UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE PARA LA WEB

Trabajo Fin de Máster

ESTUDIO DEL FRAMEWORK MICRONAUT PARA EL DESARROLLO DE MICROSERVICIOS REACTIVOS

Justin Hernández Jover

2022

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN

INGENIERÍA DEL SOFTWARE PARA LA WEB

Trabajo Fin de Máster

"ESTUDIO DEL FRAMEWORK MICRONAUT"

Autor: **Justin Hernández Jover**

Director: **Salvador Otón Tortosa**

Tribunal:

Presidente: .....

Vocal 1º: ..... Vocal 2º: .....

Calificación: .....

Fecha: ..... de ..... de .....

ÍNDICE RESUMIDO

ÍNDICE DETALLADO

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

La arquitectura orientada a microservicios se basa en la descomposición de un sistema en subsistemas más pequeños y aislados que se comunican mediante protocolos bien definidos. Son el contraste directo a la arquitectura tradicional de monolitos; aplicaciones cuyos módulos se encuentran estrechamente integrados y que debido a dicho acoplamiento son difíciles de modificar, escalar y mantener [1].

Para maximizar la eficiencia en una arquitectura orientada a microservicios, la comunicación entre ellos y el acceso a recursos externos como bases de datos debe basarse en la asincronía y mecanismos no bloqueantes. De esta forma se hace un uso mucho más eficiente de los recursos proporcionados a la aplicación, lo cual es especialmente importante en el ámbito de los microservicios, donde la escalabilidad, consumo energético y coste operativo son factores a tener en cuenta [2].

El *framework* Micronaut[1] surge como alternativa a otras herramientas disponibles en el ecosistema de la Máquina Virtual Java (del término anglosajón *Java Virtual Machine*, o JVM) para la creación de microservicios. A diferencia de otros *frameworks* más longevos como Spring[2], Micronaut ha sido desarrollado desde los cimientos con grandes mejoras orientadas a la creación de microservicios y aplicaciones sin servidor (del término anglosajón *Serverless Computing*), ofreciendo por defecto una serie de herramientas que soportan y facilitan la creación de microservicios reactivos [3].

[1] https://micronaut.io
[2] https://spring.io

2. OBJETIVOS DEL PROYECTO

El objetivo principal de este proyecto consiste en el estudio del framework Micronaut, enfocado principalmente a sus capacidades para la creación de microservicios reactivos. Para la consecución de este objetivo, se requiere alcanzar una serie de subobjetivos como siguen:

- Estudio del framework Micronaut y su posicion dentro del ecosistema Java, haciendo hincapié en las características enfocadas al desarrollo de microservicios que lo hacen destacar por encima de otros frameworks que ofrecen soluciones similares.
- Estudio de las herramientas y mecanismos disponibles y soportados por Micronaut para la creación de microservicios reactivos.
- Desarrollo del prototipo de una aplicación compuesta de microservicios reactivos desarrollados en Micronaut para demostrar las capacidades del framework mediante un caso práctico.

El estudio llevado a cabo en este proyecto y las conclusiones obtenidas servirán como guía a la hora de valorar Micronaut, en su versión más reciente disponible a la fecha de redacción de este documento, como herramienta de desarrollo para futuros proyectos basados en JVM cuya arquitectura y diseño justifiquen el uso de esta herramienta. Así mismo, existen otros aspectos de Micronaut que resultan especialmente interesantes y merecedores de atención [4][5], pero se encuentran fuera del ámbito de este proyecto y se identifican como líneas de investigación futuras.

3. ESTADO DEL ARTE

En este apartado se introducen los conceptos teóricos necesarios para la comprensión del trabajo. En primer lugar, se habla sobre la arquitectura orientada a microservicios. En segundo lugar, sobre el framework Micronaut y su posición dentro del ecosistema Java. Finalmente, sobre los sistemas reactivos y lo que significa que un microservicio sea reactivo.

3.1. Microservicios

La arquitectura basada en microservicios describe el enfoque usado a la hora de desarrollar una aplicación que consiste en la creación de múltiples servicios más pequeños e independientes que se comunican entre ellos usando protocolos ligeros, usualmente exponiendo APIs REST [3]. Cada microservicio aísla funciones comerciales específicas, siguiendo el Principio de Responsabilidad Única (del término anglosajón *Single Responsibility Principle*) [6], buscando la autonomía reduciendo el acoplamiento y la dependencia con otros microservicios al máximo. Para conseguir dicha autonomía, los microservicios no solo aíslan funciones comerciales específicas, sino que también se responsabilizan de sus datos siguiendo el patrón de una base de datos por servicio [2][3].

La arquitectura orientada a microservicios es el contraste directo a la arquitectura tradicional de monolitos; grandes y complejas aplicaciones cuyos módulos se encuentran estrechamente integrados y se despliegan como un solo ejecutable [1]. Debido a su estructura, nuevos cambios y actualizaciones requieren reconstruir y redesplegar todo el monolito, y si es necesario escalar el sistema, la aplicación en su totalidad es replicada en nuevos servidores (lo que se conoce como escalabilidad horizontal). Por otro lado, al ser los microservicios independientemente desplegables y escalables, modificaciones en la funcionalidad de la aplicación solo requiere reconstruir y redesplegar los microservicios afectados. Además, se pueden escalar individualmente las partes de la aplicación que soporten mayor carga, sin necesidad de replicar todo el sistema.

3.2. Micronaut

Micronaut es un framework de desarrollo software basado en la Máquina Virtual Java de código abierto publicado en mayo de 2018 por Object Computing Inc[3]. Fue desarrollado por los creadores de Grails[4], aprendiendo de las lecciones obtenidas durante años migrando aplicaciones monolíticas a microservicios usando Spring, Spring Boot y Grails [7].

[3] https://objectcomputing.com
[4] https://grails.org

Con la llegada de la arquitectura orientada a microservicios, las aplicaciones nativas en la nube (del término anglosajón *Cloud Native Applications*) y la computación sin servidor (del término anglosajón *Serverles Computing*), muchos frameworks tradicionales más especializados en aplicaciones monolíticas añadieron las modificaciones necesarias para afrontar estos nuevos paradigmas. Sin embargo, estos cambios no fueron suficientes, y las mismas estrategias utilizadas para el desarrollo de aplicaciones monolíticas fueron llevadas al mundo de los microservicios, produciendo aplicaciones con mayores tiempos de arranque y consumo de memoria.

Micronaut fue desarrollado desde los cimientos para el desarrollo de microservicios, con el objetivo de reducir al máximo el consumo de memoria y los tiempos de arranque de las aplicaciones. Esto lo consigue mediante lo que se conoce como compilación anticipada (del término anglosajón *Ahead Of Time Compilation*) al realizar inyección de dependencias (del término anglosajón *Dependency Injection*, DI) [8], gestión de configuración [9] y generación de las clases resultantes (también conocidas como proxys) en la programación orientada a aspectos (del término anglosajón *Aspect Oriented Programming*, AOP) [10]. Micronaut procesa las anotaciones presentes en el código (por ejemplo @Override) para generar metadatos en tiempo de compilación en forma de *bytecode* procesable por la JVM, que además son posteriormente optimizados por el compilador JIT (del término anglosajón *Just In Time*). Frameworks como Spring hacen uso de otro mecanismo conocido como reflexión (del término anglosajón *reflection*) [11] a la hora de procesar las anotaciones, donde se escanean todos los *classpath* existentes para generar los metadatos al iniciar la aplicación, aumentando así el tiempo de arranque y el consumo de memoria.

Los resultados de aplicar estas estrategias para optimizar los tiempos de arranque y el consumo de memoria de la aplicación se traducen en despliegues casi un 50% más rápidos y un consumo de memoria 40% menor comparado con la misma aplicación desarrollada en Spring Boot [3][12].

3.3. Sistemas reactivos

Los requisitos de las aplicaciones han ido evolucionando con el tiempo para adaptarse a las demandas y necesidades de los usuarios con mejores tiempos de respuesta, mayor disponibilidad y un manejo de volumen de datos creciente. Los sistemas reactivos, como se definen en el manifiesto [13], deben cumplir una serie de características para cumplir las demandas de los sistemas actuales, que no son cubiertas por arquitecturas de software tradicionales. Los sistemas reactivos deben ser:

- Sensibles (del término anglosajón *Responsive*): Los sistemas deben gestionar peticiones lo antes posible, en ventanas de tiempo razonables. La rapidez con la que responde el sistema influye en gran medida a su usabilidad y utilidad general, además de facilitar la detección y gestión de errores. La sensibilidad del sistema es importante a la hora de proporcionar una calidad de servicio consistente.
- Resistente (del término anglosajón *Resilient*): Los sistemas se mantienen disponibles y sensibles en caso de fallos, entendiendo fallo como un evento que no puede ser controlado por el

desarrollador y es, por tanto, inesperado (fallos de hardware, sistema operativo) [14]. Un sistema no puede ser sensible si no es capaz de recuperarse después de un fallo. La resistencia de un sistema se consigue mediante mecanismos de replicación, delegación y aislamiento [14]. De esta forma, un fallo no se propaga por todo el sistema, sino que se contiene en el componente donde ocurrió y gracias a la replicación, otra instancia del componente puede atender a las peticiones entrantes sin que el sistema comprometa su sensibilidad.

- Elástico: El sistema se mantiene sensible bajo cargas de trabajo variables, aumentando o disminuyendo la cantidad de recursos que consume para satisfacer la demanda. Un sistema reactivo debe soportar algoritmos predictivos de escalado, además de reactivos, alimentados por métricas relevantes de consumo reportadas en vivo.
- Basado en intercambio de mensajes (del término anglosajón *Message-Driven*): Los sistemas reactivos se basan en el intercambio de mensajes (del término anglosajón *Message-Passing*) de manera asíncrona entre los distintos componentes. En este sistema los componentes destinatarios se encuentran en espera hasta la llegada de los mensajes y según su disponibilidad, reaccionan a ellos. Este mecanismo permite gestionar la carga y elasticidad del sistema, además de controlar del flujo al monitorear las colas de mensajes, aplicando lo que se conoce como *back-pressure* [14] según sea necesario. *Back-Pressure* es un mecanismo que monitoriza los signos vitales de los diversos componentes del sistema y si detecta que alguno se encuentra abrumado por la carga de trabajo, reduce el flujo de mensajes hasta que el componente sea capaz de volver a un estado normal. Este mecanismo contribuye a la resistencia y elasticidad general del sistema.

En la Figura 1 se muestran las características propias de los sistemas reactivos y como interaccionan y se refuerzan entre ellas. La resistencia y elasticidad del sistema contribuyen a la sensibilidad de este ya que se encargan de, por un lado, protegerlo antes fallos aislando los componentes afectados, y por otro lado escalando o disminuyendo los recursos en base a la demanda. Al basarse en intercambio de mensajes, se pueden monitorear las colas para predecir el nivel de escalado necesario, se aumenta la robustez general del sistema y al usar comunicación no bloqueante el mismo es más sensible. La elasticidad del sistema contribuye por defecto a la resistencia de este y viceversa.

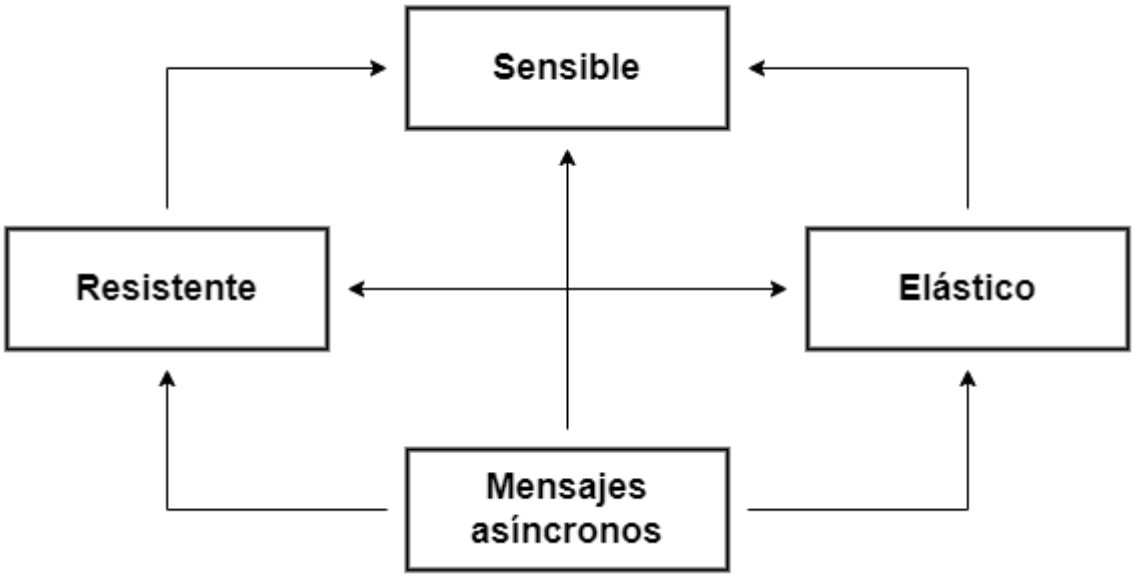

Figura 1: Relación entre las características de los sistemas reactivos

3.4. Microservicios reactivos

En base a las características que definen un sistema reactivo expuestas en el apartado 3.3, se puede concluir que los microservicios reactivos deben ser sensibles, resistentes, elásticos y deben basar su comunicación en el intercambio de mensajes. Aunque factores como la resistencia ante fallos y la elasticidad (capacidad de escalar acordemente bajo demanda) sean gestionados en gran parte por las plataformas donde se desplieguen los microservicios (por ejemplo, Kubernetes[5]), es importante que la implementación a nivel de aplicación de cada uno de los microservicios se base en conceptos reactivos para garantizar que el sistema se mantenga sensible y se utilicen los recursos disponibles de la forma más eficiente posible.

A nivel de aplicación, los microservicios reactivos deben mantenerse disponibles la mayor cantidad de tiempo posible para gestionar peticiones haciendo un uso eficiente de los recursos proporcionados. Los mecanismos y herramientas reactivas disponibles en Micronaut, comentadas en la sección 4, permiten desarrollar microservicios que cumplen dicho objetivo.

