Trabajo Fin de Master

Estudio del framework Micronaut

Máster Universitario en Desarrollo Ágil de Software para la Web

Presentado por:
D. Santiago Tamariz – Martel Marco

Dirigido por:
D. Salvador Otón Tortosa

Alcalá de Henares, a 21 de Julio de 2022

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

Máster en Desarrollo Ágil de Software para la Web

Trabajo Fin de Máster

Estudio del framework Micronaut

Autor: Santiago Tamariz-Martel Marco
Director: Salvador Otón Tortosa

TRIBUNAL:

Presidente:

Vocal 1º:

Vocal 2º:

CALIFICACIÓN: _____ **FECHA:**


Universidad
de Alcalá

Quisiera agradecer sencillamente a las personas que me han acompañado durante esta etapa, en especial a mi madre y a mi padre. A mis amigos por ayudarme a desconectar y, por último, gracias abuela por poner siempre una velita en las situaciones más difíciles

Índice Resumido

Índice Detallado

Índice De Figuras

1. Introducción

1.1 Motivación

En la actualidad, el mundo de los frameworks[5] está altamente explorado, sin embargo, no compartido, es decir, muchos frameworks quedan sin conocer.
Existen numerosos frameworks tanto para el desarrollo back-end como para el desarrollo front-end. Estos frameworks facilitan principalmente la configuración de la estructura de trabajo, ofreciendo arquitecturas definidas, librerías operativas e incluso conexiones con bases de datos auto configurables.

Dada la gran cantidad de frameworks que existen, resulta difícil conocerlos todos. Además, una vez que un desarrollador se familiariza con un framework en particular, tiende a centrarse en trabajar exclusivamente con ese, independientemente de que aparezcan nuevos más útiles para su labor. Esta tendencia es natural, causada por la comodidad y la costumbre. No obstante, es importante estar abiertos a explorar y utilizar nuevas opciones aprovechando las innovaciones de la actualidad.

La motivación principal de este Trabajo de Fin de Máster (TFM) consiste en dar a conocer un framework de desarrollo back-end que ofrece una amplia gama de funcionalidades y que puede resolver las necesidades de los desarrolladores. En particular, también tiene como propósito mostrar la facilidad de implementación de este framework y los beneficios que tiene y aporta en comparación con otros más antiguos y conocidos.

Al dar a conocer el framework Micronaut[13], se espera que los desarrolladores adquieran un mayor grado de conocimiento sobre sus características, sus implementaciones y sus capacidades para cumplir objetivos e intereses en el desarrollo de aplicaciones. Además, este TFM busca fomentar la adopción de Micronaut como una alternativa viable y moderna, promoviendo así la innovación en el campo de desarrollo back-end y fomentando la creación de aplicaciones más eficientes y escalables.

1.2 Objetivo

El objetivo principal de este TFM titulado “Estudio del Framework Micronaut”, es realizar un estudio, análisis y evaluación de Micronaut como framework de desarrollo de aplicaciones web.

Los objetivos específicos son:

- Generar el interés entre el público objetivo, concretamente desarrolladores, en el uso y adopción del framework Micronaut.
- Proporcionar un conocimiento profundo del framework, profundizando en el funcionamiento interno, explorando su arquitectura, principios de diseño y componentes clave.
- Mostrar un ejemplo práctico con un caso de estudio que demostrará la implementación de Micronaut y las facilidades que ofrece.

Con la consecución de estos objetivos, este TFM pretende no sólo aumentar el conocimiento y la comprensión del framework Micronaut, sino también animar a los desarrolladores a adoptar Micronaut como una alternativa moderna y eficiente para sus necesidades de desarrollo back-end.

1.3 Estructura

Este TFM se divide en siete capítulos diferenciados.

En este primer capítulo, “Introducción”, se ha presentado la motivación y objetivos de este TFM, estableciendo el contexto y la importancia del estudio del framework.

En el segundo capítulo, “Estado del arte”, el lenguaje de programación java es introducido, se proporciona una visión general del panorama actual de frameworks y se realiza un análisis de otro framework popular, Spring Boot.

El tercer capítulo, "Micronaut", explora los principios fundamentales y la filosofía de diseño de Micronaut. También se analizarán los componentes arquitectónicos, características específicas y las ventajas ofrecidas por la inyección de dependencias.

El cuarto capítulo, “Caso de estudio Micronaut”, presenta un caso práctico que demuestra la aplicación de Micronaut en una aplicación web real. Se muestra el proyecto junto con un imágenes demostrativas.

En el quinto capítulo, “Conclusiones”, se incluyen las conclusiones obtenidas por el autor de este TFM y se ofrecen recomendaciones para futuros trabajos y aplicaciones de Micronaut.

Finalmente, se recoge en el último capítulo la bibliografía con los recursos utilizados para el desarrollo de este TFM.

2. Estado del Arte

En este apartado se van a dar a conocer los conceptos teóricos para la compresión del trabajo y con carácter introductorio para en el siguiente apartado profundizar más en las posibilidades que ofrece el framework Micronaut.

En primer lugar, se introduce el lenguaje de programación Java, tras ello se introduce el concepto teórico de los frameworks y para finalizar se analiza Spring Framework como uno de los frameworks más populares en la actualidad.

2.1 Jakarta EE

Jakarta EE[9] es un conjunto de componentes de software y APIs diseñados específicamente para el desarrollo de aplicaciones empresariales en Java. Estas APIs se conocen como especificaciones, que amplían Java SE (Java Standard Edition) para adaptarse a las necesidades de las aplicaciones empresariales.

Los beneficios que ofrece Jakarta EE incluyen la capacidad de crear aplicaciones escalables, seguras y capaces de manejar grandes volúmenes de datos. Estas especificaciones también ayudan a acelerar el desarrollo, reducir la complejidad y mejorar el rendimiento de las aplicaciones.

