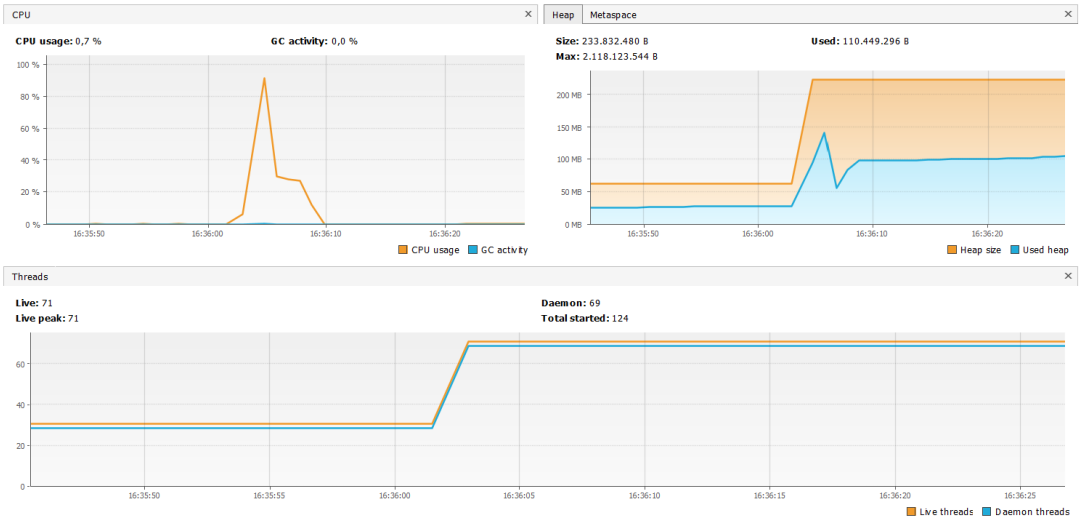


Fig. 8.2.2.1.3.1 Uso de Recursos. *Get One Film* - Reactive

Para la aplicación imperativa la CPU y la memoria tienen un incremento constante sin subir demasiado, por lo que se ve no aprovecha de manera óptima los recursos. También mantiene un uso de más de doscientos *threads* activos.

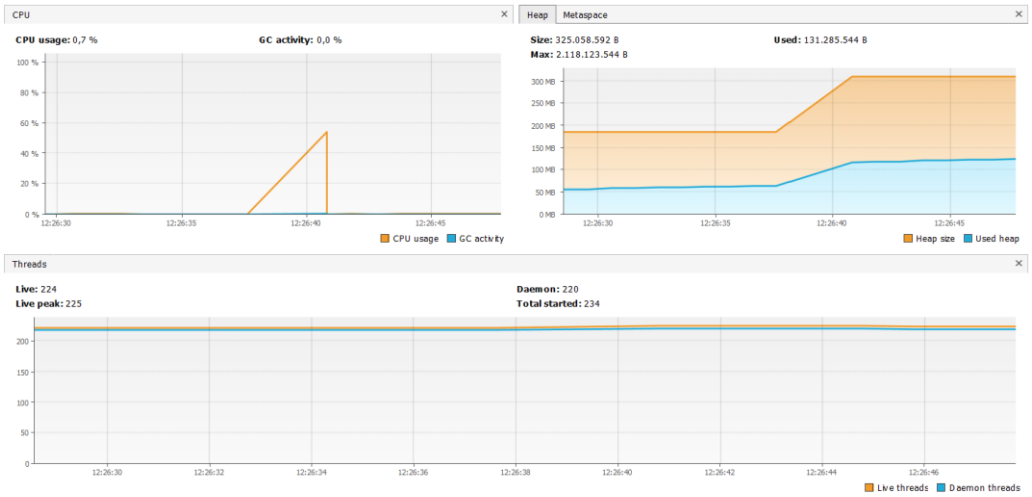


Fig. 8.2.2.1.3.2 Uso de Recursos. *Get One Film* - MVC

8.2.2.2 Prueba de Estrés: *Get All Films*

Para este test se llegan nuevamente a realizar mil peticiones concurrentes, que devuelven todos los *films* de un usuario, en este caso mil *films*.

8.2.2.2.1 Resultados Prueba de Estrés: *Get All Films* - MVC

Al realizar esta prueba se ve que el tiempo de respuesta se reduce levemente respecto a la prueba anterior. El procesamiento de respuestas por segundo y los datos recibidos por segundo son altos ya que recupera mil filas de la base de datos. Además, sigue teniendo un gran porcentaje de error siendo de un cuarto del total.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
HTTP Request	5000	1295	0	4019	1111.97	29.36%	32.4/sec	2579.10	8.51	81611.2
TOTAL	5000	1295	0	4019	1111.97	29.36%	32.4/sec	2579.10	8.51	81611.2

Fig. 8.2.2.2.1 Resultados prueba de Estrés. *Get All Films* - MVC

8.2.2.2.2 Resultados Prueba de Estrés: *Get All Films* - Reactive

En este caso la mejora es muy notoria, siendo el porcentaje de error igual a cero, y el tiempo de respuesta máximo es bastante alto ya que recibe una gran cantidad de datos. Tiene un buen resultado en cuanto a transacciones por segundo.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
HTTP Request	5000	14001	381	26256	6732.77	0.00%	24.3/sec	2749.79	8.06	115942.0
TOTAL	5000	14001	381	26256	6732.77	0.00%	24.3/sec	2749.79	8.06	115942.0

Fig. 8.2.2.2.2 Resultados prueba de Estrés. *Get All Films* - Reactive

8.2.2.2.3 Comparativa Uso de Memoria y CPU: *Get All Films*

En este caso el uso de CPU de la solución reactiva crece hasta llegar a un cien por ciento, luego baja y se mantiene constante entorno al veinte por ciento. El uso de memoria va subiendo junto al número de peticiones que llega, hasta doscientos cincuenta megabytes. En este caso crea cinco *threads* y se mantiene con un uso de cuarenta de forma activa.