3.5. Flujos reactivos en Java

La especificación de flujos reactivos para la JVM (del término anglosajón *Reactive Streams*) es una iniciativa que proporciona un estándar para el procesamiento de flujos asíncronos de forma no bloqueante y con *back-pressure* [14][21]. Esta especificación surge con el objetivo de que las implementaciones desarrolladas siguiendo dicho estándar sean capaces de interoperar entre ellas con fluidez y sin problemas.

[5] https://kubernetes.io

La especificación de flujos reactivos incluye definición de los distintos componentes que deben ser proporcionados por toda implementación de la especificación:

1. Publicador (*Publisher*):

Un publicador es un proveedor de un numero potencialmente ilimitado de elementos secuenciados, los cuales publica según la demanda recibida de sus suscriptores. La interfaz de un publicador se puede definir de la siguiente forma:

```
1 public interface Publisher<T> {
2         public void subscribe(Subscriber<? Super T> s)
3 }
```

Una invocación a *subscribe()* funciona como una petición de un Suscriptor a un Publicador para establecer una relación y posteriormente comenzar a transmitir datos. Cada invocación a este método genera una nueva Suscripción, y cada Suscripción funciona para un único Suscriptor.

2. Suscriptor (*Subscriber*):

El Suscriptor decide cuando y cuantas señales del Publicador recibe (*back-pressure*). Un Suscriptor solo puede tener una Suscripción activa a un Publicador en un momento dado. Su interfaz se puede definir de la siguiente forma:

```
1 public interface Subscriber<T> {
2     public void onSubscribe(Subscription s);
3     public void onNext(T t);
4     public void onError(Throwable t);
5     public void onComplete();
6 }
```

El método *onSubscribe()* se invoca cuando una instancia de Suscriptor se pasa a *Publisher.subscribe()*. Los datos solo comenzarán a fluir una vez *Suscripcion.request(Long n)* ha sido invocado, y con *onNext()* se definen las operaciones a ejecutar sobre cada elemento individual. De esta forma el Suscriptor controla la cantidad de señales que puede procesar ya que, independientemente del número de señales que el Publicador tenga disponible, pide las que pueda procesar por lotes de cantidad *n* con *Suscription.request(Long n)*. Los dos últimos métodos controlan el final del flujo, ya sea debido a un error con *onError()*, o correctamente con *onComplete()*. Una invocación de alguno de estos dos métodos anula la Suscripción de manera que el Suscriptor no puede pedir nuevos lotes.

3. Suscripción (*Subscription*):

Una Suscripción representa la relación entre un único Publicador y Suscriptor. El Suscriptor obtiene la referencia a la Suscripción creada al invocar *Publicador.subscribe(Suscriptor s)* y tiene el control sobre cuándo y cuantos elementos son pedidos y cuando dejan de ser necesarios. La interfaz de Suscripción se puede definir de la siguiente manera:

```
1 public interface Subscription {
2     public void request(Long n);
3     public void cancel();
4 }
```

El método *request(Long n)* es invocado por el Suscriptor para informar de que puede procesar *n* elementos. El método *cancel()* es invocado por el Suscriptor en caso de que desee finalizar el flujo y anular la Suscripción. Si se invoca *cancel()* pero no se han acabado de enviar los elementos de previas invocaciones a *request()*,estos son entregados y posteriormente se anula la Suscripción.

4. Procesador (*Processor*):

Un Procesador representa una etapa de procesamiento que es tanto un Suscriptor como un Publicador. A la hora de realizar transformaciones sobre el flujo de datos, como por ejemplo una función *map()* o *filter()*, es necesario implementar una o mas clases de tipo Procesador. Se puede definir la interfaz Procesador de la siguiente forma:

```
1 public interface Processor<T, R> extends Subscriber<T>, Publisher<R> {
2 }
```

El objetivo principal de los flujos reactivos es controlar el intercambio de datos de un flujo a través de limites asíncronos (del término anglosajón *asynchronous boundaries*), como, por ejemplo, pasar elementos de un flujo de un hilo a otro, asegurando que el lado receptor no se ve obligado a procesar mayor cantidad de datos de la que puede/quiere. Es importante que, en caso de que ejecuten tareas bloqueantes, el procesamiento de los elementos en la función *Suscriptor.onNext()*, y el procesamiento de las señales de terminación *Suscriptor.onError()* y *Suscriptor.onComplete()* se hagan de manera asíncrona en otro hilo de manera que no bloqueen el Publicador. Los Procesadores, en cambio, pueden realizar sus tareas de manera síncrona o asíncrona. En los aparatados 4.3 y 4.4 se describen implementaciones concretas de la especificación de flujos reactivos en Java disponibles en Micronaut.

4. EVALUACIÓN DE HERRAMIENTAS REACTIVAS EN MICRONAUT

En este apartado se explican las distintas herramientas disponibles en Micronaut para el desarrollo de microservicios reactivos. Las siguientes secciones explican cómo se puede conseguir reactividad en cada una de las capas de desarrollo de los microservicios, empezando por la capa de transporte, seguido por la implementación de la lógica de negocio, hasta el acceso a base de datos.

4.1. Cliente y servidor HTTP basado en Netty

La arquitectura orientada a microservicios basa su funcionamiento en la comunicación entre cada uno de los componentes del sistema. Por este motivo es importante la gestión eficiente de recursos en la implementación de los protocolos de comunicación entre microservicios, siendo el más común el protocolo HTTP, intercambiando datos en formato JSON (del término anglosajón *JavaScript Object Notation*). Esto significa que el modelo tradicional de usar un hilo (del término anglosajón *thread*) para gestionar cada una de las peticiones HTTP recibidas, común en los servidores web tradicionales, es inadecuado. Si para resolver la petición es necesario realizar alguna tarea de tipo Entrada/Salida (del término anglosajón *Input/Output*, IO) como realizar otra petición HTTP a un servicio externo o leer un archivo del sistema de archivos, dicho hilo se bloquea hasta que la petición se complete, desperdiciando tiempo que puede dedicar a realiza otras tareas. En la Figura 2 se muestra un ejemplo del comportamiento de un hilo al realizar una tarea bloqueante de tipo entrada/salida mientras espera que el kernel del sistema operativo gestione dicha operación a bajo nivel.

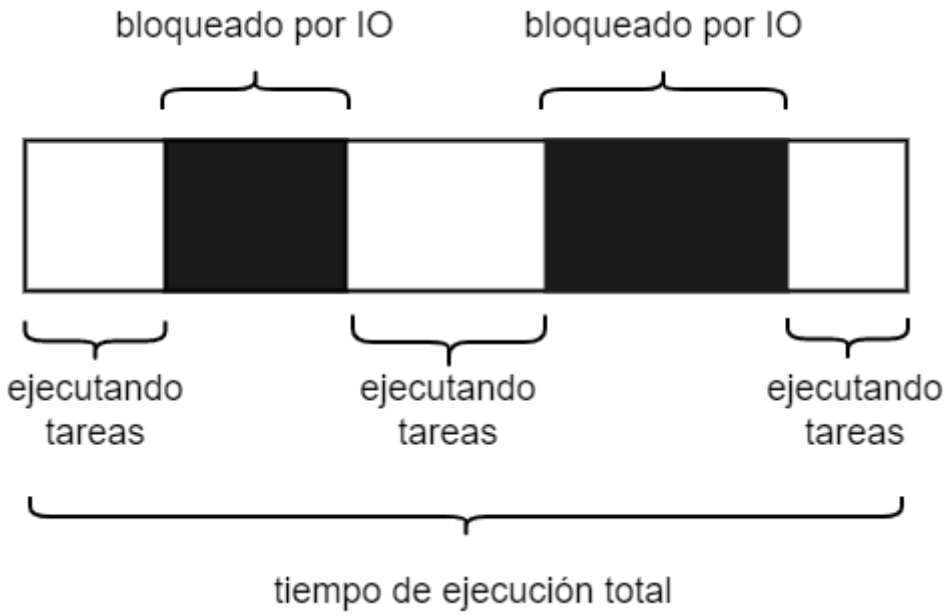

Figura 2. Hilo procesando petición HTTP bloqueante de manera secuencial.

Micronaut implementa un cliente y servidor HTTP basado en Netty [15]. Netty[6] es un framework que hace uso de la API NIO[7] (del término anglosajón *New Input Output*) disponible en Java para el desarrollo de aplicaciones de red, como clientes y servidores HTTP (o cualquier otro tipo de protocolo), enfocado en el alto rendimiento y la asincronía. El diseño e implementación del cliente y servidor HTTP en Micronaut están optimizados para el intercambio de mensajes entre microservicios en formato JSON, usando la biblioteca Jackson para la serialización/deserialización de objetos de forma asíncrona [16].

Para conseguir procesar mayor número de peticiones con un menor número de hilos, Netty hace uso de un modelo denominado bucle de eventos (del término anglosajón *Event-Loop*). En la Figura 3 se muestra el funcionamiento, donde en vez de bloquear un hilo cuando ocurre una operación de tipo entrada-salida bloqueante, dicho hilo pasa a ejecutar otras tareas disponibles hasta que la inicial vuelva a estar lista para continuar su procesamiento.

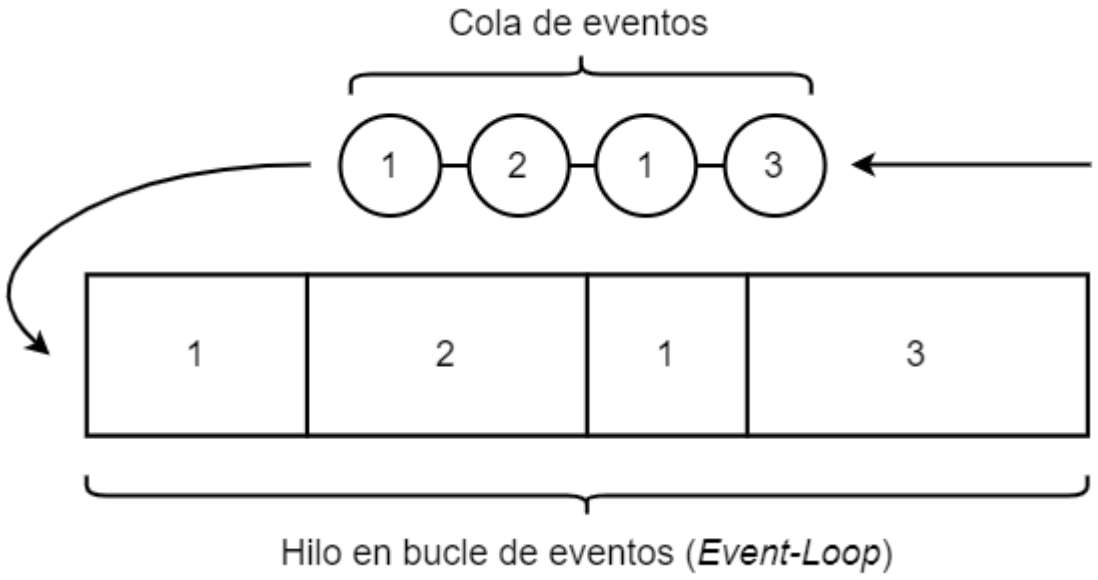

Figura 3. Hilo en bucle de eventos procesando tareas.

El código que se ejecuta en los bucles de eventos no debe realizar tareas bloqueantes o de larga duración, ya que afectarían a la sensibilidad del sistema (véase la sección 3.3). Si la aplicación necesita realizar procesamiento de este tipo, es posible delegar la ejecución de dichas tareas a lo que se conoce como hilos de trabajo (del término anglosajón *worker threads*) separados del hilo que ejecuta el bucle de eventos. Estos hilos de trabajo se encuentran en lo que se denominan piscinas de hilos (del término anglosajón *thread pools*), listos para realizar tareas sin necesidad de ser instanciados cada vez para no penalizar el rendimiento. Cuando el hilo de trabajo finalice la ejecución de su tarea bloqueante, inserta la continuación del evento (función *callback*) en la cola de eventos, como se muestra en la Figura 4.

[6] https://netty.io/index.html
[7] https://docs.oracle.com/en/java/javase/15/core/java-nio.html

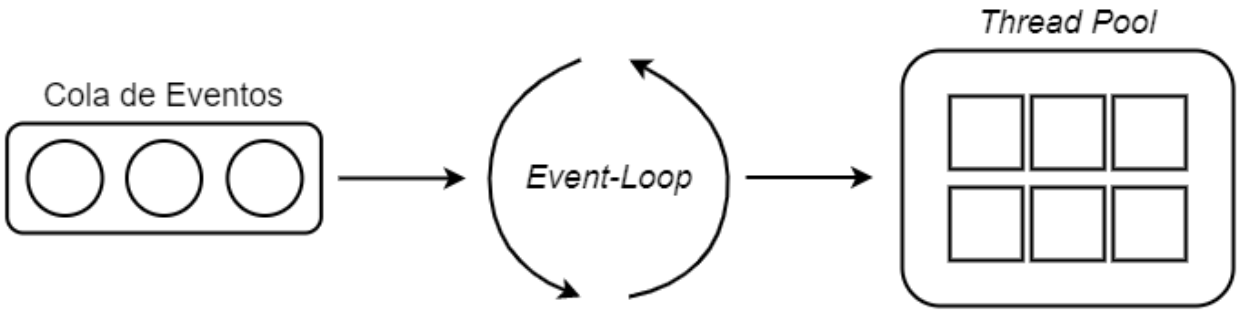

Figura 4. Procesamiento de tareas bloqueantes en un bucle de eventos.

Al definir un controlador en Micronaut, el código de cada uno de los métodos es ejecutado en el hilo del bucle de eventos. Si la operación llevada a cabo dentro de cada método requiere de mucho procesamiento o es bloqueante (por ejemplo, acceso a base de datos mediante Hibernate/JPA no reactivo), es importante que se delegue la ejecución a un hilo dentro de una piscina de hilos para no bloquear el bucle de eventos. Micronaut permite configurar piscinas de hilos (llamados *executors*) en el archivo de configuración *application.yml*, de la siguiente forma:

```
micronaut:
  executors:
    tareas_io:
      type: fixed
      nThreads: 20
```

Posteriormente, en la definición del controlador, o a nivel de método, se hace uso de la anotación *@ExecuteOn()* para indicar el nombre de la piscina de hilos a utilizar al procesar las peticiones, ejemplo:

```
@Get("/peliculas")
@ExecuteOn(TaskExecutors.tareas_io)
Pelicula getAllPeliculas() {
    // realizar tarea bloqueante
}
```

Micronaut proporciona otros mecanismos a la hora de ejecutar tareas de manera asíncrona fuera del hilo del bucle de eventos en los controladores. Estas alternativas se describen en los apartados 4.3 y 4.4.