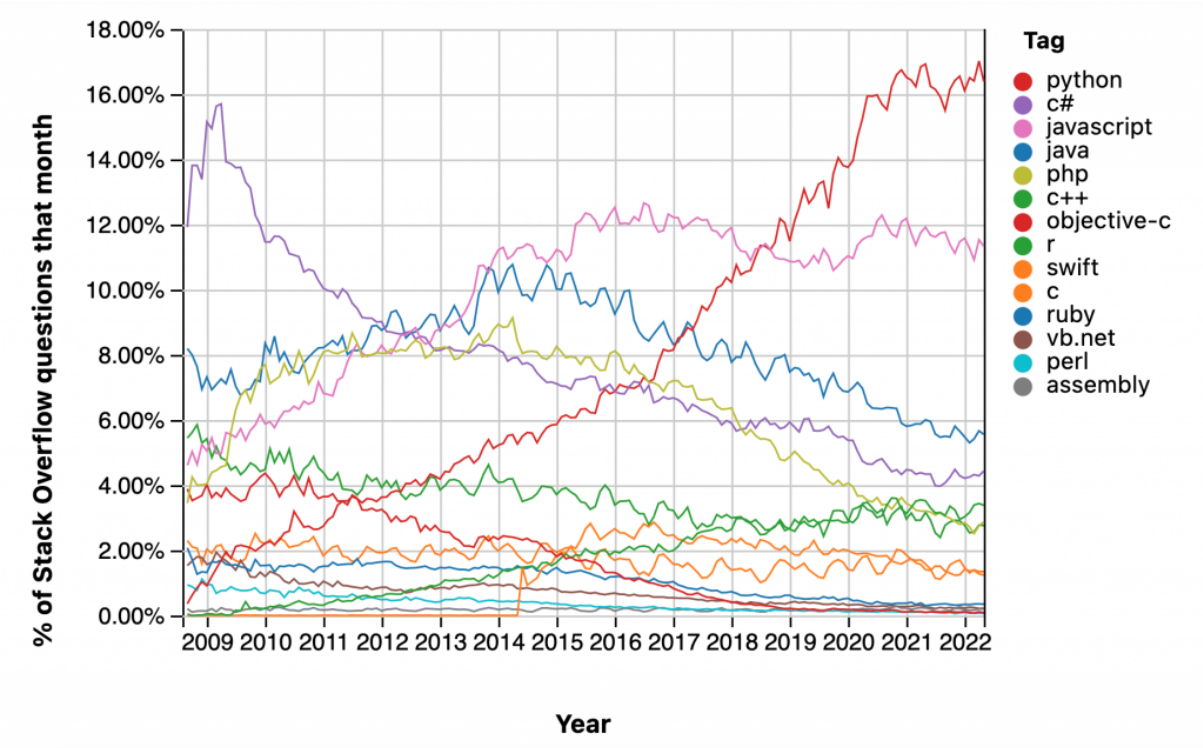

Ilustración 1: Porcentaje de preguntas en Stack Overflow en referencia al lenguaje de programación

Java[10] es un lenguaje de programación de alto nivel, orientado a objetos y seguro. Fue desarrollado por Jame Gosling en Sun Microsystems, Inc en 1991 y posteriormente adquirido por Oracle.

Las características de Java son:

- Sencillez: Java tiene una sintaxis simple y legible, lo que facilita su comprensión. Además, elimina conceptos complejos presentes en otros lenguajes como los punteros de C++. Java proporciona una manera sencilla de implementar funciones y enfocarse en las tareas necesarias.

- Orientación a Objetos: En Java, todo es un objeto, posee unos datos y un comportamiento. Para ejecutar programas Java se debe tener al menos una clase y un objeto.

- Robustez: Java comprueba los errores en tiempo de ejecución y en tiempo de compilación. Utiliza un recolector de basura (garbage collector) para la gestión automática de memoria y ofrece un sólido sistema de manejo de excepciones.

- Seguridad: Es un lenguaje seguro ya que se ejecutan los programas en la máquina virtual (JVM) y que contiene un gestor de seguridad que define el acceso de las clases Java.

Conceptos básicos de Java:

- Portabilidad: El código escrito en Java puede ejecutarse en cualquier plataforma, sin depender de características específicas de implantación.

- Alto rendimiento: Ofrece un alto rendimiento mediante el uso del compilador Just-In-Time (JIT).

- Multihilos: Java admite la ejecución simultánea de múltiples hilos, lo que permite que varios procesos se ejecuten de manera concurrente.

- Distribuido: Java está diseñado para entornos distribuidos, como internet, y es compatible con el protocolo TCP/IP. Puede funcionar a través de la red y comunicarse con otros sistemas de forma eficiente.

En 2018, Java EE recibió el nombre de Jakarta EE[11] después de que Oracle decidiera donar las especificaciones, el desarrollo y la gestión a la Fundación Eclipse. Sin embargo, Oracle retuvo la propiedad de la marca registrada, por lo que la fundación tuvo que elegir un nuevo nombre. A través de una encuesta realizada por la fundación, “Jakarta EE” se convirtió en el nombre ganador.

Jakarta EE se considera un estándar ampliamente reconocido para el desarrollo web. Proporciona un conjunto completo de técnicas para implementar los aspectos de las aplicaciones empresariales, como las solicitudes web, acceder a bases de datos, integrarse con otros sistemas e implementar servicios web.

Además, Jakarta EE ofrece un sólido ecosistema de bibliotecas, frameworks, como el que se va a estudiar, y herramientas que respaldan el desarrollo de aplicaciones empresariales. Promueve una arquitectura basada en componentes, lo que permite a los desarrolladores construir componentes de software modulares y reutilizables.