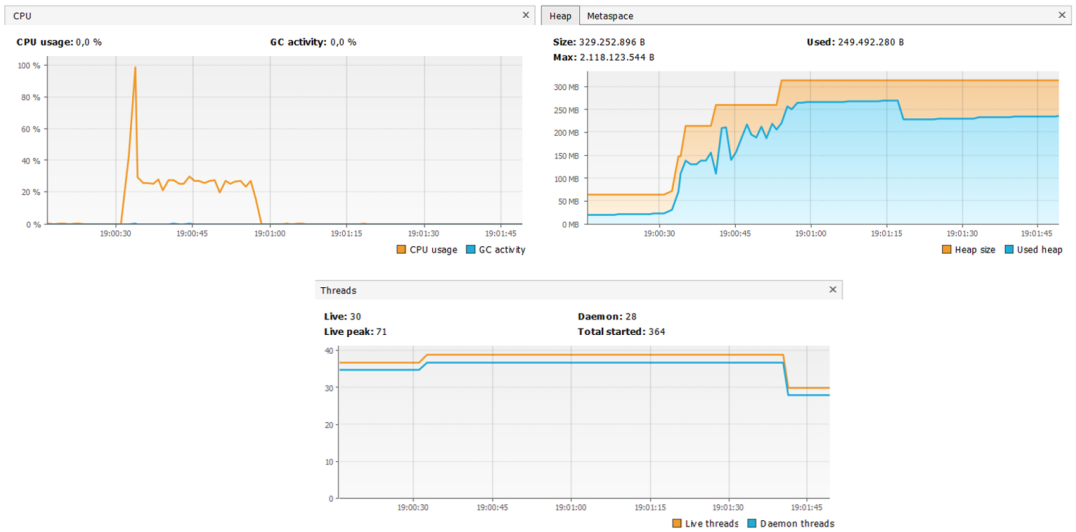


Fig. 8.2.2.2.3.1 Uso de Recursos. *Get All Films* - Reactive

Para la aplicación imperativa la CPU llega al máximo de forma muy rápida y luego cae en picado. En cuanto a la memoria también sube ciento cincuenta megabytes de golpe. También mantiene un uso de más de doscientos *threads* activos.

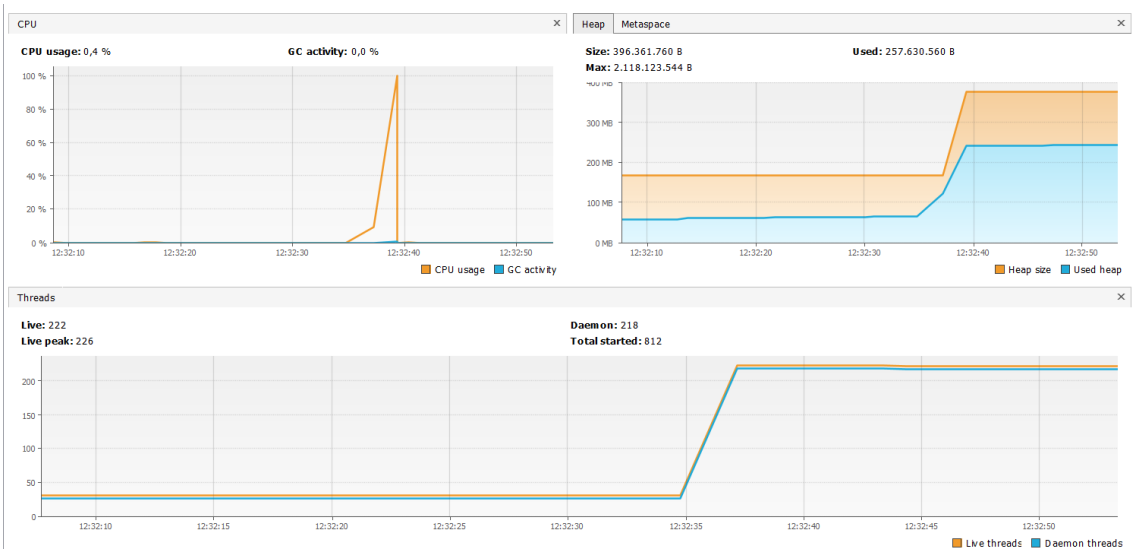


Fig. 8.2.2.2.3.2 Uso de Recursos. *Get All Films* - MVC

8.2.2.3 Prueba de Estrés: *Update Film*

En este caso, se realizan peticiones actualizando un film aleatorio en el caso de la aplicación reactiva, y el primer film en el caso de la imperativa. Nuevamente se ha llegado a un resultado con mil peticiones concurrentes.