4.2. Acceso asíncrono a base de datos

Los drivers de base de datos como JDBC son síncronos por defecto, lo que significa que el hilo donde se ejecutan las operaciones se bloquea mientras espera el resultado por parte del driver. Herramientas como Hibernate/JPA sufren de esta limitación, aún cuando se use el modelo de piscinas de conexiones activas a

la base de datos (del término anglosajón *connection pools*) que se rehúsan por cada petición según disponibilidad. La solución se encuentra en la implementación de la reactividad a nivel de driver de base de datos. En las siguientes secciones se explican algunas de las opciones más relevantes que ofrece Micronaut a la hora de interactuar con bases de datos tanto SQL como NoSQL de forma no bloqueante y asíncrona.

4.2.1. Micronaut Data MongoDB

Micronaut proporciona compatibilidad con la especificación de flujos reactivos (del término anglosajón *Reactive Streams* [21]) de Java para MongoDB [17]. Este driver implementa la especificación de flujos reactivos en Java proporcionando procesado asíncrono no bloqueante con *back pressure* [14] para MongoDB. Además, el módulo Micronaut Data ofrece compatibilidad específica con MongoDB, de manera que se puede configurar la capa de persistencia de la aplicación de una forma muy similar a como se haría con una base de datos relacional con JPA [18].

Para hacer uso de MongoDB con el driver reactivo y Micronaut Data, se gestionan las dependencias y se incluye en application.yml la configuración del servidor de MonogDB:

```
1 mongodb:
2     uri: mongodb://username:password@localhost:27017/dbName
```

Posteriormente, se definen las clases entidad de la siguiente forma, usando la anotación *@MappedEntity* de manera que se marca la entidad a ser persistida:

```
1 @MappedEntity
2 class Pojo {
3     @Id
4     @GenerateValue
5     private ObjectId id;
6     // otras propiedades, getters y setters
7 }
```

En la capa del repositorio, se crea la interfaz extendiendo *ReactiveStreamsCrudRepository* para gestionar el acceso asíncrono a la base de datos. Finalmente se marca la interfaz con la anotación *@MongoRepository*:

```
1 @MongoRepository
2 interface PojoRepository extends ReactiveStreamsCrudRepository <Pojo,
3 ObjectId> {
4     // definir operaciones especificas
5     // por defecto ya implementa operaciones CRUD
  }
```

Se crea la interfaz del servicio y su implementación, atendiendo al tipo de dato devuelto:

```
interface PojoService {
    Publisher<Pojo> list();
    Publisher<Pojo> save(Pojo pojo);
    // resto de operaciones que se deseen
}

@Sigleton
class PojoServiceImpl implements PojoService {

    private final PojoRepository pojoRepository;

    public Publisher<Pojo> list() {
        return pojoRepository.findAll();
    }

    public Publisher<Pojo> save(Pojo pojo) {
        return pojoRepository.save(pojo);
    }

    // resto de operaciones a implementar del servicio
}
```

Finalmente, el controlador se puede definir de la siguiente manera:

```
@Controller("/")
Class pojoController {

    Private final PojoService pojoService;

    @Get("/pojos")
    Publisher<Pojo> list() {
        return pojoService.list();
    }

    @Post("/pojos")
    Publisher<Pojo> save(@Valid @NotNull Pojo pojo) {
        return pojoService.save(pojo)
    }

    // resto de metodos
}
```

El tipo de dato Publisher<T> forma parte de la especificación de flujos reactivos de Java (véase la sección 3.5). Micronaut soporta la devolución de diversos tipos reactivos en los controladores, siendo Publisher<T> uno de ellos. Cuando Micronaut devuelve un tipo reactivo en un controlador, el hilo del

bucle de eventos que gestiona la petición HTTP se suscribe al tipo reactivo devuelto. Una vez la clase del repositorio reactivo ha gestionado el acceso a la base de datos, y ha resuelto la invocación de uno de sus métodos por parte del servicio, por ejemplo, *findAll()*, el hilo del bucle de eventos devuelve el resultado. De esta forma nunca se bloquea el bucle de eventos, y la reactividad se encuentra en el driver de MongoDB.

4.2.2. Micronaut Data Hibernate Reactive

Hibernate Reactive es una API reactiva para el mapeador objeto-relacional Hibernate (del término anglosajón *Object-Relational Mapper*) que soporta drivers reactivos para interactuar con bases de datos relacionales de manera no bloqueante. Hibernate, JPA y JDBC hacen uso de operaciones de tipo Entrada/Salida bloqueantes a la hora de interactuar con la base de datos, por tanto, su uso no es adecuado en un entorno reactivo. Micronaut Data se combina con Hibernate Reactive para proporcionar las características y mecanismos disponibles en JPA, pero de forma reactiva. Hibernate Reactive hace uso de los drivers reactivos creados por el proyecto Vertx[8], de forma que se debe elegir el adecuado según la base de datos a utilizar. Actualmente se soportan drivers para MySQL, PostgreSQL, Microsoft SQLServer y Oracle [19].

Una vez gestionadas las dependencias de Micronaut Data y el driver reactivo a utilizar, para hacer uso de Hibernate Reactive se debe configurar JPA en *application.yml* de forma tradicional, pero activando la propiedad "reactive":

```
jpa:
  default:
    reactive: true
    properties:
      hibernate:
        connection:
          url: jdbc:postgresql:database
          username: myUsername
          password: myPassword
```

Posteriormente, se configuran las clases entidad de forma tradicional de la siguiente forma:

```
@Entity
@Table(name="Pojos")
class Pojo {
    @Id
    @GenerateValue (strategy = GenerationType.IDENTITY)
    private Long id;

    // otras propiedades, getters y setters
}
```

[8] https://vertx.io

En la capa del repositorio, la interfaz o clase abstracta creada debe extender alguna de las interfaces disponibles, mostradas en la Tabla 1, que soportan distintos tipos reactivos.

Interfaz	Descripción
ReactiveStreamsCrudRepository	Extiende GenericRepository y añade métodos CRUD que devuelven el tipo Publisher<T> de la API de flujos reactivos de Java.
ReactorCrudRepository	Extiende ReactiveStreamsCrudRepository y hace uso de los tipos reactivos de la biblioteca Project Reactor (véase la sección 4.3).
RxJavaCrudRepository	Extiende GenericRepository y hace uso de los tipos reactivos de la biblioteca RxJava (véase la sección 4.4).
CoroutineCrudRepository	Extiende GenericRepository y hace uso de las corrutinas de Kotlin para conseguir reactividad.
ReactiveStreamsJpaSpecificationExecutor	Representa la versión de flujos reactivo de JpaSpecificationExecutor
ReactorJpaSpecificationExecutor	Representa la versión de JpaSpecificationExecutor haciendo uso de los tipos reactivos disponibles en la biblioteca Reactor (véase la sección 4.3).

Tabla 1.Interfaces de repositorio reactivas.

```
1 @Repository
2 interface PojoRepository extends ReactorCrudRepository <Pojo, Long> {
3     // definir operaciones especificas
4     // por defecto ya implementa operaciones CRUD
5 }
```

Se crea la interfaz del servicio y su implementación, atendiendo al tipo de dato devuelto:

```
1 interface PojoService {
2     Flux<Pojo> list();
3     Mono<Pojo> save(Pojo pojo);
4     // resto de operaciones que se deseen
5 }
6
```

```
@Sigleton
class PojoServiceImpl implements PojoService {

    private final PojoRepository pojoRepository;

    public Flux<Pojo> list() {
        return pojoRepository.findAll();
    }

    public Mono<Pojo> save(Pojo pojo) {
        return pojoRepository.save(pojo);
    }

    // resto de operaciones a implementar del servicio
}
```

Finalmente, el controlador se puede definir de la siguiente manera:

```
@Controller("/")
class pojoController {

    private final PojoService pojoService;

    @Get("/pojos")
    Flux<Pojo> list() {
          return pojoService.list();
    }

    @Post("/pojos")
    Mono<Pojo> save(@Valid Pojo pojo) {
          return pojoService.save(pojo)
    }

    // resto de metodos
}
```

El tipo Mono<T> y Flux<T> pertenecen a la biblioteca Project Reactor y se explican en detalle en la sección 4.3. Con Micronaut Data y Hibernate Reactive se consigue acceso reactivo a base de datos con todas las ventajas que ofrece JPA a la hora de definir las clases de entidad y las relaciones. Además, Micronaut ofrece una serie de interfaces repositorio que soportan las bibliotecas y tipos reactivos más comunes, de manera que el desarrollador no esta limitado en opciones. Como se explica en la sección 4.2.1, Micronaut gestiona los tipos reactivos en los métodos de sus controladores de manera que no se bloquea el hilo que gestiona el bucle de eventos, ya que el acceso a base de datos se realiza a través de drivers reactivos con Entrada/Salida no bloqueante.

4.2.3. Micronaut Data R2DBC

Si se desea interactuar a bajo nivel con una base de datos en Java, a diferencia de usar un ORM como Hibernate, lo más común es utilizar el driver y la API JDBC (del término anglosajón *Java Database Connectivity*), sin embargo, en un entorno reactivo, esta no es una opción viable ya que JDBC es bloqueante y síncrono. R2DBC (del término anglosajón *Reactive Relational Database Connectivity)* es un proyecto que define una API, basada en la especificación de flujos reactivos [21], para proporcionar acceso reactivo a bases de datos [20]. R2DBC no es una implementación, sino una especificación a seguir a la hora de implementar drivers reactivos para bases de datos SQL. Los drivers que siguen esta especificación implementan el acceso a base de datos sobre una capa de Entrada/Salida asíncrona y no-bloqueante. Actualmente se encuentran disponibles drivers R2DBC para MySQL, PostgreSQL, H2, MariaDB, SQL Server y Oracle [22].

Una vez gestionadas las dependencias, es necesario incluir en el archivo *application.yml* la configuración para conectarse a la base de datos de la siguiente forma:

```
r2dbc:
  datasources:
    default:
      url: r2dbc:postgres:///database
```

Al usar Micronaut Data junto con R2DBC se puede hacer uso de las anotaciones propias de Micronaut Data para facilitar la creación de las clases de entidad. Con la anotación *@MappedEntity* se indica que la entidad va a ser persistida en base de datos:

```
@MappedEntity
class Pojo {

    @Id
    @GenerateValue
    private Long id;

    // otras propiedades, getters y setters
}
```

En la capa del repositorio, la interfaz o clase abstracta creada debe extender alguna de las interfaces recomendadas para ser usadas junto con drivers R2DBC, mostradas en la Tabla 2. Además, se debe prestar atención al dialecto utilizado, que varía en función de la base de datos a utilizar.

Interfaz	Descripción

ReactiveStreamsCrudRepository	Extiende GenericRepository y añade métodos CRUD que devuelven el tipo Publisher<T> de la API de flujos reactivos de Java.
ReactorCrudRepository	Extiende ReactiveStreamsCrudRepository y hace uso de los tipos reactivos de la biblioteca Reactor (véase la sección 4.3).
RxJavaCrudRepository	Extiende GenericRepository y hace uso de los tipos reactivos de la biblioteca RxJava (véase la sección 4.4).
CoroutineCrudRepository	Extiende GenericRepository y hace uso de las corrutinas de Kotlin para conseguir reactividad.

Tabla 2. Interfaces de repositorio reactivas recomendadas para R2DBC.

```
@R2dbcRepository(dialect = Dialect.POSTGRES)
interface PojoRepository extends ReactiveStreamsCrudRepository <Pojo, Long>
{
    // definir operaciones específicas
    // por defecto ya implementa operaciones CRUD
}
```

Se crea la interfaz del servicio y su implementación, atendiendo al tipo de dato devuelto:

```
interface PojoService {
    Flux<Pojo> list();
    Mono<Pojo> save(Pojo pojo);
    // resto de operaciones que se deseen
}

@Sigleton
class PojoServiceImpl implements PojoService {

    private final PojoRepository pojoRepository;

    public Flux<Pojo> list() {
        return pojoRepository.findAll();
    }

    public Mono<Pojo> save(Pojo pojo) {
        return pojoRepository.save(pojo);
    }

    // resto de operaciones a implementar del servicio
}
```

Finalmente, el controlador se puede definir de la siguiente manera:

```
1 @Controller("/")
2 class pojoController {
3
4     private final PojoService pojoService;
5
6     @Get("/pojos")
7     Flux<Pojo> list() {
8         return pojoService.list();
9     }
10
11    @Post("/pojos")
12    Mono<Pojo> save(@Valid @NotNull Pojo pojo) {
13        return pojoService.save(pojo)
14    }
15
16    // resto de metodos
17 }
```

Con Micronaut Data y drivers que implementen la especificación R2DBC, se consigue interactuar directamente con una base de datos relacional a bajo nivel de forma asíncrona y no bloqueante. Como se ha explicado en secciones anteriores, Micronaut gestiona los tipos reactivos en los métodos de sus controladores de manera que no se bloquea el hilo que gestiona el bucle de eventos, ya que el acceso a base de datos se realiza a través de drivers reactivos con Entrada/Salida no bloqueante.

4.3. Project Reactor

Project Reactor[9] es una biblioteca reactiva que implementa la especificación de flujos reactivos (véase la sección 3.5) cuyo objetivo es proporcionar las herramientas necesarias para el desarrollo de aplicaciones asíncronas, no bloqueantes de todo tipo en la JVM. Project Reactor ofrece una serie de ventajas comparado con los modelos que Java ofrece por defecto a la hora de conseguir programación asíncrona con los *Callbacks* y *Futures* [23]:

- Capacidad de encadenar y orquestar múltiples tareas asíncronas de forma mucho más legible con menos código, lo que contribuye positivamente al mantenimiento de la base de código de la aplicación.
- La forma de interactuar con las operaciones y los datos se asemeja a la especificación de los flujos reactivos y al paradigma de programación reactiva en general.