En general, Jakarta EE desempeña un papel fundamental al permitir a los desarrolladores construir aplicaciones web robustas y escalables, utilizando APIs y tecnologías estandarizadas. Su adopción continúa creciendo, impulsada por comunidad de Jakarta EE y su compromiso de ofrecer soluciones innovadoras para el desarrollo empresarial.

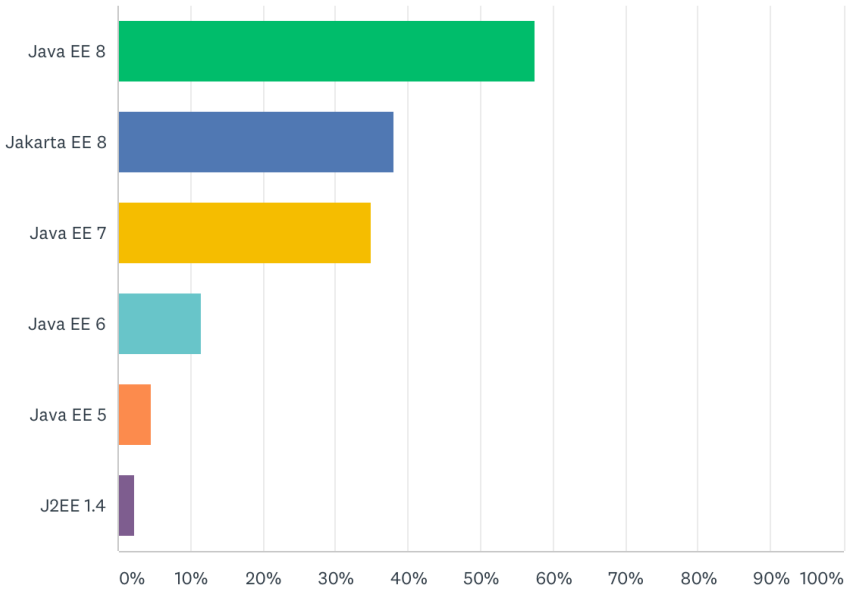

Ilustración 2: Versión de Java EE más usada

2.2 Frameworks

Un Framework[5], es un código software pre-desarrollado que proporciona estructura y funcionalidad para el desarrollo de software. Se utiliza como base para el desarrollo de proyectos software de gran tamaño, permitiendo a los desarrolladores centrarse únicamente en la creación de código para la lógica de negocio y funcionalidades.

En lugar de repetir la estructuración del código repetidamente para el mismo tipo de aplicaciones, un framework proporciona una forma estandarizada de trabajar y las funciones necesarias para que la aplicación funcione.

La mayoría de los frameworks siguen una estructura MVC (Modelo-Vista-Controlador). Esta arquitectura separa los componentes de una aplicación en tres grupos o capas diferenciadas.

La capa del modelo es la capa que contiene los datos de la aplicación. Incluidos en ellos los datos que van a manejarse, los mecanismos de persistencia para la base de datos y la lógica de negocio.

La capa de la vista es la responsable de generar la interfaz de la aplicación. También es conocida como la interfaz gráfica que va a manejar el usuario y que muestra una parte de los datos e interactúa con él.

La capa del controlador actúa como intermediaria entre la vista y el modelo, gestiona las notificaciones de los cambios de estado y el flujo de información. Es la encargada de recoger peticiones del cliente por comunicación HTTP y de enviar respuestas.

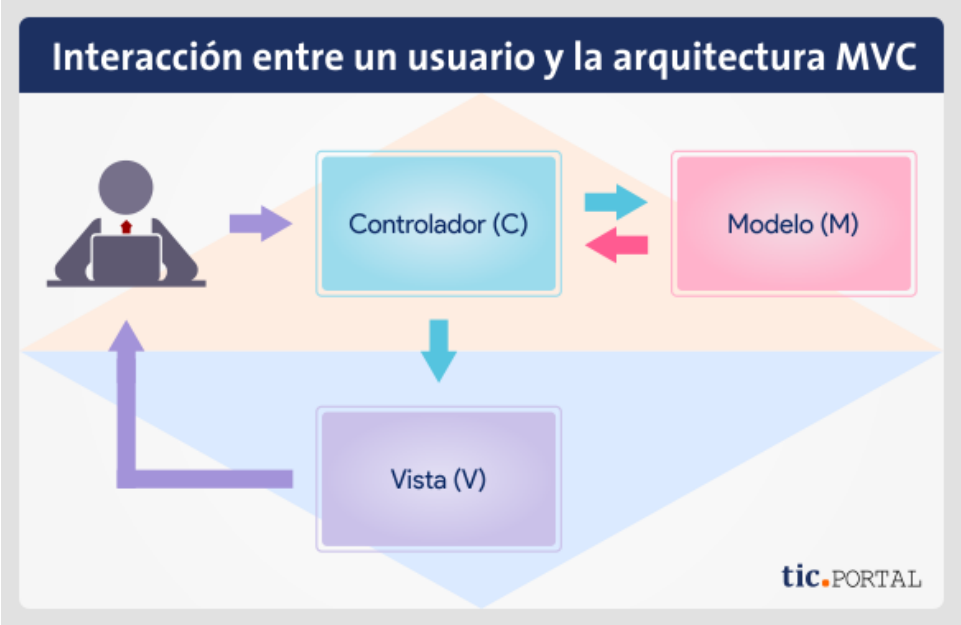

Ilustración 3: Modelo Vista Controlador

Los frameworks son desarrollados y probados por múltiples desarrolladores, aumentando así su fiabilidad y seguridad. Gracias a que proporcionan un arquetipo del proyecto, los desarrolladores les dan uso para ahorrar tiempo en el desarrollo.