8.2.2.3.1 Resultados Prueba de Estrés: *Update Film* - MVC

Al realizar esta prueba se ve que el tiempo de respuesta máximo y medio es bajo, en concordancia con las anteriores pruebas. El procesamiento de respuestas por segundo, baja levemente debido a que es una operación de actualización y no es tan rápida como las operaciones de consulta. Sin embargo, de nuevo el porcentaje de error es de más de un tercio del total.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
HTTP Request	5000	1181	0	3859	1158.45	37.86%	24.4/sec	32.65	6.85	1372.0
TOTAL	5000	1181	0	3859	1158.45	37.86%	24.4/sec	32.65	6.85	1372.0

Fig. 8.2.2.3.1 Resultados prueba de Estrés. *Update Film* - MVC

8.2.2.3.2 Resultados Prueba de Estrés: *Update Film* - Reactive

Para la solución reactiva también hay una mejora enorme, ya que el porcentaje de error es prácticamente cero. El procesamiento de respuestas por segundo es un poco mayor a la solución imperativa pero el tiempo medio es cercano al máximo. Encontramos una mejora respecto a la salida de transacciones por segundo.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	5000	4907	0	5980	964.09	0.08%	18.3/sec	6.67	7.55	373.7
TOTAL	5000	4907	0	5980	964.09	0.08%	18.3/sec	6.67	7.55	373.7

Fig. 8.2.2.3.2 Resultados prueba de Estrés. *Update Film* - Reactive

8.2.2.3.3 Comparativa Uso de Memoria y CPU: *Update Film*

En este caso el uso de CPU de la solución reactiva llega hasta el sesenta por ciento, luego baja y sube hasta el veinte por ciento, se ve que no se le exige mucho al procesador. El uso de memoria sube bastante, hasta doscientos cincuenta megabytes. En este caso crea veinte *threads* y se mantiene con un uso de cincuenta de forma activa.

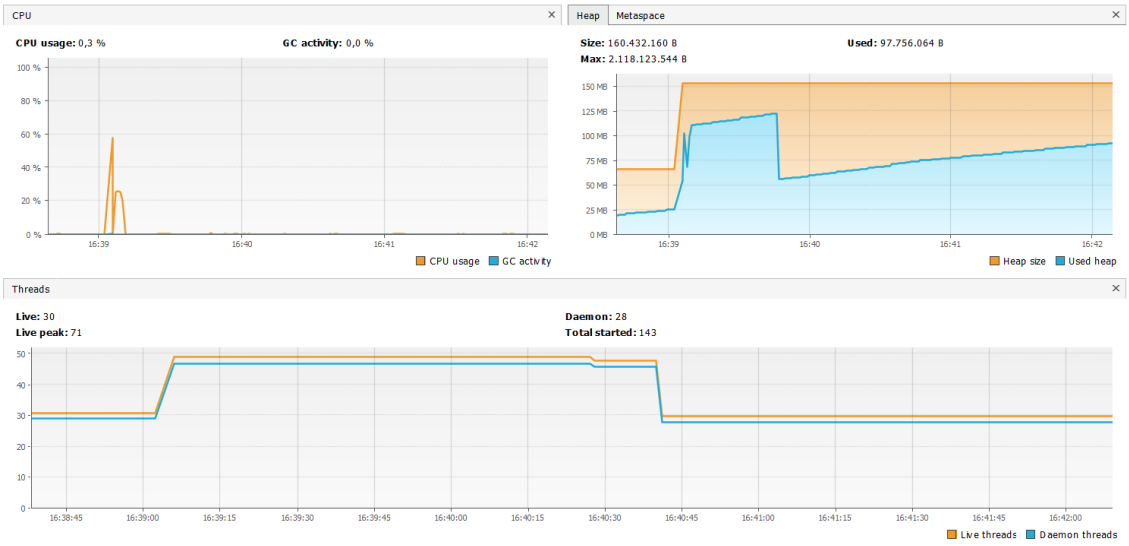


Fig. 8.2.2.3.3.1 Uso de Recursos. *Update Film* - Reactive

Para la aplicación imperativa la CPU llega al ochenta por ciento de golpe y mantiene un uso elevado. En cuanto a la memoria también sube doscientos megabytes de golpe. También mantiene un uso de más de doscientos *threads* activos.

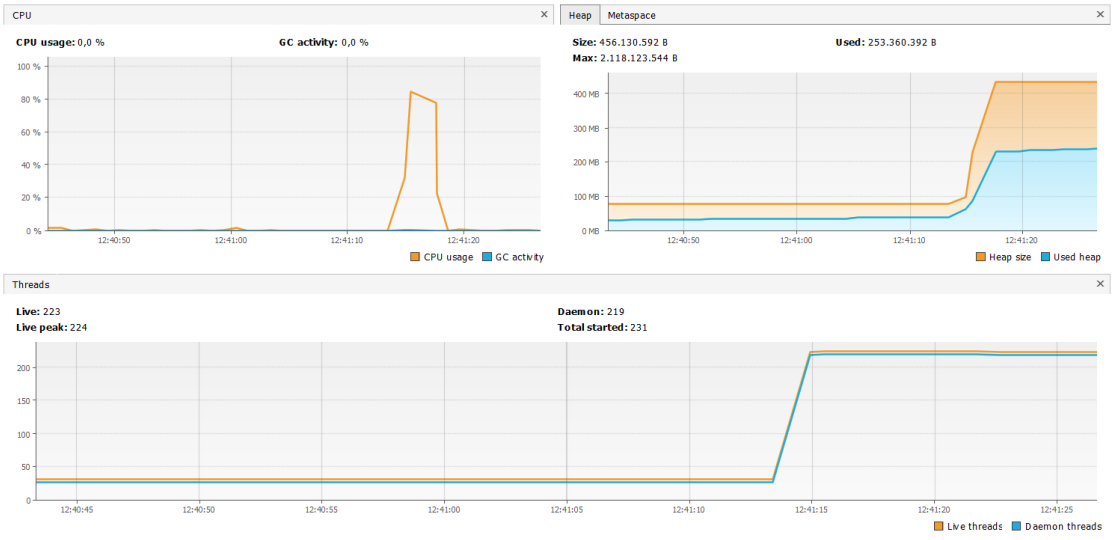


Fig. 8.2.2.3.3.2 Uso de Recursos. *Update Film* - MVC

8.3 Funcionalidad 3: *Reviews*

Los test para esta segunda funcionalidad se centran en realizar peticiones relacionadas con las reseñas, pero estas peticiones en el backend hacen llamadas tanto a los repositorios de *users* como de *films* y *reviews,* como se ha mostrado en el apartado número siete. Esta será la prueba con operaciones más costosas, en este caso cada llamada recupera cien *films* con sus *reviews* y modifica la primera *review* de la lista.