[9] https://projectreactor.io

- Implementa un amplio abanico de operadores (llamados Procesadores en la especificación de flujos reactivos, véase la sección 3.5). Estos operadores cubren desde simples transformaciones y filtros de los datos en el flujo hasta orquestación compleja y gestión de errores.
- Se define el flujo reactivo, pero hasta que no se realice una suscripción no se comienzan a enviar datos, lo que contribuye a la reusabilidad y composición del código.
- Implementa uno de los requisitos en los sistemas reactivos (véase la sección 3.3), que es la habilidad de controlar el flujo de datos que fluyen de un componente a otro mediante *back-pressure*.

La base de Project Reactor son sus dos implementaciones de la entidad Publisher<T> de la especificación de flujos reactivos en Java. La primera de ellas se denomina Mono<T>, el cual emite como mucho un elemento, el cual puede estar vacío o no (también llamado flujo de 0 o 1) . La ultima se asemeja mas a la idea de Publisher<T> tradicional; se denomina Flux<T> y representa una secuencia de 0 a N elementos.

Como se ha explicado anteriormente, una vez instanciado un Publicador es necesario suscribirse de manera que se comiencen a enviar los elementos. A continuación, se muestran ejemplos simples del comportamiento de Mono<T> y Flux<T>:

```
1 Mono<String> mensaje = Mono.just("Ejemplo Mono");
2 mensaje.subscribe(valor -> System.out.println(valor));
3 // consola:
4 // Ejemplo Mono
5
6 Flux<String> mensajes = Flux.just("Ejemplo", "Flux");
7 mensajes.subscribe(valor -> System.out.println(valor));
8 // consola:
9 // Ejemplo
10 // Flux
```

Se pueden definir operadores en la definición de Mono<T> y Flux<T> de manera que se realicen transformaciones antes de entregar los datos a los suscriptores, por ejemplo:

```
1 Mono<String> mensaje = Mono.just("Ejemplo Mono")
2     .map(elemento -> elemento.toLowerCase());
3
4 mensaje.subscribe(valor -> System.out.println(valor));
5 // consola:
6 // ejemplo mono
7
8 Flux<String> mensajes = Flux.just("Ejemplo", "Flux")
9     .map(elemento -> elemento.toUpperCase());
```

```
10
11 mensajes.subscribe(valor -> System.out.println(valor));
12 // consola:
13 // EJEMPLO
14 // FLUX
```

Declarar un Flux o un Mono no implica que la ejecución ocurra en otro hilo, sino que por defecto se procesa en el mismo hilo en el que se ha invocado al método *subscribe()* (en los casos anteriores, el hilo *main*). En Project Reactor se deben utilizar los métodos estáticos de la clase Schedulers para controlar el contexto de la ejecución donde se llevan a cabo las operaciones, los cuales se describen en la Tabla 3.

Contexto	Descripción
Schedulers.immediate()	La tarea se ejecuta directamente en el hilo actual.
Schedulers.single()	Hace uso de un único hilo reusable. El mismo hilo se usa para cada invocación de single().
Schedulers.newSingle()	Similar a single(), pero en vez de reutilizar el mismo hilo, se crea uno nuevo cada vez.
Schedulers.boundedElastic()	Crea nuevas piscinas de hilos de trabajo bajo demanda y reutiliza las que ya han sido creadas que estén disponibles. Las piscinas de hilos que se encuentren libres por más de 60 segundos (valor por defecto) son eliminadas. Útil cuando se necesita crear una nueva piscina de hilos para la realización de alguna tarea bloqueante de tipo Entrada/Salida, por ejemplo.
Schedulers.parallel()	Accede a una piscina de hilos fija configurada y optimizada para la ejecución de tareas en paralelo. Crea tantos hilos en la piscina de hilos como núcleos tenga el procesador. Debe ser usado para la ejecución de tareas rápidas no bloqueantes ya que hace uso de todos los núcleos del procesador y es posible que se bloquee la aplicación hasta que la ejecución de las tareas finalice y los hilos se liberen.

Tabla 3. Métodos disponibles en Schedulers para el cambio de contexto de ejecución en Project Reactor.

Se puede modificar el contexto de ejecución de las tareas dentro de una cadena reactiva haciendo uso de los métodos *publishOn()* y *subscribeOn()*, los cuales reciben una instancia de Scheduler. Por ejemplo:

```
1 Scheduler scheduler = Schedulers.single();
2
3 Flux<String> mensajes = Flux.just("Ejemplo", "Flux")
```

```
    .publishOn(scheduler)
    .map(elemento -> elemento.toUpperCase());

mensajes.subscribe(valor -> {
    System.out.print(Thread.currentThread().getName() + ": ");
    System.out.println(valor);
});

System.out.println(Thread.currentThread().getName());
Thread.sleep(250);

// consola:
// main
// single-1: EJEMPLO
// single-1: FLUX
```

En el ejemplo anterior, la suscripción se realiza en el hilo principal de la aplicación, pero la operación *map()* que se encuentra justo después de *publishOn()* se ejecuta en un nuevo hilo llamado "single-1". Cualquier operación que se defina después de *publishOn()* se ejecuta en el contexto definido en el Scheduler pasado como parámetro. Se pueden incluir múltiples invocaciones a *publishOn()* en la cadena reactiva para cambiar de contexto según se necesite.

```
Scheduler scheduler = Schedulers.single();

Flux<String> mensajes = Flux.just("Ejemplo", "Flux")
    .map(elemento -> elemento.toUpperCase())
    .subscribeOn(scheduler);

mensajes.subscribe(valor -> {
    System.out.print(Thread.currentThread().getName() + ": ");
    System.out.println(valor);
});

System.out.println(Thread.currentThread().getName());
Thread.sleep(250);

// consola:
// main
// single-1: EJEMPLO
// single-1: FLUX
```

En este ejemplo, se modifica el hilo donde se realiza la suscripción, de manera que se ejecuta en un nuevo hilo llamado "single-1". Solo se toma en consideración una invocación a *subscribeOn()*, de manera que siguientes invocaciones en la cadena reactiva son inútiles. Haciendo uso conjunto de *subscribeOn()* y *publishOn()* se puede, en primer lugar, modificar el contexto de donde se realiza la suscripción, y posteriormente en la cadena reactiva modificar el contexto donde se ejecutan los siguientes operadores, por ejemplo:

```
Scheduler scheduler = Schedulers.single();
Scheduler scheduler2 = Schedulers.boundedElastic();

Flux<String> mensajes = Flux.just("Flux")
  .map(elemento -> {
      System.out.println("Primer mapeo ejecutado en: " + Thread.currentThread().getName());
      return elemento.toUpperCase();
  })
  .publishOn(scheduler2)
  .map(elemento -> {
      System.out.println("Segundo mapeo ejecutado en: " + Thread.currentThread().getName());
      return elemento.toLowerCase();
  })
  .subscribeOn(scheduler);

mensajes.subscribe(valor -> {
    System.out.println("Suscriptor procesa el valor en: " + Thread.currentThread().getName());
});

System.out.println(Thread.currentThread().getName());
Thread.sleep(250);

// consola:
// main
// Primer mapeo ejecutado en: single-1
// Segundo mapeo ejecutado en: boundedElastic-1
// Suscriptor procesa el valor en: boundedElastic-1
```

De esta forma, Project Reactor ofrece una implementación muy potente a la hora de intercambiar datos entre barreras asíncronas (por ejemplo, hilos) de forma transparente al programador, mediante mecanismos más sencillos y legibles que las alternativas ofrecidas por defecto en Java.

4.4. RxJava

RxJava es una biblioteca reactiva que surge como implementación del proyecto ReactiveX (Reactive Extensions)[10] para la JVM. ReactiveX es una biblioteca implementada en varios lenguajes (siendo RxJava su implementación en Java) para la creación de programas asíncronos y basados en eventos mediante el uso de secuencias observables. Es una combinación de las mejores ideas del patrón observador, el patrón iterador y la programación funcional. Existen tres versiones principales de RxJava, siendo v3 la actual que reemplaza a la anteriores v1 y v2, marcadas como deprecadas y sin soporte de cara a futuro [24]. Debido a que el lanzamiento de RxJava v1 precedió a la creación de la especificación de flujos reactivos en Java, no fue hasta la llegada de la versión dos que se incluyó una implementación directa de Publisher<T> con *back-pressure*. Los tipos disponibles en RxJava v3 se describen en la Tabla 4.

[10] https://reactivex.io

Tipo	Descripción	Consumidor
Flowable	Implementación directa de Publisher<T> proveniente de la especificación de flujos reactivos en Java introducido en RxJava v2. Soporta *back-pressure* y se define como un flujo de 0 a N elementos.	Subscriber
Observable	Tipo proveniente de RxJava v1 mantenido en las siguientes versiones. No se basa en la especificación de flujos reactivos en Java, por tanto, no soporta *back-pressure*. Se define como un flujo de 0 a N elementos.	Observer
Single	Tipo proveniente de RxJava v1 mantenido en las siguientes versiones. Flujo de un elemento o error.	SingleObserver
Maybe	Introducido en RxJava v2. Representa un flujo sin elementos, un elemento, o error.	MaybeObserver
Completable	Tipo proveniente de RxJava v1 mantenido en las siguientes versiones. Flujo sin elementos que se completa normalmente o con error.	CompletableObserver

Tabla 4. Tipos reactivos RxJava v3.x.

Al igual que Project Reactor, en RxJava se define un Observable y solo cuando se realiza la suscripción se comienzan a enviar los elementos hacia el Suscriptor. Los siguientes fragmentos de código replican los ejemplos mostrados en la sección anterior con tipos reactivos de RxJava v3. El tipo Observable define un flujo igual a Flowable (0 a N), aunque no soporta *back-pressure*. En los ejemplos siguientes se pueden intercambiar estos dos tipos y se conseguiría el mismo resultado:

```
1 Single<String> mensaje = Single.just("Ejemplo Single");
2 mensaje.subscribe(valor -> System.out.println(valor));
3 // consola:
4 // Ejemplo Single
5
6 Observable<String> mensajes = Observable.just("Ejemplo", "Observable");
7 mensajes.subscribe(valor -> System.out.println(valor));
8 // consola:
9 // Ejemplo
10 // Observable
```

Se pueden definir operaciones en la cadena reactiva de manera que se realicen transformaciones antes de entregar los datos a los suscriptores, por ejemplo:

```
1 Single<String> mensaje = Single.just("Ejemplo Single")
2     .map(elemento -> elemento.toLowerCase());
3
4 mensaje.subscribe(valor -> System.out.println(valor));
5 // consola:
6 // ejemplo single
7
8 Observable<String> mensajes = Observable.just("Ejemplo", "Observable")
9     .map(elemento -> elemento.toUpperCase());
10
11 mensajes.subscribe(valor -> System.out.println(valor));
12 // consola:
13 // EJEMPLO
14 // OBSERVABLE
```

A la hora de ejecutar operaciones de la cadena reactiva de forma asíncrona en nuevos hilos, RxJava proporciona su propia implementación de la clase Schedulers, similar a la implementada en Project Reactor. Los métodos disponibles se describen en la Tabla 5.

Método	Descripción
Schedulers.computation()	Ejecuta operaciones computacionalmente intensivas en una piscina de hilos con tamaño menor o igual a la cantidad de procesadores disponibles.
Schedulers.io()	Ejecuta operaciones de tipo Entrada/Salida o bloqueantes en una piscina de hilos que crece de forma dinámica según demanda.
Schedulers.single()	Ejecuta tareas en un hilo único siguiendo un modelo FIFO (del término anglosajón *First In First Out*).
Schedulers.trampoline()	Encola las tareas siguiendo un modelo FIFO en el mismo hilo donde se invoca.

Tabla 5. Métodos disponibles en Schedulers para el cambio de contexto de ejecución en RxJava.

Se puede modificar el contexto de ejecución de las tareas dentro de una cadena reactiva haciendo uso de los métodos *observeOn()* y *subscribeOn()*, que reciben una instancia de Scheduler indicando el contexto deseado:

```
1 Scheduler scheduler = Schedulers.single();
2
3 Flowable<String> mensajes = Flowable.just("Ejemplo", "Flowable")
4         .observeOn(scheduler)
5         .map(elemento -> elemento.toUpperCase());
6
```

```
 7 mensajes.subscribe(valor -> {
 8     System.out.print(Thread.currentThread().getName() + ": ");
 9     System.out.println(valor);
10 });
11
12 System.out.println(Thread.currentThread().getName());
13 Thread.sleep(250);
14
15 // consola:
16 // main
17 // RxSingleScheduler-1: EJEMPLO
18 // RxSingleScheduler-1: FLOWABLE
```

En el ejemplo anterior, la suscripción se realiza en el hilo principal, pero la operación *map()* se realiza en un hilo creado con *Schedulers.single()*. De la misma forma que con *publishOn()* en Project Reactor, cualquier operación que se defina después de *observeOn()* en RxJava se ejecuta en el contexto definido en el Scheduler pasado como parámetro. Se pueden incluir múltiples invocaciones a *observeOn()* en la cadena reactiva para cambiar de contexto según se necesite. Se ha incluido *Thread.sleep(250)* de manera que el hilo principal (main) no acabe la ejecución del código instantáneamente y espere a que el Suscriptor procese los mensajes en el otro hilo (RxSingleScheduler-1).

```
 1 Scheduler scheduler = Schedulers.single();
 2
 3 Flowable<String> mensajes = Flowable.just("Ejemplo", "Flowable")
 4     .map(elemento -> elemento.toUpperCase())
 5     .subscribeOn(scheduler);
 6
 7 mensajes.subscribe(valor -> {
 8     System.out.print(Thread.currentThread().getName() + ": ");
 9     System.out.println(valor);
10 });
11
12 System.out.println(Thread.currentThread().getName());
13 Thread.sleep(250);
14
15 // consola:
16 // main
17 // RxSingleScheduler-1: EJEMPLO
18 // RxSingleScheduler-1: FLOWABLE
```

El método *subscribeOn()* presenta el mismo comportamiento que en Project Reactor, una vez invocado, independientemente del lugar de invocación en la cadena reactiva, se modifica el hilo donde se realiza la suscripcion según el Scheduler proporcionado como parámetro. Solo se toma en consideración una invocación a *subscribeOn()*, de manera que siguientes invocaciones en la cadena reactiva son inútiles. Se puede combinar *subscribeOn()* y *observeOn()* para modificar, en primer lugar, el contexto donde se realiza la suscripcion, y posteriormente controlar el contexto donde se ejecutan cada una de las operaciones de la cadena reactiva, como se muestra en el siguiente ejemplo:

```
1  Scheduler scheduler = Schedulers.single();
2  Scheduler scheduler2 = Schedulers.io();
3
4  Flowable<String> mensajes = Flowable.just("Flowable")
5    .map(elemento -> {
6        System.out.println("Primer mapeo ejecutado en: " + Thread.currentThread().getName());
7        return elemento.toUpperCase();
8    })
9    .observeOn(scheduler2)
10   .map(elemento -> {
11       System.out.println("Segundo mapeo ejecutado en: " + Thread.currentThread().getName());
12       return elemento.toLowerCase();
13   })
14   .subscribeOn(scheduler);
15
16 mensajes.subscribe(valor -> {
17     System.out.println("Suscriptor procesa el valor en: " + Thread.currentThread().getName());
18 });
19
20 System.out.println(Thread.currentThread().getName());
21 Thread.sleep(250);
22
23 // consola:
24 // main
25 // Primer mapeo ejecutado en: RxSingleScheduler-1
26 // Segundo mapeo ejecutado en: RxCachedThreadScheduler-1
27 // Suscriptor procesa el valor en: RxCachedThreadScheduler-1
```

Tanto Project Reactor como RxJava v3 proporcionan implementaciones muy potentes a la hora de intercambiar datos entre barreras asíncronas (por ejemplo, hilos) de forma transparente al programador, mediante mecanismos más sencillos y legibles que las alternativas ofrecidas por defecto en Java. Aunque pueda parecer que son iguales, internamente RxJava v3 y Project Reactor presentan diferencias importantes cuyo análisis no es el enfoque de este trabajo.