Ventajas de usar frameworks:

- Proporciona funcionalidad integrada, estructura y una forma unificada de trabajar.
- Facilita un desarrollo más rápido y rentable con menos esfuerzo.
- Evita duplicidad de código.
- Reduce la cantidad de errores y ayuda a resolverlos más fácilmente.

Desventajas de los frameworks:

- Tienen una curva de aprendizaje pronunciada.
- El tamaño del programa se ve aumentado por su uso.
- Tienen impuesta una forma específica de trabajar reduciendo la flexibilidad para el desarrollo.
- Para aplicaciones de pequeño tamaño no merecen la pena ya que el coste de usar el framework es mayor que desarrollarlo desde cero.

En general, los frameworks ofrecen numerosas ventajas como un desarrollo rápido, reutilización de código y una amplia comunidad que se apoya, pero también supone un reto en cuanto a curva de aprendizaje y adhesión a los estándares definidos.

2.3 Spring

Spring es un framework que ofrece un modelo de configuración y programación para el desarrollo de aplicaciones, principalmente web, basando el lenguaje de programación Java.

Actualmente, es muy popular debido a la posibilidades que ofrece, incluyendo las herramientas incorporadas y la agilidad que proporciona para el desarrollo y despliegue de aplicaciones web.

Spring Framework tuvo un gran impulso gracias al uso de la inyección de dependencias y al contenedor de inversión de control. Estos conceptos, presentes en el framework Micronaut, se analizarán en las siguientes secciones.

Spring facilita la configuración de la estructura de trabajo, ofreciendo una arquitectura definida para que el equipo de programación se centre únicamente en el desarrollo de la lógica de negocio.

Spring Framework[33] está dividido en proyectos y módulos que cubren diversas necesidades, desde la configuración y la seguridad hasta las aplicaciones web y el big data. Existe un proyecto Spring que facilita el desarrollo en cada caso.

Ilustración 4: Spring projects

Dado el alto número de proyectos disponibles, se analizarán los más utilizados:

- Spring Boot: Simplifica la creación de aplicaciones basadas en el framework Spring, proporcionando una instalación sencilla y una configuración mínima. Spring Boot automatiza la gestión de dependencias con Maven y gradle, así como la configuración del despliegue en servidores, utilizando Apache Tomcat como servidor incorporado.

- Spring Framework: Proporciona un modelo completo de programación y configuración para cualquier plataforma de despliegue, evitando la necesidad de configuración específica para cada entorno de implementación.

- Spring Data: Facilita el acceso a los datos mediante un modelo de programación que soporta tecnologías de acceso a bases de datos y servicios en la nube.

- Spring Cloud: Proporciona herramientas para que los desarrolladores implementen patrones de sistemas distribuidos poniendo en marcha rápidamente servicios y aplicaciones que los implementen.

- Spring Cloud Data Flow: Permite el procesamiento de datos en batch y en streaming basado en microservicios para Cloud Foundry y Kubernetes. Proporciona herramientas para crear topologías de pipelines de streaming y batch, admitiendo una amplia gama de casos de uso de procesamiento de datos como ETL y análisis predictivo.

- Spring Security: Es un framework para la autenticación y control de acceso personalizable. Es el estándar para la seguridad de las aplicaciones con Spring.

- Spring HATEOAS: Proporciona APIs para facilitar la creación de representaciones REST que siguen el principio HATEOAS cuando se trabaja con Spring, especialmente con Spring MVC. Su objetivo principal es la creación de enlaces y el ensamblaje de representaciones.

- Spring Batch: Ofrece funciones reutilizables para procesar grandes volúmenes de registros en sistemas empresariales. Estas funciones incluyen registro, gestión de transacciones, estadísticas etc. Además, proporciona servicios y funciones avanzadas para optimizar y particionar los batch Jobs.

- Spring Web MVC: Implementa la arquitectura MVC (Model View Controller) para crear aplicaciones web. Este módulo separa, como indica el patrón MVC[4], el código de la vista y los componentes que implementan el controlador y el modelo de datos. Está basado principalmente en servlets[37] que proveen herramientas para procesar información de aplicaciones web y servicios.

- Spring Web Flow: Extensión del modulo Spring Web MVC. Este módulo además de los controladores implementados por el módulo Spring MVC, ayuda a definir el archivo XML (Extensible Markup Language)[38] o clase Java que permitirá gestionar el flujo de trabajo.

- Spring Web Services: Framework para crear servicios basados en documentos, centrado en el desarrollo de servicios SOAP (Simple Object Access Protocol)[30] basados en contratos. Permite la creación de servicios web flexibles con cargas útiles de XML.

3. Micronaut Framework

Micronaut[13] es un framework moderno de código abierto basado en la JVM[8] (Java Virtual Machine) que soporta Java, Groovy y Kotlin. Fue desarrollado en mayo de 2018 por los creadores de Grails, aprovechando su experiencia en la migración de aplicaciones monolíticas a microservicios utilizando Spring[32], Spring Boot y Grails[6].

El auge de la arquitectura de microservicios[27], las aplicaciones nativas en la nube y la computación sin servidor han generado la necesidad de frameworks que puedan abordar estos desafíos de forma rápida y eficiente. Muchos frameworks diseñados para aplicaciones monolíticas[1], como Grails y Spring, intentaron adaptarse, pero se quedaron cortos en términos de arranque y consumo de recursos.

Para ello, se creó Micronaut. Diseñado desde cero con un enfoque en el desarrollo de microservicios, su objetivo es minimizar el uso de memoria y mejorar los tiempos de arranque. Para obtener esto, hace uso de enfoques innovadores como la compilación anticipada (AOT)[16] y la inyección de dependencias (DI)[14][7].