8.3.1 Prueba de Carga: *Update Review*

Para este test se hará update de la *review* de un usuario para ver cómo reaccionan los dos sistemas al lanzar las dos mil peticiones en diez segundos. Se ha optado por hacer la funcionalidad de *update* en vez de *create* debido a que esta última tiene un costo demasiado grande y requiere mucho tiempo para hacer las pruebas.

8.3.1.2 Resultados Prueba de Carga: *Update Review* - MVC

Como se puede observar, la máxima de diecisiete segundos que no es muy alta para ser una operación tan costosa. Además, hay una buena salida de sesenta transacciones por segundo. Como se ha visto previamente, MVC destaca por su velocidad.

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**1000**	**1**	**0.10%**	**4014.19**	**1**	**15900**	**2521.50**	**12169.30**	**14110.70**	**14730.96**	**60.03**	**32.36**	**26.41**
HTTP Request	1000	1	0.10%	4014.19	1	15900	2521.50	12169.30	14110.70	14730.96	60.03	32.36	26.41

Fig. 8.3.1.2.1 Estadísticas básicas resultados de Carga. *Reviews* - MVC

Mediante estas estadísticas se puede ver que recibe una cantidad de datos bastante grande en cuanto a bytes pos segundo se refiere, ya que cada respuesta contiene la lista de los cien *films*, con sus *reviews*.

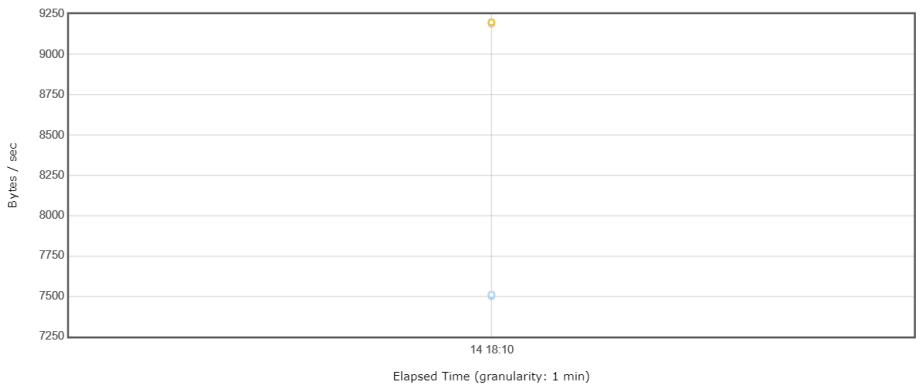


Fig. 8.3.1.2.2 Rendimiento de bytes a través del tiempo. *Reviews* - MVC

Aquí queda claramente reflejado que cuanto mayor es el tiempo de respuesta menor es el número de respuestas que se reciben. Lo cual quiere decir, que cuanto más tarde en responder más posibilidades de tener un cuello de botella tiene.

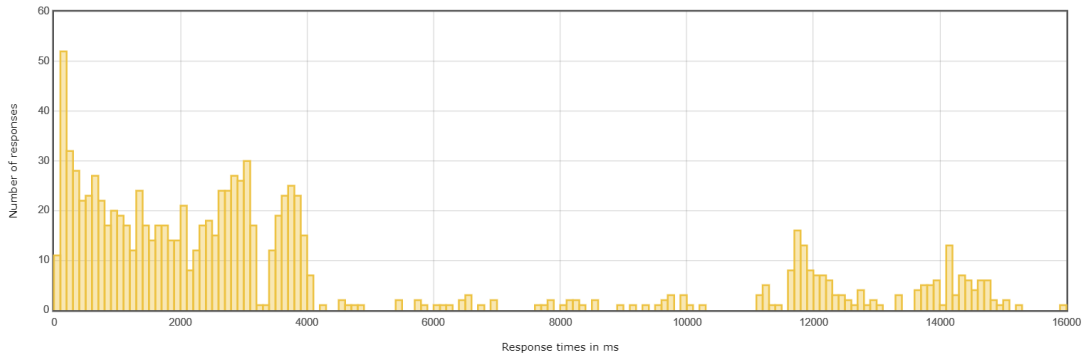


Fig. 8.3.1.2.3 Distribución del tiempo de respuesta. *Reviews* - MVC

8.3.1.3 Resultados Prueba de Carga: *Update Review* - Reactive

En este caso, nuevamente se vuelve a encontrar un tiempo de respuesta máximo muy superior a la solución imperativa, además el tiempo de respuesta medio sigue siendo bastante alto y las transacciones por segundo son la mitad que con MVC.

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	1000	0	0.00%	23866.37	380	30972	27349.50	29925.00	30217.95	30567.95	27.45	9.11	11.23
HTTP Request	1000	0	0.00%	23866.37	380	30972	27349.50	29925.00	30217.95	30567.95	27.45	9.11	11.23

Fig. 8.3.1.3.1 Estadísticas básicas resultados de Carga. *Reviews* - Reactive

Para este caso, en contraposición con la solución imperativa, y de acuerdo a lo estudiado a lo largo del proyecto. Contamos con que se reciben menor cantidad de bytes por segundo, debido a que al ser un flujo de datos constante recibe los *films* y sus *reviews* a medida que están disponibles y no se espera a que este la lista completa.