5. DESARROLLO DE PROTOTIPO REACTIVO

Para demostrar el funcionamiento del framework mediante un caso práctico, se ha desarrollado un prototipo del backend de un portal de críticas de películas similar a Filmaffinity[11]. El prototipo consta de dos microservicios reactivos, *películas-service* y *reviews-service*; en las siguientes secciones se detalla el diseño de cada uno y los pasos necesarios para desplegarlos.

5.1. peliculas-service

Este microservicio se encarga de gestionar las peliculas, su dirección (formada por un director), y el reparto (formado por 0 o N actores). En la Figura 5 se muestra el esquema de base de datos correspondiente.

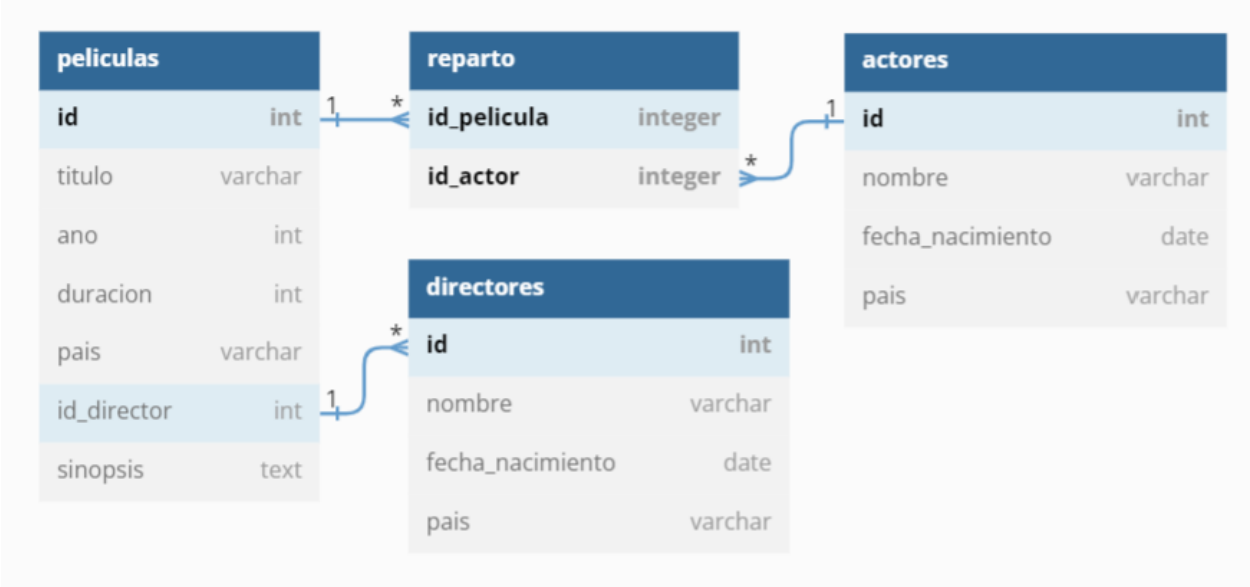

Figura 5. Esquema de base de datos peliculas-service.

La base de datos usada para este microservicio fue PostgreSQL, desplegada en forma de contenedor en Docker. Dentro de los archivos del proyecto, el archivo *docker-compose.yaml* (véase la Figura 6) contiene la definición del despliegue de la base de datos y la interfaz pgAdmin, además del archivo de creación de tablas e inserción de documentos en la ruta */db/PostgreSQL/db.sql* (véase la Figura 7).

[11] https://www.filmaffinity.com

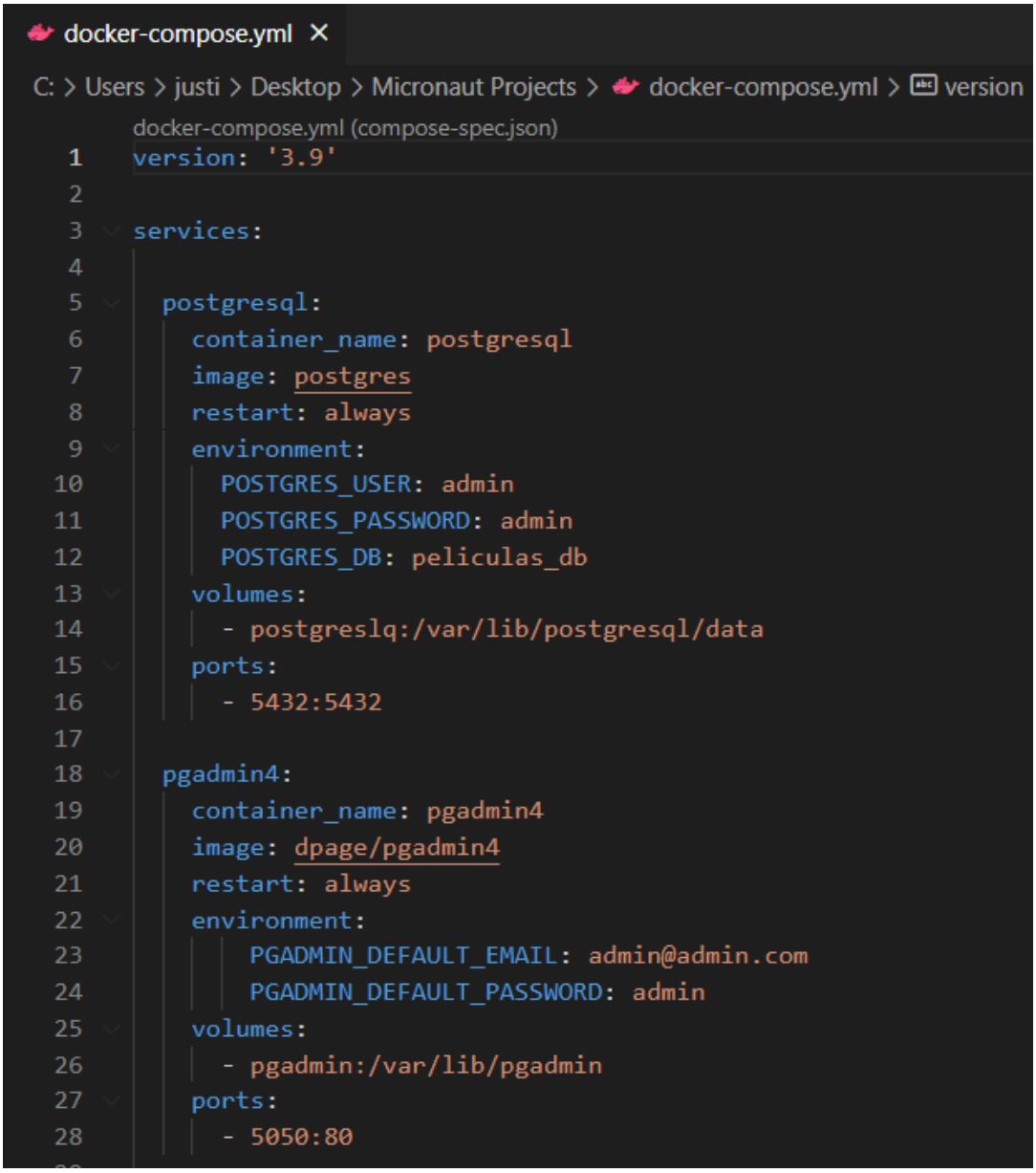

Figura 6. Despliegue PostgreSQL.

db.sql

C: > Users > justi > Desktop > Micronaut Projects > db > PostgreSQL > db.sql

```sql
-- Crea nuevo rol para el microservicio
CREATE ROLE peliculas_service WITH
  LOGIN
  SUPERUSER
  INHERIT
  CREATEDB
  CREATEROLE
  REPLICATION
  PASSWORD 'peliculas_service';

-- Crea nuevo schema con autorizacion para el nuevo rol creado
CREATE SCHEMA peliculas_schema
    AUTHORIZATION peliculas_service;

-- Cambia del schema publico a peliculas_schema
SET SCHEMA 'peliculas_schema';

DROP TABLE IF EXISTS peliculas_schema.reparto;
DROP TABLE IF EXISTS peliculas_schema.direccion;
DROP TABLE IF EXISTS peliculas_schema.peliculas;
DROP TABLE IF EXISTS peliculas_schema.actores;
DROP TABLE IF EXISTS peliculas_schema.directores;

CREATE TABLE actores(
    id serial PRIMARY KEY,
    nombre character varying(80),
    fecha_nacimiento date,
    pais character varying(60)
);
CREATE INDEX idx_actores_nombre ON actores (nombre);
```

Figura 7. Archivo de creación de la base de datos.

En la Figura 8 se muestra la configuración inicial del proyecto junto con las características necesarias para su correcto funcionamiento usando la utilidad Micronaut Launch[12].

[12] https://micronaut.io/launch/

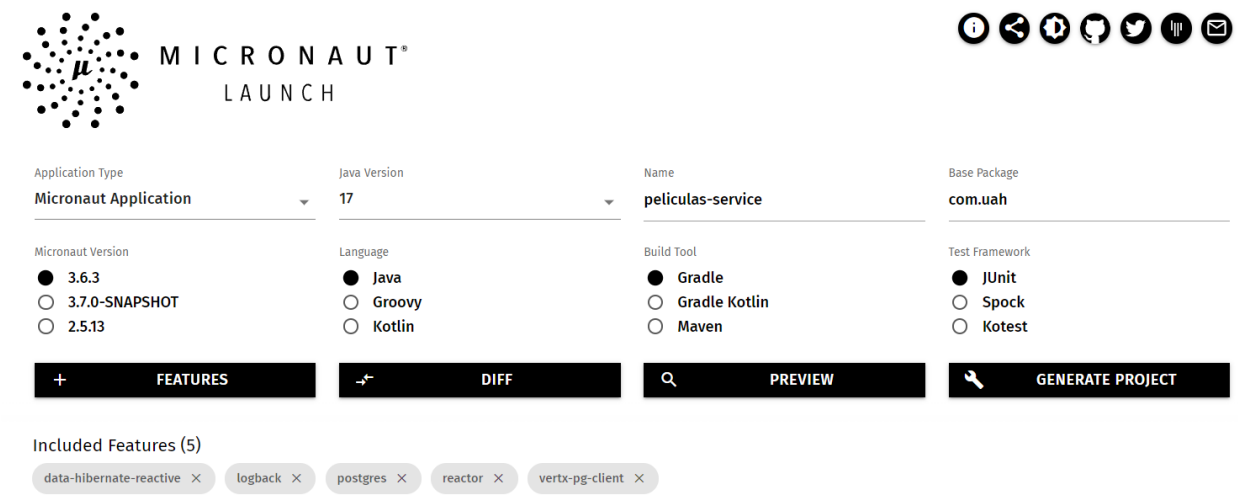

Figura 8. Dependencias y configuración peliculas-service.

Se incluye la dependencia de Hibernate Reactive y el driver reactivo para PostgreSQL de forma que la interacción desde la aplicación sea asíncrona y no bloqueante, tal y como se explica en la sección 4.2.2. Finalmente se incluyen dependencias para la biblioteca reactiva Project Reactor y la gestión de logs dentro de la aplicación usando el framework Logback.

Se prepara el archivo de configuración de la aplicación con los parámetros mostrados en la Figura 9. En primer lugar, se configura la cantidad de hilos usado por el servidor http Netty encargado de gestionar las peticiones de los controladores a 1, de forma que se pueda apreciar mejor el mecanismo del bucle de eventos. Finalmente, se configura Hibernate Reactive indicando los parámetros para conectarse a la base de datos una vez esta desplegada y se permite el uso de drivers reactivos con la propiedad "reactive".