El consumo de memoria y el tiempo de arranque de Micronaut no incrementa de forma lineal con el número de líneas de código. Esto se debe a que el procesamiento de las anotaciones, el comportamiento de la AOP[12] (Aspect Oriented Programming) y la generación de proxys sucede en tiempo de compilación utilizando los procesadores AST[2] (Abstract Syntax Tree), generando metadatos que luego son optimizados por el compilador.

La naturaleza ligera y eficiente de Micronaut permite a los desarrolladores crear aplicaciones robustas y escalables minimizando el uso de recursos. Al adoptar una filosofía de diseño que prioriza la eficiencia y la mínima sobrecarga, Micronaut proporciona un framework moderno y eficaz para crear aplicaciones Java de alto rendimiento.

Los objetivos de Micronaut, tal y como fue creado y ha sido actualizado, incluyen bajo consumo de memoria, inyección de dependencias, AOP, arranque casi instantáneo con archivos JAR de pequeño tamaño, IoC y uso mínimo de proxies. Además, su objetivo es tener dependencias externas mínimas y permitir pruebas unitarias sencillas.

Micronaut se adhiere al manifestó Twelve-Factor[36], que indica los puntos fundamentales a cumplir para desarrollar una aplicación que se distribuya como servicio.

En las siguientes secciones, se profundizará en las características y funcionalidades principales de Micronaut, incluyendo su mecanismo de inyección de dependencias, compilación anticipada, soporte para programación reactiva y su idoneidad para arquitecturas de microservicios. También exploraremos los componentes arquitectónicos clave que forman la base del framework y una comparación con Spring Framework.

3.1 Características principales

3.1.1 Inversión de Control (IOC)

El contenedor de IoC en Micronaut juega un papel crucial para lograr su arranque rápido y ligero. Su objetivo es crear y gestionar objetos inyectando en ellos dependencias previamente declaradas. Estos objetos, denominados beans, se crean en base a la definición proporcionada. Los beans pueden definirse anotando una clase como bean o creando factory methods. De este modo, el control de la inicialización de los objetos está bajo control del contenedor.

El contenedor de Micronaut implementa la especificación JSR-330[15], que es el estándar para la inyección de dependencias en Java. Aprovecha la compilación Ahead-Of-Time (AOT) para minimizar el uso de la reflexión y para reducir el tiempo de inicio de la aplicación. Gracias a este enfoque, Micronaut es muy ligero y compatible con la compilación nativa.

Los objetivos del contenedor IoC de Micronaut son:

- Utilizar la reflexión como último recurso: La intención principal de Micronaut es minimizar el uso de reflexión, ya que esta puede tener un impacto negativo en el rendimiento. En su lugar, se favorece el análisis estático y la compilación AOT para resolver dependencias y conectar objetos

- Evitar los proxies: A diferencia de otros frameworks, Micronaut evita la creación de proxies dinámicos para gestionar las dependencias. Con esto, se elimina la sobrecarga asociada a la creación de estos y se mejora el rendimiento.

- Optimizar los tiempos de arranque: Gracias a la compilación AOT, el contendor puede realizar la resolución de dependencias y la inicialización de los objetos de forma eficiente optimizando los tiempos de arranque de la aplicación.

- Reducir la huella de memoria: Con todos los objetivos anteriores, utilizando AOT y evitando proxies innecesarios, se optimiza el uso de memoria, lo que hace adecuado que sea adecuado para entornos con recursos limitados.

- Proporcionar un manejo de errores claro y comprensible: El contenedor IoC de Micronaut está diseñado para proporcionar mensajes de error claros y compresibles cuando hay problemas con la resolución de dependencias o la creación de objetos. Con esto se ayuda a la resolución de problemas y hace que sea más fácil identificar y resolver los problemas durante el desarrollo.

En general, el contenedor IoC de Micronaut desempeña un papel fundamental a la hora de permitir una inyección de dependencias eficiente y gestionar los ciclos de vida de los objetos, contribuyendo a las características de ligereza y alto rendimiento del framework.

3.1.2 Compilación Ahead of Time (AOT)

La compilación anticipada en Micronaut implica la compilación de clases Java en código nativo para posteriores ejecuciones. El compilador genera dinámicamente código nativo mientras se ejecuta una aplicación y copia cualquier código generado por AOT[3] en la memoria caché de datos compartidos. Las máquinas virtuales Java (JVM) que ejecuten el método pueden cargar y utilizar el código AOT de la caché de datos sin la sobrecarga de rendimiento asociada al código nativo compilado justo a tiempo (JIT).

El compilador AOT está inhabilitado de forma predeterminada en las máquinas virtuales java. Sin embargo, en Micronaut, la compilación AOT está activada, lo que aporta numerosas ventajas.

La compilación AOT reduce el tiempo de inicio de la aplicación y el tamaño del despliegue al realizar una serie de operaciones durante la compilación. Puede pre-calcular los requisitos de los beans y realizar sustituciones en tiempo de compilación, garantizando que sólo se incluyan las clases necesarias en el entorno de producción.

Con Micronaut AOT, pueden beneficiarse tanto las aplicaciones Micronaut normales como las imágenes GraalVM nativas de las aplicaciones Micronaut. Esta mejora se puede observar en la siguiente gráfica, un aumento del 26% en la velocidad del tiempo de arranque para un jar JIT y un aumento del 46% para una imagen GraalVM nativa.