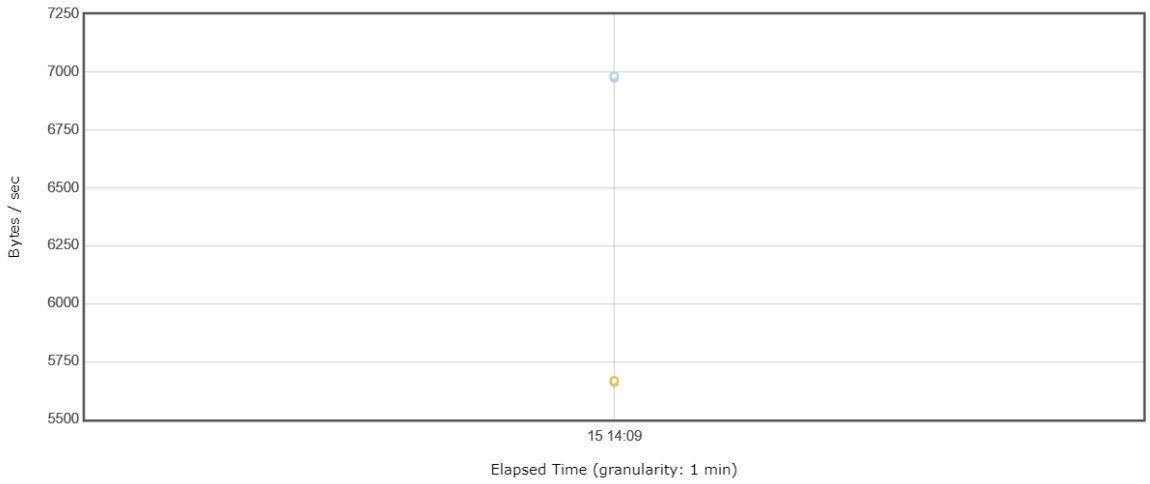


Fig. 8.3.1.3.2 Rendimiento de bytes a través del tiempo. *Reviews* - Reactive

Otra vez encontramos un caso opuesto a la solución imperativa, ya que en cuanto a los tiempos de respuesta se mantiene constante en cuanto a número de respuestas por tiempo de respuesta. Sin embargo, se puede ver que, cuantas más respuestas, mayores es el tiempo de respuesta, lo cual puede deberse a que no se recibe la respuesta completa hasta que el último *film* ha llegado.

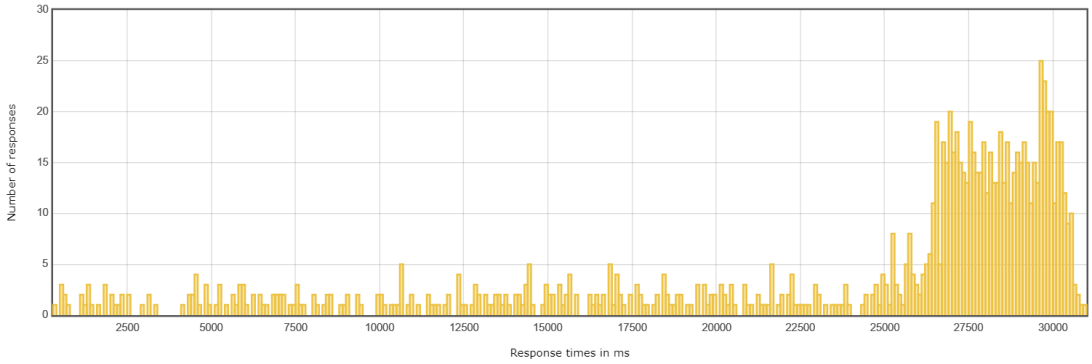


Fig. 8.3.1.3.3 Distribución del tiempo de respuesta. *Reviews* - Reactive

8.3.2 Pruebas de Estrés

8.3.2.1 Prueba de estrés: *Get One Review*

En esta primera prueba de estrés se ha llegado a un máximo de dos mil peticiones concurrentes. En este caso, cada petición recupera todos los *films* con sus *reviews* de un *user* y devuelve el primero de la lista.

8.3.2.1.1 Resultados Prueba de Estrés: *Get One Review* - MVC

Los primeros resultados de esta prueba muestran que el tiempo de respuesta máximo ha aumentado considerablemente respecto a las pruebas anteriores, debido al aumento de complejidad de la función llamada. Nuevamente hay un gran porcentaje de error, siendo menor que las otras pruebas y con más peticiones, pero hay que recalcar que en las otras pruebas se recuperaban mil *films* cada vez, y ahora cien.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
HTTP Request	10000	11746	0	30130	8455.58	13.58%	51.3/sec	45.18	16.83	902.5
TOTAL	10000	11746	0	30130	8455.58	13.58%	51.3/sec	45.18	16.83	902.5

Fig. 8.3.2.1.1 Resultados prueba de Estrés. *Get One Review* - MVC

8.3.2.1.2 Resultados Prueba de Estrés: *Get One Review* - Reactive

Respecto a la prueba usando programación reactiva, en este caso aumenta ligeramente el porcentaje de error respecto a las pruebas anteriores, ya que, al tener más lógica en el código, tiene mayor dificultado de realizar las operaciones usando *Publishers* y *Subscribers*. Sin embargo, sigue teniendo una gran diferencia con la solución imperativa. El tiempo de respuesta se mantiene alto igual que las otras pruebas, y las transacciones por segundo también sigue siendo baja.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
HTTP Request	10000	34538	75	62955	16330.10	2.17%	12.7/sec	5.97	4.34	480.1
TOTAL	10000	34538	75	62955	16330.10	2.17%	12.7/sec	5.97	4.34	480.1