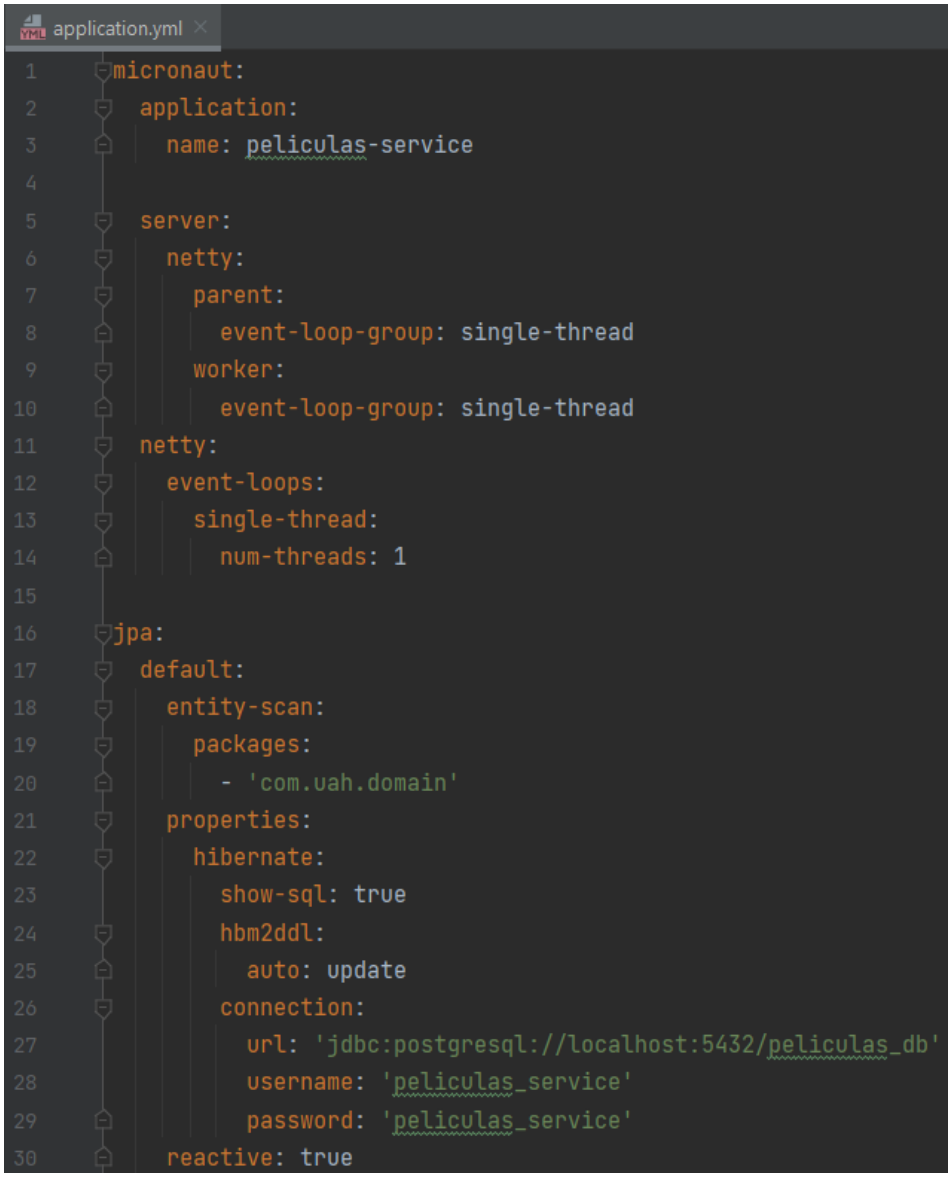

Figura 9. Archivo de configuracion application.yaml de peliculas-sevice.

5.1.1. Clases de entidad, repositorios, servicios y controladores

Como se ha mencionado previamente, el microservicio gestiona tres entidades principales, las peliculas, los actores y los directores. Se han creado recursos para ejecutar operaciones CRUD de cada una de estas entidades, resultando en la estructura de archivos que se muestra en la Figura 10. En primer lugar, se generan las clases de entidad y se configuran correctamente las relaciones entre ellas, posteriormente se crean las interfaces en la capa del repositorio para gestionar el acceso a base de datos, seguido por la creación de las interfaces de los servicios y su implementación, finalizando por la creación de los controladores.

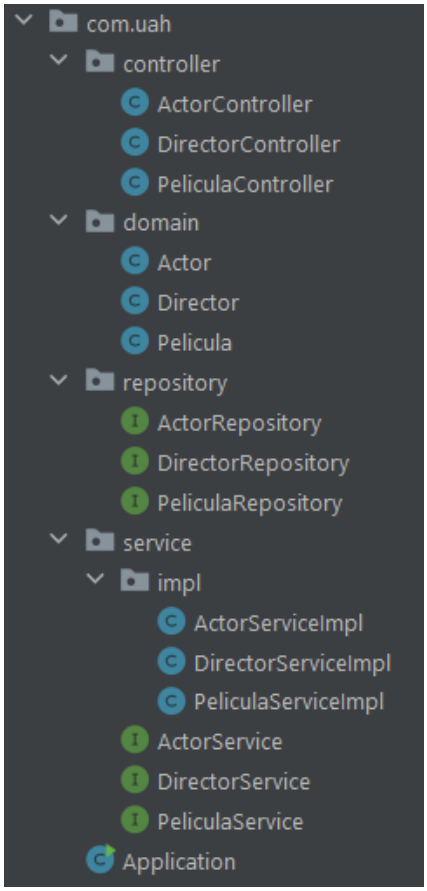

Figura 10. Estructura peliculas-service.

En la capa del repositorio, las interfaces creadas extienden ReactorCrudRepository<T, K>. Esta interfaz hace uso de Hibernate Reactive y el driver reactivo para PostgreSQL para interactuar con la base de datos de manera asíncrona y no bloqueante. Por este motivo, cada uno de los recursos disponibles en el repositorio devuelven tipos reactivos de la biblioteca Project Reactor como Flux<T> y Mono<T> (véase la sección 4.3). Estos tipos reactivos son propagados en la capa del servicio hasta llegar al controlador, donde son devueltos como respuesta. Haciendo uso de la función *log()* disponible en Project Reactor podemos observar el comportamiento de la cadena reactiva y ver el contexto se han ejecutado cada una de las fases.

En la Figura 11 se puede observar la traza resultante de logs al realizar una petición al recurso [GET] /peliculas haciendo uso de la función *log()*. Como se ha explicado en secciones previas, al devolver un tipo reactivo en los controladores, Micronaut se subscribe al tipo reactivo en el hilo que gestiona el bucle de eventos, por este motivo se observa que *onSubscribe()* se realiza en el hilo *single-thread-nioEventLoopGroup-3-1*. Una vez suscrito comienza a requerir elementos de uno en uno mediante la función *request(1)*, es aquí cuando el repositorio gestiona el acceso a base de datos de forma asíncrona en el driver reactivo de Vert.x para PostgreSQL y se puede apreciar que el primer elemento se procesa en el hilo *vert.x-eventloop-thread-1* mediante la función *onNext()*. Una vez se han leído los datos de base de datos de forma asíncrona, se procesan el resto de los elementos en el hilo donde se realizó la suscripcion. Gracias al uso de los drivers reactivos se delega el acceso a base de datos a otro hilo de forma que el bucle de eventos no se bloquea nunca y se mantiene disponible para gestionar otras peticiones entrantes.

```
12:03:01.764 [single-thread-nioEventLoopGroup-3-1] INFO  c.uah.controller.PeliculaController - [GET] /peliculas
12:03:01.817 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - onSubscribe(FluxMap.MapSubscriber)
12:03:01.821 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - request(1)
12:03:02.027 [vert.x-eventloop-thread-1] INFO  reactor.Flux.Lift.1 - onNext({id: 1,ano:2022,duracion:240,pais:United
12:03:02.188 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - request(1)
12:03:02.188 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - onNext({id: 3,ano:2022,duracion:139,pa
12:03:02.190 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - request(1)
12:03:02.190 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - onNext({id: 2,ano:2022,duracion:131,pa
12:03:02.190 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - onComplete()
12:03:02.191 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - request(1)
12:03:02.191 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - request(1)
```

Figura 11. Acceso reactivo a base de datos.

En la Figura 12 se muestran los recursos expuestos por peliculas-service y en la Figura 13 parte de la respuesta de [GET] /peliculas.

Figura 12. Recursos expuestos por peliculas-service.

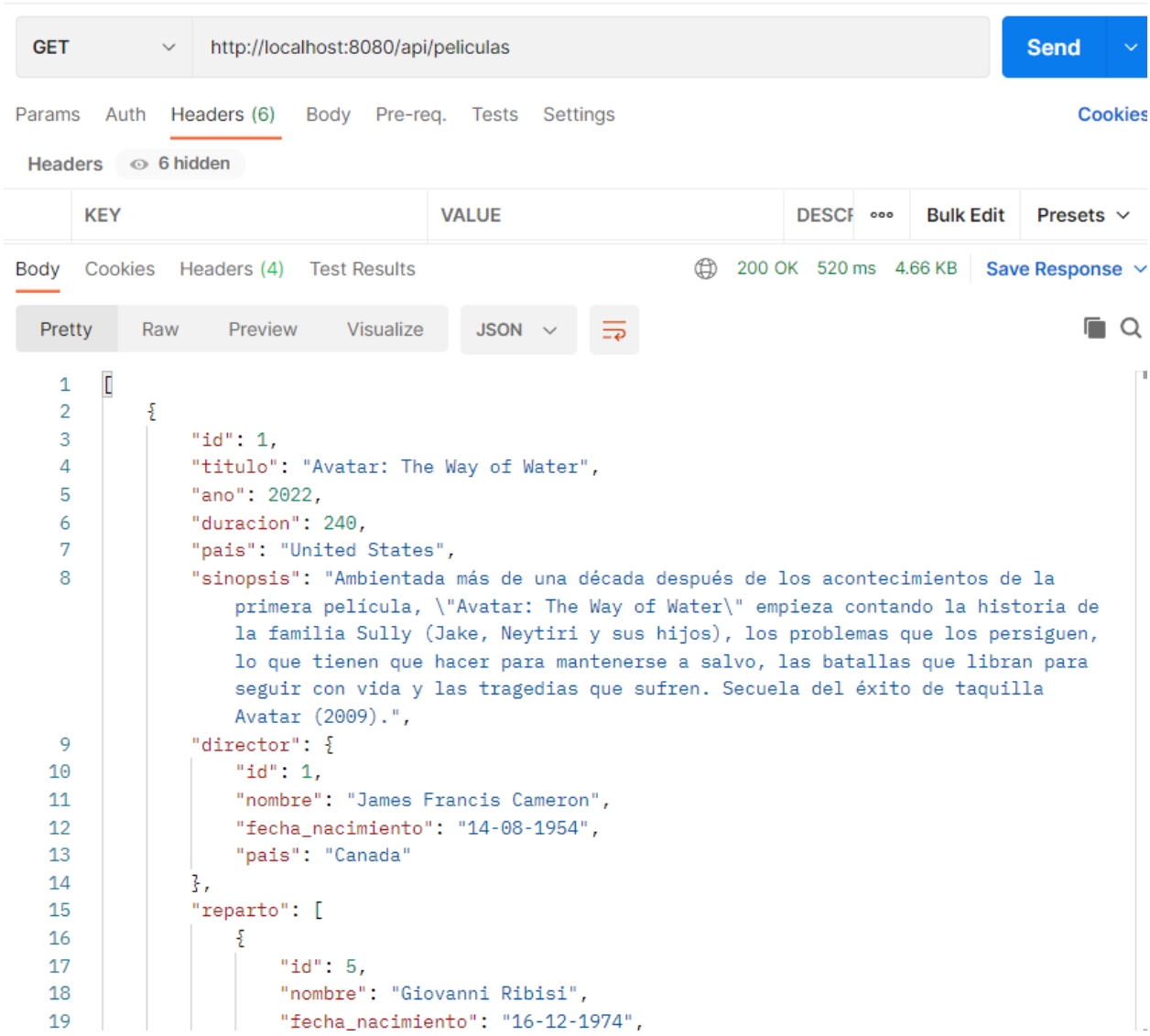

Figura 13. GET /peliculas.

5.2. reviews-service

Este ultimo microservicio se encarga de gestionar las reviews hechas por los usuarios a cada una de las peliculas. En la Figura 14 se muestra el esquema de base de datos correspondiente.

Figura 14. Esquema de base de datos reviews-service.

Para este microservicio se decidió usar la base de datos NoSQL MongoDB, desplegada en forma de contenedor en Docker. Dentro de los archivos del proyecto, el archivo *docker-compose.yaml* (véase la Figura) contiene la definición del despliegue de la base de datos, además de los archivos con los datos de prueba a importar en cada una de las colecciones en */db/MongoDB*.

```
30    monogdb:
31      container_name: mongodb
32      image: mongo
33      restart: always
34      environment:
35        MONGO_INITDB_ROOT_USERNAME: admin
36        MONGO_INITDB_ROOT_PASSWORD: admin
37      volumes:
38        - mongo:/data/db
39      ports:
40        - 27017:27017
41
42  volumes:
43    postgreslq:
44    pgadmin:
45    mongo:
```

Figura 15. Despliegue MonogDB.

En la Figura 16 se muestra la configuración inicial del proyecto junto con las características necesarias para su correcto funcionamiento usando la utilidad Micronaut Launch[13].

[13] https://micronaut.io/launch/

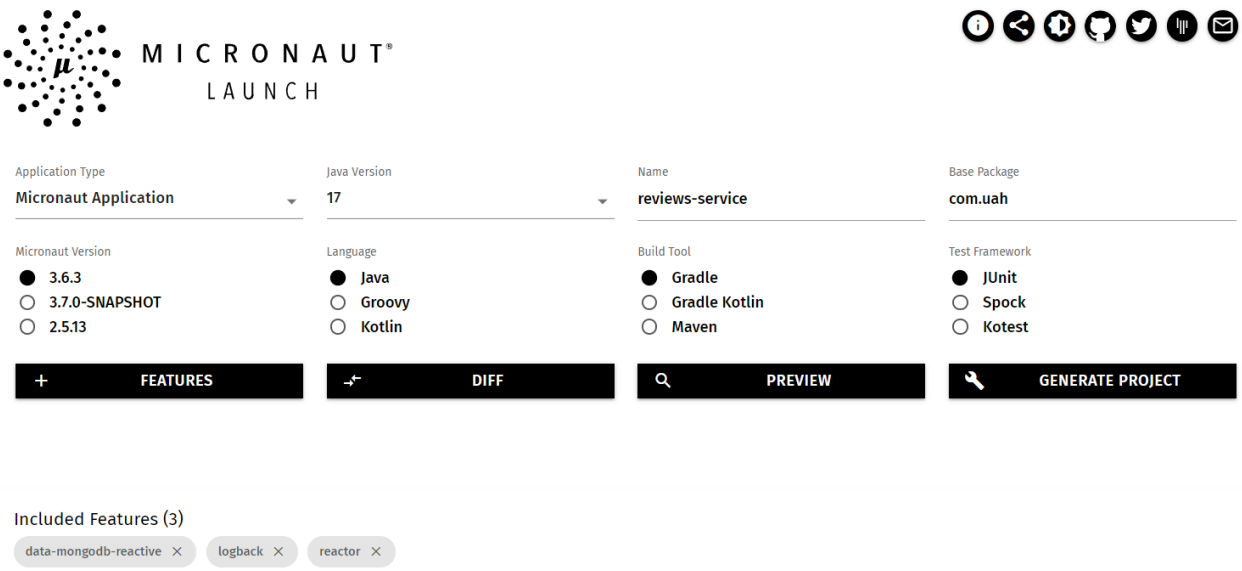

Figura 16. Dependencias y configuracion reviews-service.