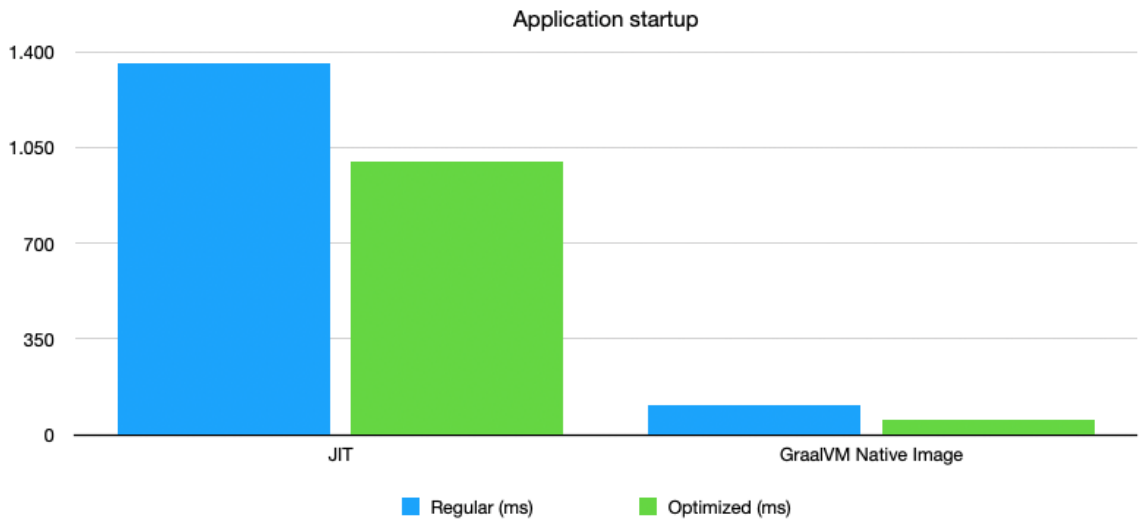

Ilustración 5: Tiempo de arranque JIT vs GraalVM

La función de compilación AOT de Micronaut tiene un papel muy importante en la optimización del rendimiento de las aplicaciones, la reducción del tiempo de arranque y la minimización del consumo de recursos. Con el uso de AOT, los desarrolladores de software tienen la posibilidad de conseguir una ejecución más rápida y eficiente de sus aplicaciones, lo que es un beneficio ideal para construir sistemas de alto rendimiento y escalables.

3.1.3 Programación Reactiva

Micronaut soporta programación reactiva[21], tanto para clientes como para servidores, implementando Reactive Streams[29], haciendo posible el uso de cualquier librería que lo soporte, como RxJava2, Reactor o Akka. Además, ofrece muchas otras ventajas que lo hacen muy útil para la programación reactiva.

Las ventajas que ofrece Micronaut para la programación reactiva incluyen:

- Soporte integrado para la programación reactiva[31]: Micronaut tiene un soporte incorporado para programación reactiva, lo que facilita la construcción de microservicios reactivos.

- Clientes HTTP reactivos declarativos[22]: Micronaut permite la construcción de clientes HTTP reactivos declarativos que se implementan en tiempo de compilación, reduciendo significativamente el consumo de memoria.

- Bajo consumo de memoria[17]: Con su huella de memoria ligera, Micronaut solo necesita 8MB de heap para ejecutarse, manteniendo el tiempo de inicio y la huella de memoria bajos, al tiempo que aprovecha todo el potencial de la programación reactiva.

- Compatible con la implementación de Netty: Micronaut permite el uso de Netty, un framework Java para desarrollar aplicaciones cliente-servidor asíncronas y de alto rendimiento.

- Drivers para conexiones reactivas a bases de datos: Micronaut proporciona compatibilidad con drivers que implementan flujos reactivos en Java, ofreciendo procesamiento asíncrono no bloqueante para bases de datos. Algunos ejemplos son Micronaut Data Hibernate Reactive para bases de datos relacionales y Micronaut Data MongoDB para bases de datos no relacionales.

- Integración con bibliotecas de programación reactiva: Micronaut es compatible con cualquier framework que implemente el estándar Reactive Streams. Permite a los desarrolladores el uso de ReactiveX o Reactor dentro del framework para crear aplicaciones reactivas.

En general, Micronaut proporciona un buen soporte para la programación reactiva, incluyendo clientes HTTP declarativos, bajo consumo de memoria, integración con librerías reactivas y compatibilidad con conexiones a bases de datos reactivas. Estas características hacen que Micronaut sea una valiosa herramienta para crear aplicaciones reactivas eficientes y escalables.

A continuación, se va a mostrar un ejemplo del uso de la programación reactiva en Micronaut[23]. Para ello se utilizará RxJava.

En primer lugar, es necesario configurar las dependencias, el soporte para programación reactiva con RxJava está habilitado por defecto y la dependencia io.reactivex.rxjava2:rxjava se incluye junto con las librerías de core de Micronaut. En todo caso, las dependencias necesarias para el ejemplo son las siguientes:

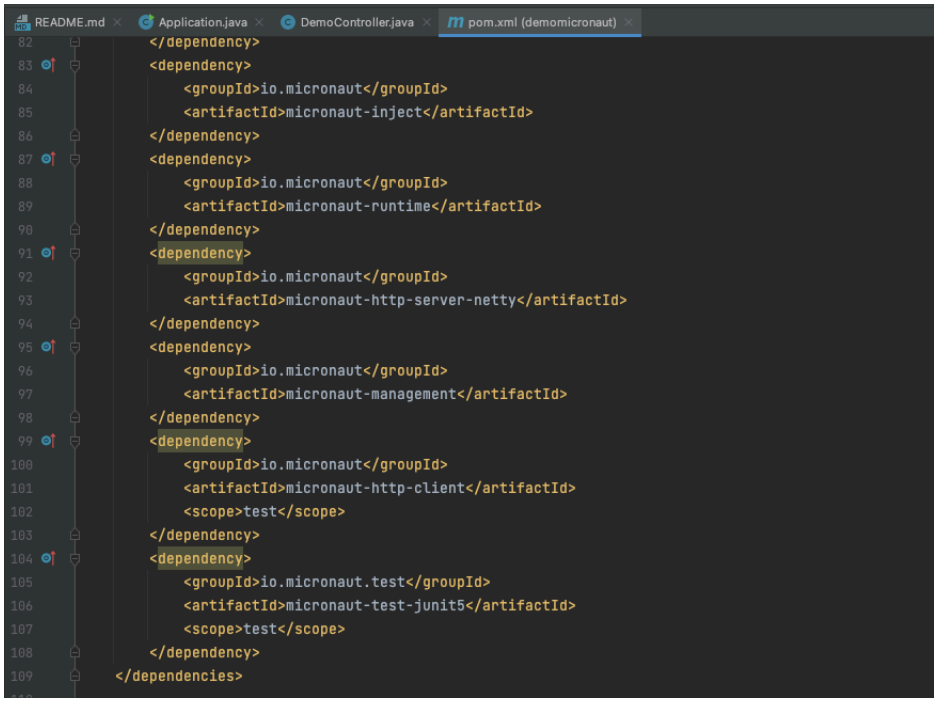

Ilustración 6: Dependencias programación reactiva

El siguiente paso será empezar con la aplicación por el lado del servidor, por lo que se va a implementar un controlador. Este controlador utiliza los objetos RxJava en las sentencias return: Single, Maybe y Flowable. El resto de la implementación es la común para un controlador REST.

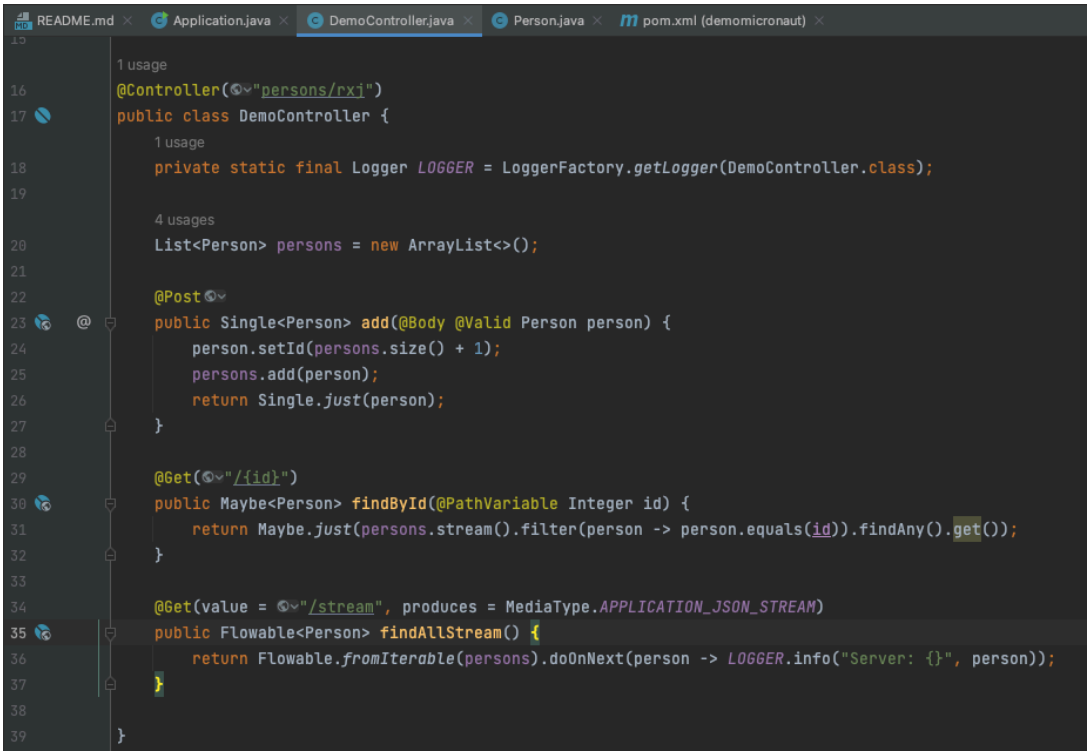

Ilustración 7: Implementación controlador reactivo

En la implementación del controlador se han usado 3 tipos observables de RxJava2:

1. Single: Emite un único elemento y luego se completa. Garantiza que se emitirá al menos un elemento, por lo que es adecuado para salidas no vacías

2. Maybe: Es similar al Single, pero puede completar sin emitir un valor. Es útil para emitir opcionales.

3. Flowable: Emite un flujo de elementos y soporta mecanismo de contrapresión, permitiendo el control sobre la tasa de emisión.

Los métodos que se han definido son:

- add: Para añadir un nuevo elemento persona a la lista.
- findById: Busca el elemento persona por el ID, podrá devolver ningún valor en caso de no existir.
- findAllStream: Emite todos los elementos como un stream. Para aprovechar las ventajas de los flujos reactivos en el lado del cliente, este método produce un tipo de contenido application/x-json-stream. Utilizando este tipo de contenido, los eventos son recuperados continuamente por el cliente HTTP mediante el uso del tipo Flowable.

Para acceder a las APIs, Micronaut ofrece dos tipos de clientes, el cliente de bajo nivel y el cliente declarativo. Se podrá elegir entre HttpClient, RxHttpClient y RxStreamingHttpClient con soporte para streaming de datos con HTTP.

La forma recomendada para acceder a la referencia de un cliente es inyectándolo con la anotación @Client. Sin embargo, se va a mostrar cómo se haría la inyección del cliente de bajo nivel dentro del test Junit del método "add".