Fig. 8.3.2.1.2 Resultados prueba de Estrés. *Get One Review* - Reactive

8.3.2.1.3 Comparativa Uso de Memoria y CPU: *Get One Review*

Como se puede ver en la aplicación reactiva, al recibir tanta información de forma tan seguida, el incremento de memoria es muy elevado. Por otro lado, utiliza una gran cantidad de porcentaje del procesador al inicio, tras recibir la sobrecarga de peticiones, después se mantiene bastante constante. En esta ocasión solo crea diez *threads* nuevos para un total de cuarenta y cinco.

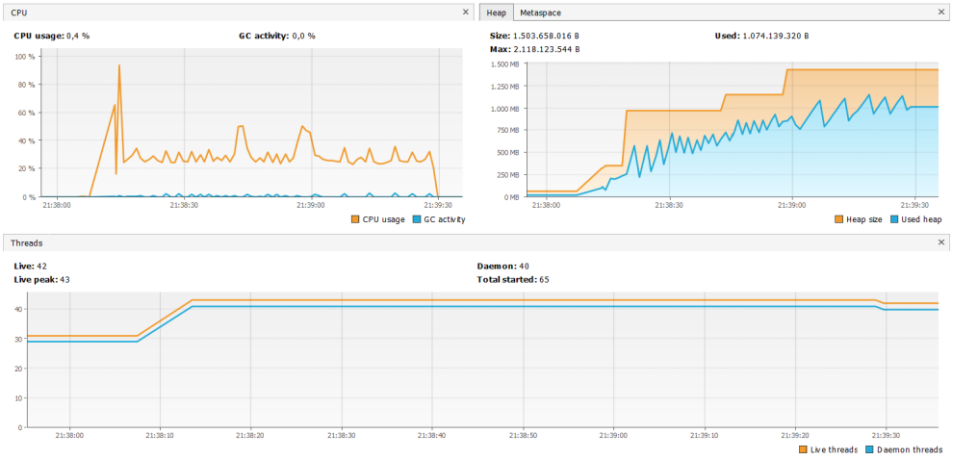


Fig. 8.3.2.1.3.1 Uso de Recursos. *Get One Review* - Reactive

En el caso de la aplicación imperativa, no utiliza una gran cantidad de memoria, pero si tiene alguna caída. Por otro lado, usa casi un ochenta por ciento de CPU, con una caída a 0 por el medio. Como es habitual en MVC, crea doscientos *threads*, hasta utilizar doscientos cincuenta.

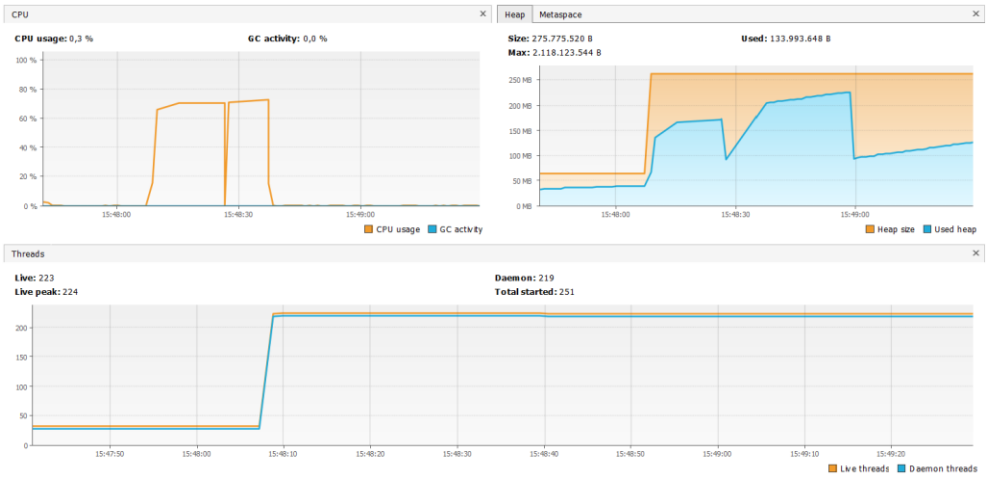


Fig. 8.3.2.1.3.2 Uso de Recursos. *Get One Review* - MVC

8.3.2.2 Prueba de Estrés: *Get All Films with Review*

Para esta prueba se realizan peticiones que devuelven todos los *films* de un *user*, como el anterior apartado son cien en total incluyendo sus *reviews*. Nuevamente se ha llegado al máximo de dos mil peticiones como máximo.

8.3.2.2.1 Resultados Prueba de Estrés: *Get All Films with Review* - MVC

En este caso, el tiempo de respuesta aumenta respecto a la prueba anterior ya que devuelve la lista entera de *films* con las *reviews*, para hacerlo más equitativo con R2DBC no se ha usado mapeo de entidades. Como se ve, tiene una buena salida de transacciones por segundo, aunque sigue dando muchos errores.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
HTTP Request	10000	13822	0	40162	9682.00	18.68%	63.2/sec	683.48	19.57	11079.5
TOTAL	10000	13822	0	40162	9682.00	18.68%	63.2/sec	683.48	19.57	11079.5