Se incluye la dependencia Micronaut Data para MongoDB reactive y Project Reactor para gestionar el acceso a base de datos de forma asíncrona y no bloqueante, tal y como se explica en la sección 4.2.1. Se incluye también el framework Logback para la gestión de logs dentro de la aplicación.

El archivo de configuracion mostrado en la Figura 17 muestra los parámetros configurados para el correcto funcionamiento de la aplicación. De la misma forma que para peliculas-service, se configura la cantidad de hilos usado por el servidor http Netty encargado de gestionar las peticiones de los controladores a 1, de forma que se pueda apreciar mejor el mecanismo del bucle de eventos. Finalmente se indica la cadena de conexión a MongoDB.

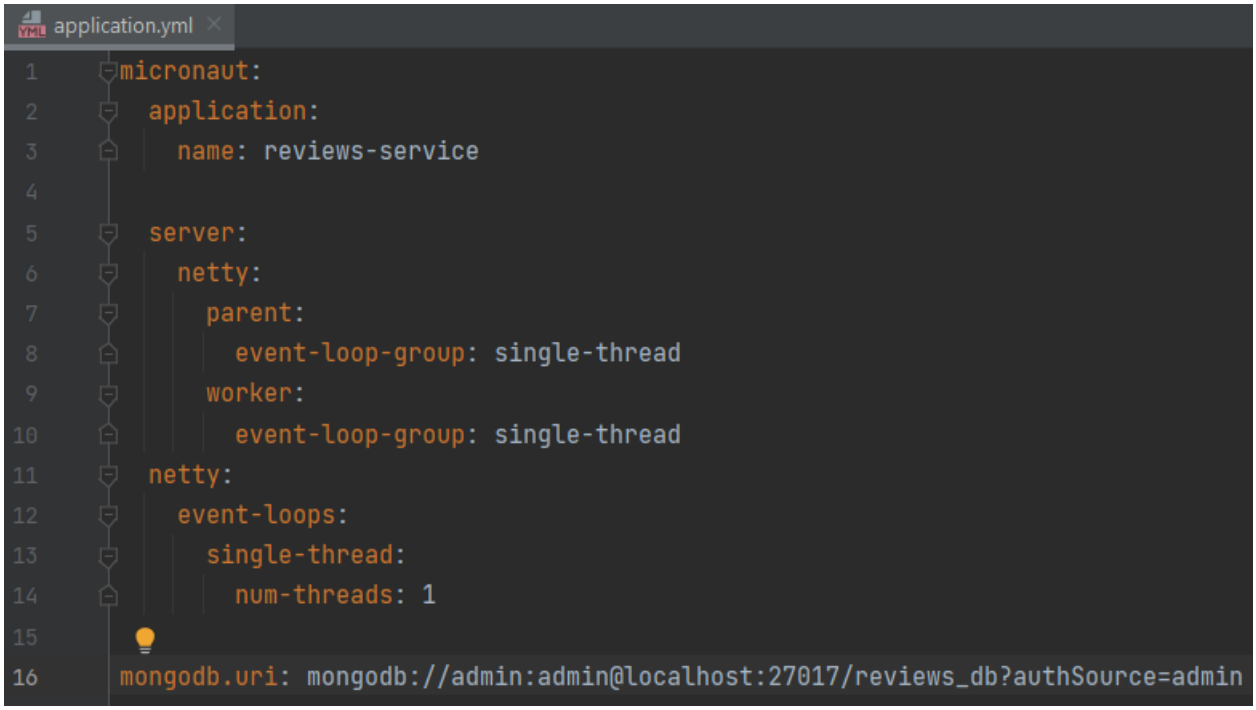

Figura 17. Archivo de configuracion application.yml de reviews-service.

5.2.1. Clases de entidad, repositorios, servicios y controladores

Este microservicio gestiona dos entidades principales, los usuarios y las reviews. Se han creado recursos para ejecutar operaciones CRUD de cada una de estas entidades, resultando en la estructura de archivos que se muestra en la Figura 18. En primer lugar, se generan las clases de entidad y se configuran correctamente las relaciones entre ellas, posteriormente se crean las interfaces en la capa del repositorio para gestionar el acceso a base de datos, seguido por la creación de las interfaces de los servicios y su implementación, finalizando por la creación de los controladores.

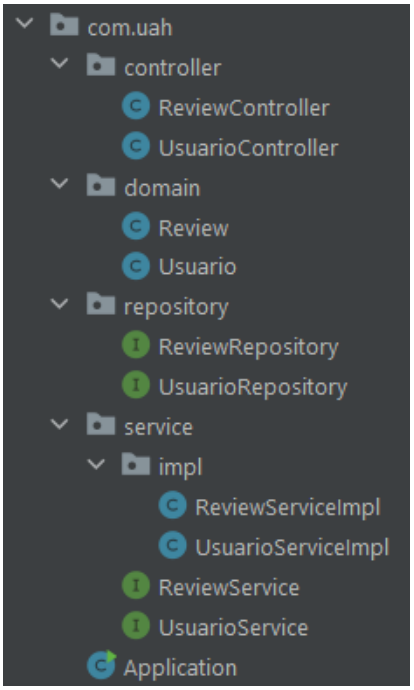

Figura 18. Estructura reviews-service.

En la capa del repositorio, las interfaces creadas extienden ReactorCrudRepository<T, K>. En este caso, esta interfaz hace uso de Micronaut Data y el driver reactivo para MongoDB para interactuar con la base de datos de manera asíncrona y no bloqueante. Por este motivo, cada uno de los recursos disponibles en el repositorio devuelven tipos reactivos de la biblioteca Project Reactor como Flux<T> y Mono<T> (véase la sección 4.3). Estos tipos reactivos son propagados en la capa del servicio hasta llegar al controlador, donde son devueltos como respuesta. Haciendo uso de la función *log()* disponible en Project Reactor podemos observar el comportamiento de la cadena reactiva y ver el contexto se han ejecutado cada una de las fases.

En la Figura 19 se puede observar la traza resultante de logs al realizar una petición al recurso [GET] /usuarios/{id}/reviews haciendo uso de la función *log()*.

```
13:07:41.443 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - onSubscribe(FluxMap.MapSubscriber)
13:07:41.454 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - request(1)
13:07:41.548 [Thread-17] INFO  org.mongodb.driver.connection - Opened connection [connectionId{localValue:3, serverValue:209}] to localhost:27017
13:07:41.566 [Thread-17] INFO  reactor.Flux.Lift.1 - onNext({id:63272f40269d8a54618a79df, usuario:{id:63272927269d8a54618a79b5,nombre:Javier, apell
13:07:41.717 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - request(1)
13:07:41.717 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - onNext({id:63272f40269d8a54618a79e0, usuario:{id:63272927269d8a54618
13:07:41.718 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - request(1)
13:07:41.722 [Thread-16] INFO  reactor.Flux.Lift.1 - onNext({id:63272f40269d8a54618a79e1, usuario:{id:63272927269d8a54618a79b8,nombre:Kevin, apelli
13:07:41.724 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - request(1)
13:07:41.724 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - onNext({id:63272f40269d8a54618a79e2, usuario:{id:63272927269d8a54618
13:07:41.725 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - request(1)
13:07:41.728 [Thread-17] INFO  reactor.Flux.Lift.1 - onComplete()
13:07:41.729 [single-thread-nioEventLoopGroup-3-1] INFO  reactor.Flux.Lift.1 - request(1)
```

Figura 19. Comportamiento MonogDB Reactive.

Como se ha explicado en secciones previas, al devolver un tipo reactivo en los controladores, Micronaut se subscribe al tipo reactivo en el hilo que gestiona el bucle de eventos, por este motivo se observa que *onSubscribe()* se realiza en el hilo *single-thread-nioEventLoopGroup-3-1*. Una vez suscrito comienza a requerir elementos de uno en uno mediante la función *request(1)*, es aquí cuando el repositorio gestiona el acceso a base de datos de forma asíncrona en el driver reactivo para MonogDB y se puede apreciar que el hilo llamado "Thread-17" abre la conexión a MonogDB y procesa el primer valor. Posteriormente se procesa otro documento en el hilo del bucle de eventos, pero se vuelve a hacer uso de otro hilo llamado "Thread-16" para recuperar el resto de los documentos de base de datos. El comportamiento de este driver es un poco distinto al descrito en el apartado anterior para PostgreSQL ya que no abre la conexión a la base de datos al arrancar la aplicación sino al ser requerida por primera vez, y además no es capaz de recuperar tantos documentos a la vez por eso se vuelve a conectar a la base de datos con el hilo "Thread-16" y obtener los restantes. Gracias al uso de los drivers reactivos se delega el acceso a base de datos a otro hilo de forma que el bucle de eventos no se bloquea nunca y se mantiene disponible para gestionar otras peticiones entrantes.

En la Figura 20 se muestran los recursos expuestos por peliculas-service y en la Figura 21 parte de la respuesta de [GET] /usuarios/{id}/reviews.

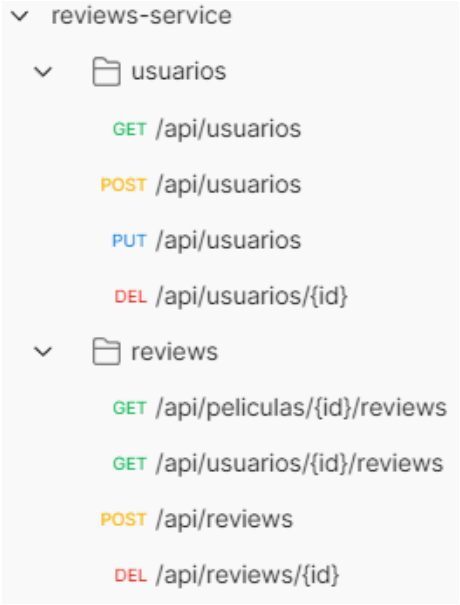

Figura 20. Recursos expuestos por reviews-service.

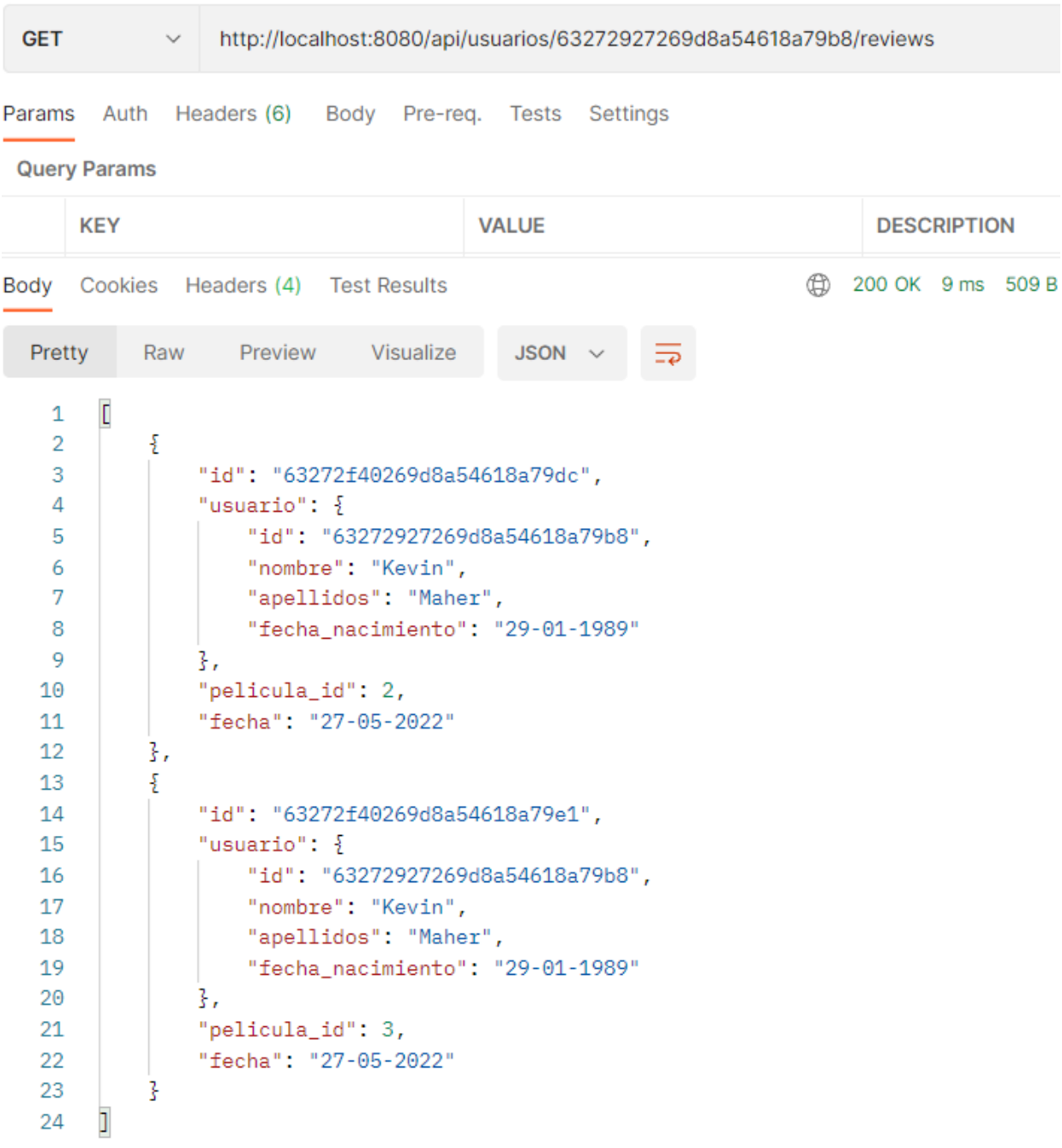

Figura 21. GET /usuarios/{id}/reviews

6. RESUMEN Y CONCLUSIÓN

Este apartado se resume el contenido del trabajo, se especifica el presupuesto total necesario para su desarrollo, se obtienen conclusiones y finalmente se definen líneas de trabajo futuras que no han sido exploradas en este trabajo.

6.1. Resumen

Los microservicios surgen como evolución a los monolitos ya que proporcionan mayor flexibilidad a la hora del desarrollo y mantenimiento de grandes sistemas. Micronaut es un framework orientado al desarrollo de microservicios que ha sido creado desde sus cimientos para reducir al máximo el consumo de memoria y el tiempo de arranque de las aplicaciones; además, proporciona compatibilidad nativa con drivers y bibliotecas reactivas para el desarrollo de microservicios eficientes que cumplan los requisitos y demandas de las aplicaciones actuales. Esto lo convierte en uno de los mejores frameworks actualmente para el desarrollo de microservicios dentro del ecosistema Java.

6.2. Presupuesto

En este apartado se detalla el presupuesto total necesario para el desarrollo del proyecto, organizado por el coste de equipamiento, software necesario, recursos humanos y gastos generales.

6.2.1. Coste de equipamiento

El coste de equipamiento utilizado para la realización de este proyecto y la escritura de la memoria se especifica en la Tabla 6. Consiste en el coste relativo de un ordenador de sobremesa durante la duración del desarrollo del proyecto, considerando un tiempo de vida estimado de cuatro años.

Componente	**Precio**
Ordenador de sobremesa: • Procesador: AMD Ryzen 7 5800X 3.8GHz • Memoria RAM: 16GB DDR4 • Disco duro: SSD NVME M.2 512GB • Tarjeta Gráfica: NVIDIA RTX 3070Ti	2.100 EUR
Coste imputable (6 meses)	**262,5 EUR**

Tabla 6. Coste de equipamiento.

6.2.2. Coste de licencias software

El coste de licencias software mostrado en la Tabla 7 estará compuesto por el precio de cada una de las licencias de software necesarias para el desarrollo del proyecto y del prototipo.

Software	Licencia	Número de Licencias	Precio
Microsoft Windows 10 Home	Licencia Digital	1	141 EUR
IDEA IntelliJ	Ultimate License	1	149 EUR
Docker	Personal	1	0 EUR
Micronaut	Apache License v2.0	1	0 EUR
MongoDB	Server Side Public License (SSPL)	1	0 EUR
PostgreSQL	PostgreSQL Licence	1	0 EUR
pgAdmin 4	PostgreSQL Licence	1	0 EUR
Project Reactor	Apache License v2.0	1	0 EUR
RxJava	Apache License v2.0	1	0 EUR
Office 365 69 EUR/Año	Microsoft 365 Personal	1	69 EUR
Total			**359 EUR**

Tabla 7. Coste de licencias de software.

6.2.3. Costes de recursos humanos

El coste de recursos humanos (véase la Tabla 8) se constituye de la cantidad de horas dedicadas por cada uno de los integrantes de un equipo de desarrollo habitual para este tipo de proyectos, y el precio por hora dependiendo del rol de cada uno. Aunque este proyecto haya sido realizado por el estudiante, se ha querido representar el coste que tendría dicho proyecto en un ámbito real.

Las entidades necesarias para la ejecución de las tareas del proyecto son la siguientes:

- **Jefe de proyecto**: Encargado/a de la gestión del equipo y toma de decisiones principales que ayuden a conseguir los objetivos del proyecto.

- **Analista**: Encargado/a del análisis del problema y de las herramientas y tecnologías necesarias que se adecúen a los objetivos del proyecto y sus limitaciones.
- **Programador**: Encargado/a del desarrollo del sistema resultado del análisis del analista haciendo uso de las tecnologías requeridas.
- **Secretaría**: Encargado/a de la escritura de la memoria.

Concepto	Horas	Coste/Hora	Coste Total
Jefe de proyecto	8	50 EUR	400 EUR
Analista	50	35 EUR	1.750 EUR
Programador	100	26 EUR	2.600 EUR
Secretaría	55	18 EUR	990 EUR
Total			**5.740 EUR**

Tabla 8. Coste de recursos humanos.

6.2.4. Costes generales

Se denominan costes generales a aquellos relacionados con el lugar de trabajo, como son los gastos de material de oficina y mantenimiento de las instalaciones. Estos costes se definen como un 15% del coste de equipamiento, licencias y recursos humanos. Así, este coste incurre en **954.22 EUR**.

6.2.5. Coste Total

EL coste total vendrá determinado por los costes totales acumulados descritos en secciones previas. Una vez calculado como se muestra en la Tabla 9, el coste total de proyecto resulta en **7.315,72 EUR**.

Componente	Precio
Coste de equipamiento	262,5 EUR
Coste de licencias de software	359 EUR
Coste de recursos humanos	5740 EUR
Costes generales	954,22 EUR

Coste total	**7.315,72 EUR**

Tabla 9. Coste total desglosado del proyecto.

6.3. Conclusiones

Los sistemas reactivos surgen como respuesta a los requisitos y demandas crecientes de los usuarios para proporcionar menores tiempos de respuesta, mayor disponibilidad e incremento de volumen de datos. Los microservicios reactivos presentan todas las ventajas de la arquitectura orientada a microservicios y los sistemas reactivos, proporcionando así una alternativa más escalable, mantenible y resistente a los monolitos tradicionales que cumplen con las demandas crecientes de los usuarios. Micronaut es relevante en este aspecto ya que fue concebido para el desarrollo de microservicios de forma nativa con una serie de mejoras para reducir al máximo el consumo de memoria y el tiempo de inicialización de las aplicaciones, además de proporcionar una serie de mecanismos y herramientas para el desarrollo de microservicios reactivos que abarcan todas las capas del desarrollo, desde la comunicación hasta la interacción con bases de datos. Implementa un cliente y servidor HTTP basado en Netty, el cual hace uso de un bucle de eventos para optimizar la gestión de peticiones con el menor número de hilos posibles, soportando tipos reactivos por defecto en la respuesta de los metodos de los controladores. Proporciona integración con drivers reactivos para el acceso a bases de datos tanto SQL como NoSQL, además de otras iniciativas como Hibernate Reactive y drivers R2DBC para la interacción a más bajo nivel. De esta forma se interactúa con la base de datos de forma asíncrona y no bloqueante, a diferencia de los mecanismos y drivers tradicionales. Finalmente, soporta las bibliotecas reactivas más importantes, como Project Reactor y RxJava para desarrollo de aplicaciones reactivas que hagan uso eficiente de los recursos disponibles.

Las mejoras que presenta Micronaut con respecto a otros frameworks dentro del ecosistema Java y las herramientas reactivas que proporciona y soporta lo convierten en la opción más adecuada a la hora de implementar microservicios que aprovechen al máximo los recursos disponibles, el cual es un factor determinante en el mundo de los microservicios.

En el plano personal, la realización del proyecto me ha hecho descubrir el paradigma reactivo y familiarizarme no solo con las bibliotecas reactivas más relevantes como Project Reactor y RxJava, sino también con la especificación de flujos reactivos. He aprendido mucho sobre Micronaut como framework y lo seguiré usando fuera de este proyecto para el desarrollo de proyectos personales debido a la gran cantidad de mejoras que ofrece con respecto a otros frameworks dentro del ecosistema Java. La formación obtenida durante el Master ha sido necesaria no solo a la hora de entender conceptos teóricos presentes durante la investigación previa al proyecto, sino también a la hora de facilitar la adopción de Micronaut como framework, ya que es muy similar a Spring. La principal dificultad encontrada durante la realización del proyecto fue la comprensión de los frameworks reactivos Project Reactor y RxJava y algunos conceptos teóricos relacionados con el paradigma reactivo.

6.4. Futuras líneas de trabajo

El análisis del framework Micronaut llevado a cabo durante este trabajo se ha centrado principalmente en sus capacidades reactivas, aunque se podrían haber explorado más en profundidad otros aspectos los cuales se listan a continuación:

- Creación de ejecutables nativos para GraalVM a partir de aplicaciones Micronaut ya genera sus componentes en tiempo de compilación y no hace uso de mecanismos como reflexión en tiempo de ejecución. El despliegue de aplicaciones en GraalVM reduce enormemente el tiempo de inicio y el consumo de memoria de la aplicación. La aparición de GraalVM es comúnmente referenciada como la innovación mas excitante en el ecosistema Java.
- Proyecto Micronaut Data que hace uso de las APIs de compilación anticipada propias de Micronaut para proporcionar el mejor rendimiento de todos los frameworks Java en operaciones por segundo a la hora de interactuar con bases de datos.
- Exploración en detalle de los mecanismos usados por Micronaut para reducir al mínimo el consumo de memoria y el tiempo de arranque de las aplicaciones.
- Soporte dedicado a la integración con Kafka, RabbitMQ y Nats.io para el desarrollo de microservicios orientados a mensajes.
- Replicar la aplicación usando los frameworks Spring y Quarkus y realizar una comparación de rendimiento.

7. BIBLIOGRAFÍA

[1] Martin Fowler. (25 March 2014). *Microservices*. https://martinfowler.com/articles/microservices.html

[2] Bonér, J. (2016). *Reactive microservices architecture*. O'Reilly Media, Incorporated.

[3] Nirmal Singh & Zack Dawood. (2021). *Building Microservices with Micronaut*. Packt.

[4] Micronaut Docs. *Micronaut for GraalVM*. https://docs.micronaut.io/latest/guide/#graal

[5] Micronaut. (18 June 2019). *Announcing Micronaut Data*. https://micronaut.io/2019/07/18/announcing-micronaut-data/

[6] Martin, Robert C. (2018). *Clean architecture: a craftsman's guide to software structure and design*. Boston.

[7] Micronaut Docs. *Introduction*. https://docs.micronaut.io/latest/guide/

[8] Micronaut Docs. *Dependency Injection*. https://docs.micronaut.io/latest/guide/#ioc

[9] Micronaut Docs. *Configuration Management*. https://docs.micronaut.io/latest/guide/#config

[10] Micronaut Docs. *Aspect Oriented Programming*. https://docs.micronaut.io/latest/guide/#aop

[11] Oracle Docs. *Java Reflection API*. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html

[12] Micronaut. (7 April 2020). *Micronaut vs Quarkus vs Spring Boot Performance on JDK 14*. https://micronaut.io/2020/04/07/micronaut-vs-quarkus-vs-spring-boot-performance-on-jdk-14/

[13] Reactive Manifesto. (16 September 2014). *The Reactive Manifesto*. https://www.reactivemanifesto.org/

[14] Reactive Manifesto. *The reactive Manifesto*. Glossary. https://www.reactivemanifesto.org/glossary

[15] Micronaut Docs. *The HTTP Server*. https://docs.micronaut.io/latest/guide/index.html#httpServer

[16] Micronaut Docs. *JSON Binding with Jackson*. https://docs.micronaut.io/latest/guide/index.html#jsonBinding

[17] MongDB. *MongoDB Reactive Streams Java Driver*. https://mongodb.github.io/mongo-java-driver-reactivestreams

[18] Micronaut Docs. *Micronaut Data MongoDB*. https://micronaut-projects.github.io/micronaut-data/latest/guide/#mongo

[19] Micronaut Docs. *Micronaut Data Hibernate Reactive*. https://micronaut-projects.github.io/micronaut-data/latest/guide/#hibernateReactive

[20] R2DBC. *R2DBC*. https://r2dbc.io/

[21] Reactive-Streams. *Reactive Streams*. http://www.reactive-streams.org/

[22] Micronaut Docs. R2DBC Drivers. https://micronaut-projects.github.io/micronaut-data/latest/guide/#r2dbcConfiguration

[23] Project Reactor. *Asynchronicity to the Rescue?* . https://projectreactor.io/docs/core/release/reference/#_asynchronicity_to_the_rescue

[24] GitHub. *ReactiveX/RxJava*. https://github.com/ReactiveX/RxJava

8. ANEXO A. GLOSARIO

El objetivo de este glosario es facilitar el acceso a una definición de los principales términos que se mencionan a lo largo de este trabajo.

A

AOP: **A**spect **O**riented **P**rogramming. Paradigma de programación.

AOT: **A**head **o**f **T**ime Compilation. Proceso de compilación de un programa en un lenguaje de alto a nivel a otro lenguaje de mas bajo nivel para reducir la cantidad de trabajo que se necesita realizar en tiempo de ejecución.

Asíncrono: Dicho de un evento que se ejecuta en un momento de tiempo arbitrario.

B

Back-pressure: Característica de los sistemas reactivos. Mecanismo que permite regular el flujo de carga de las aplicaciones según su carga.

D

DI: **D**ependency **I**njection. Patron de diseño que permite la inyección de dependencias es un componente sin necesidad de que este sepa cómo construirlo o instanciarlo.

F

Framework: Estructura o conjunto de herramientas usado para el desarrollo de aplicaciones software.

Flujos Reactivos: Estándar para el procesamiento de flujos asíncronos en Java.

H

Hibernate: Framework de asignación objeto-relacional que permite a las aplicaciones comunicarse con bases de datos y conseguir persistencia.

J

JVM: **J**ava **V**irtual **M**achine.

M

Micronaut: Framework para el desarrollo de microservicios basado en JVM.

Microservicios: Tipo de arquitectura software que descompone sistemas más grandes y complejos en servicios más pequeños independientes que se comunican entre ellos.

MongoDB: Base de dato de tipo NoSQL que almacena documentos en formato JSON.

Monolito: Tipo de arquitectura software tradicional en la cual todas las partes de la aplicación se encuentran juntas en un mismo programa sobre una misma plataforma.

N

No-bloqueante: Dicho de un proceso que no se bloquea ni espera por la ejecución de otras tareas.

P

PostgreSQL: Base de datos relacional de código abierto.

Project Reactor: Biblioteca reactiva disponible en Java que implementa la especificación de flujos reactivos.

R

RxJava: Biblioteca reactiva que surge como implementación de ReactiveX para la JVM.

S

Spring: Framework para el desarrollo de aplicaciones basado en JVM.

Sistemas reactivos: Dicho de sistemas sensibles, resistentes, elásticos y basados en el intercambio de mensajes.

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE PARA LA WEB

Trabajo Fin de Máster

ESTUDIO DEL FRAMEWORK MICRONAUT PARA EL DESARROLLO DE MICROSERVICIOS REACTIVOS

Justin Hernández Jover

2022