Fig. 8.3.2.2.1 Resultados prueba de Estrés. *Get All Films with Review* - MVC

8.3.2.2.2 Resultados Prueba de Estrés: *Get All Films with Review* - Reactive

Para la solución reactiva el tiempo de respuesta es el doble a la solución imperativa, como se ha visto a lo largo de las pruebas en la solución reactiva. Sin embargo, no se ha encontrado ningún error en todas las ejecuciones de esta prueba. También se puede ver la diferencia entre los datos recibidos por segundo de ambas soluciones, ya que este caso al enviar un flujo constante no envía la lista entera de *films*.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/...	Sent KB/sec	Avg. Bytes
HTTP Request	10000	62847	1118	93938	18472.52	0.00%	8.8/sec	111.06	2.99	12995.0
TOTAL	10000	62847	1118	93938	18472.52	0.00%	8.8/sec	111.06	2.99	12995.0

Fig. 8.3.2.2.2 Resultados prueba de Estrés. *Get All Films with Review* - Reactive

8.3.2.2.3 Comparativa Uso de Memoria y CPU: *Get All Films with Review*

De igual forma que la prueba anterior, mantiene un uso constante de la CPU excepto al principio. También utiliza mucha memoria dado que se reciben muchos datos y de forma constante. En este caso ha creado cuarenta *threads*, usando un total de setenta, que es el máximo de todas las pruebas realizadas.

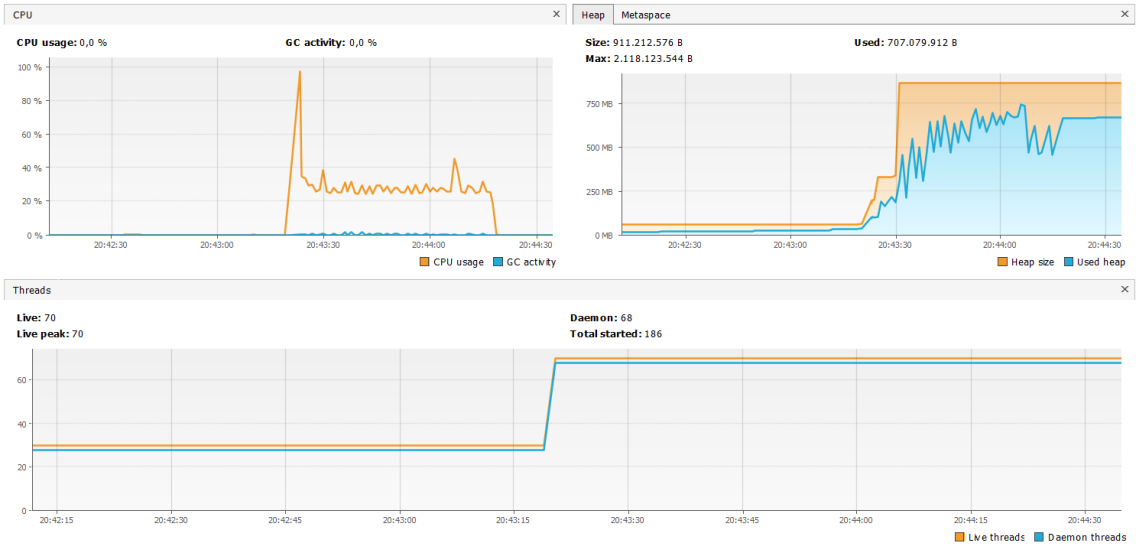


Fig. 8.3.2.2.3.1 Uso de Recursos. *Get All Films with Review* - Reactive

Para la solución tradicional, utiliza una gran parte de la CPU y de nuevo se encuentra una caída grave en medio de la prueba. En este caso tampoco ha hecho un gran uso de memoria. Nuevamente ha creado doscientos *threads* para usar un total de doscientos cincuenta.

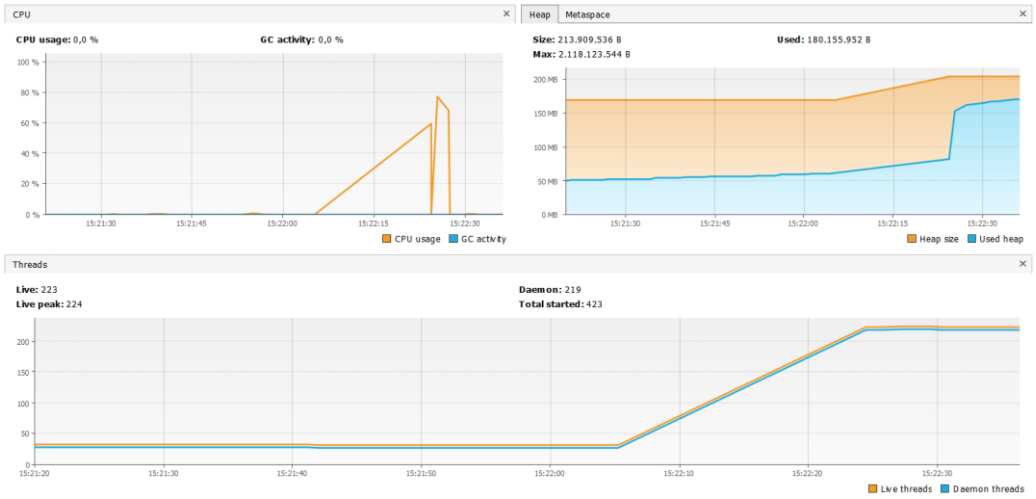


Fig. 8.3.2.2.3.2 Uso de Recursos. *Get All Films with Review* - MVC

9. Conclusiones

A lo largo del proyecto, se ha podido aprender sobre este paradigma de programación. No solo se ha estudiado sobre la programación reactiva en general, sino sobre el acceso a base de datos utilizando drivers no bloqueantes.

En cuanto al ámbito técnico, la programación reactiva puede ser más compleja que la programación tradicional, ya que este paradigma trata con tipos de datos como los *streams*, que pueden ser muy delicados y sobre todo si no se conoce sobre estas tecnologías. Por otra parte, al avanzar con las funcionalidades el margen de dificultad cada vez se hace más pequeño ya que vas aprendiendo y puedes intuir ciertas cosas que antes no lo haces, aunque ciertamente también aumenta la dificultad técnica. Sin embargo, la programación tradicional es bastante más simple de entender debido a que es la que se enseña desde el principio y a que los elementos que trata pueden no ser tan abstractos como la programación reactiva.

Por otro lado, en el acceso a base de datos, no hay un gran cambio entre los dos paradigmas, en cuanto a la configuración de los repositorios ni la configuración de las conexiones. Sin embargo, una desventaja que se ha encontrado en el driver de R2DBC es que no cuenta con mapeador de entidades, como si lo tendría con JDBC Template o las anotaciones de JPA.

En cuanto a los resultados de las pruebas realizadas hay varias conclusiones que se han sacado. Primero de todo está claro que en términos de velocidad a la hora de procesar las peticiones el driver bloqueante, en este caso JDBC, es bastante superior al driver de R2DBC. Sin embargo, en cuanto a consistencia y robustez la solución reactiva ha demostrado en todas las pruebas de carga ser muy superior a la solución imperativa. Esto quiere decir que la aplicación reactiva es capaz de soportar mayor carga de usuarios concurrentes. Cabe recalcar, que las pruebas donde más ha destacado la programación reactiva han sido aquellas que devolvían un flujo de datos ya que a un stream de datos se le envían varios objetos y se le saca mayor rendimiento. Por otro lado, las que devuelven un único objeto, el crear un stream de datos para cada petición tiene mayor coste y no genera tan buenos resultados.

Finalmente, en cuanto al uso de recursos, se ha visto que la aplicación reactiva puede mantener un uso de la CPU menor sin llegar a fallar, aunque es cierto que usa mucha memoria, es debido a que abre streams de datos y tiene un flujo de datos constante, por lo tanto, requiere más memoria, aunque solo ha sido en casos que exigen una gran cantidad de información. Por otro lado, la solución imperativa tiene un uso bastante menor de memoria, pero también bastante más alto y variante de CPU, lo cual puede hacer colapsar antes el sistema al no tener capacidad del procesador disponible. Además, cada ejecución de la solución imperativa ha llegado a utilizar alrededor de doscientos cincuenta *threads*, mientras que la solución reactiva ha rondado los cincuenta, siendo el máximo setenta.

Como conclusión final, se puede concluir que la solución reactiva consume los recursos del sistema de manera más eficiente. Por otro lado, la solución imperativa es mucho más rápido en cuanto a al procesar peticiones una por una, mientras que la solución reactiva destaca por su consistencia y robustez a la hora de tratar un gran número de peticiones concurrentes.

Si se quiere realizar una aplicación que se puede prever que no tenga un gran número de usuarios o peticiones concurrentes, como por ejemplo un sistema interno que solo utilizarían los empleados, entonces una aplicación usando programación imperativa es la mejor solución sin dudas.

En caso de que sea un sistema que reciba un gran número de peticiones a la vez, como puede ser un servicio de *streaming*, ahí es donde entrar a estudiar sobre la programación reactiva entrar es una buena opción ya que se puede crear este sistema usando un menor número de recursos, y escalar en conjunto al número de usuarios que la utilizan.

Una posible ampliación para este proyecto ya se había planteado al principio de este mismo. Trata de crear la parte del front-end de una aplicación utilizando la programación reactiva, para ver el comportamiento completo de una comunicación reactiva entre el front-end, la API con el back-end y las bases de datos.

10. Bibliografía

[1] " DIGITAL 2021: GLOBAL OVERVIEW REPORT ", https://datareportal.com

[2] "Estadísticas de suscriptores y crecimiento de Netflix: ¿Cuántas personas ven Netflix en 2021?" https://www.affde.com/ (Consulta 15/01/2022).

[3] Mohammed Nasiruddin, "Spring Web-Flux/Project Reactor", https://hmh.engineering, (Consulta 21/03/2022).

[4] Documentación Spring, *Web-Reactive*, Capitulo 1, Apartado 1.1.1 Párrafo 2.

[5] "The Reactive Manifesto" https://www.reactivemanifesto.org/es (Consulta 24/01/2022).

[6] "Ventajas y desventajas metodología Scrum." https://blog.wearedrew.co (Consulta 05/01/2022)

[7] *Reactive Spring* Josh Long, ISBN: 978-1732910416. Capítulo 10, Primer párrafo.

[8] Jay Phelps, "Explicación de la contrapresión: el flujo resistido de datos a través del software", Párrafo 9. https://medium.com/ (Consulta 22/03/2022)

[9] Documentación Spring, *Web-Reactive*, Capitulo 1, Apartado 1.1.1 Párrafo 4.

[10] "Programación reactiva asincrónica en Java usando Reactor Core" https://ichi.pro/es/programacion-reactiva (Consulta 23/03/2022)

[11] JavaTechie, "Spring Boot Reactive Programming Complete Tutorials for Beginners" https://youtu.be/bXcFCgQsvAE (Consulta 24/03/2022)

[12] Making Spring Data JDBC reactive, https://www.oreilly.com (Consulta 01/04/2022)

[13] Programación Reactiva con WebFlux y R2DBC, JoeDayzAcademy (Facebook) https://www.facebook.com/ (Consulta 21/04/2022)