Para leer Single o Maybe es necesario utilizar el método "retrieve" de RxHttpClient y tras subscribirse a un observable se utiliza la librería ConcurrentUnit para manejar resultados asíncronos en el test.

```java
public DemomicronautTest(EmbeddedServer server, @Client("/") Rx3HttpClient client) {
    this.server = server;
    this.client = client;
}

@Test
public void testAdd() throws MalformedURLException, TimeoutException, InterruptedException {
    final Waiter waiter = new Waiter();
    final Person person = new Person( id: 1,  name: "Name100",  surname: "Surname100",  age: 22, Gender.MALE);
    Single<Person> s = client.retrieve(HttpRequest.POST( uri: "/persons/reactive", person), Person.class).firstOrError();
    s.subscribe(person1 -> {
        LOGGER.info("Retrieved: {}", person1);
        waiter.assertNotNull(person1);
        waiter.assertNotNull(person1.getId());
        waiter.resume();
    });
    waiter.await( delay: 3000, TimeUnit.MILLISECONDS);
}
```

Ilustración 8: Test cliente bajo nivel

Además del cliente HTTP de bajo nivel, Micronaut permite crear clientes declarativos a través de la anotación @Client. Para ello, únicamente es necesario crear una interfaz con métodos similares a los métodos definidos dentro del controlador y anotarlos correctamente.

```java
1 usage
@Client("/persons/reactive")
public interface PersonReactiveClient {

    @Post
    Single<Person> add(@Body Person person);

    @Get("/{id}")
    Maybe<Person> findById(@PathVariable Integer id);

    @Get(value = "/stream", produces = MediaType.APPLICATION_JSON_STREAM)
    Flowable<Person> findAllStream();

}
```

Ilustración 9: Interfaz @Client

Una vez definida la interfaz, se inyectará en los test y se probará el método “add”.

```
1 usage
@Inject
PersonReactiveClient client;

@Test
public void testAddDeclarative() throws TimeoutException, InterruptedException {
    final Waiter waiter = new Waiter();
    final Person person = new Person( id: 1,  name: "Name100",  surname: "Surname100",  age: 22, Gender.MALE);
    Single<Person> s = client.add(person);
    s.subscribe(person1 -> {
        LOGGER.info("Retrieved: {}", person1);
        waiter.assertNotNull(person1);
        waiter.assertNotNull(person1.getId());
        waiter.resume();
    });
    waiter.await( delay: 3000, TimeUnit.MILLISECONDS);
}
```

Ilustración 10: Test cliente declarativo

Adicionalmente, Micronaut es capaz de soportar backpressure.

La contrapresión es un concepto fundamental en la programación reactiva, que hace referencia a la resistencia que se opone al flujo fluido de datos. Es decir, si se genera eventos a un ritmo superior al que el consumidor puede manejar en un determinado plazo, el consumido debe tener la capacidad de regular la frecuencia de consumo de eventos en el lado del productor.

Para controlar esta contrapresión se implementan métodos adicionales al controlador usando el método fromCallable para emitir elementos dentro de un flujo Flowable. Al llamar al método “repeat” se están produciendo nueve elementos en total. Además, el método “findAllStreamWithCallableDelayed” introduce un retardo de 100 milisegundos para cada elemento emitido en el flujo.

Todo esto permite controlar la contrapresión en el lado del servidor ajustando la tasa de emisión de eventos.

```
@Get(value = "/stream/callable", produces = MediaType.APPLICATION_JSON_STREAM)
public Flowable<Person> findAllStreamWithCallable() {
    return Flowable.fromCallable(() -> {
                int r = new Random().nextInt( bound: 100);
                Person p = new Person(r,  name: "Name"+r,  surname: "Surname"+r, r, Gender.MALE);
                return p;
            }).doOnNext(person -> LOGGER.info("Server: {}", person))
            .repeat( times: 9);
}

@Get(value = "/stream/callable/delayed", produces = MediaType.APPLICATION_JSON_STREAM)
public Flowable<Person> findAllStreamWithCallableDelayed() {
    return Flowable.fromCallable(() -> {
                int r = new Random().nextInt( bound: 100);
                Person p = new Person(r,  name: "Name"+r,  surname: "Surname"+r, r, Gender.MALE);
                return p;
            }).doOnNext(person -> LOGGER.info("Server: {}", person))
            .repeat( times: 9).delay( time: 100, TimeUnit.MILLISECONDS);
}
```

Ilustración 11: Métodos para controlar Backpressure

3.1.4 Microservicios

Micronaut es un kit de herramientas de código abierto basado en JVM para construir microservicios modulares fácilmente comprobables y aplicaciones sin servidor.
Cuando se trata del desarrollo de microservicios, Micronaut ofrece una multitud de beneficios[24] que facilitan a los desarrolladores crear soluciones eficientes y escalables:

- Desarrollo eficiente: Micronaut permite a los desarrolladores crear aplicaciones de microservicios modulares que son fácilmente comprobables. Adopta el patrón MVC (véase el punto 2.2) y está diseñado para satisfacer las necesidades de los microservicios y la programación centrada en la nube. Esto agiliza los procesos de desarrollo y mejora la productividad.

- Bajo consumo de memoria e inicio rápido: Como se ha comentado a lo largo de todo el TFM, Micronaut destaca en el desarrollo de aplicaciones con una huella de memoria mínima y un tiempo de inicio rápido. Esto lo convierte en una buena opción para microservicios y aplicaciones sin servidor. La capacidad de Micronaut para optimizar el uso de memoria y reducir los tiempos de inicio es muy ventajosa para despliegues ligeros.

- Microservicios reactivos: Al soportar la programación reactiva, se facilita el desarrollo de microservicios reactivos.

- Integración con servicios en la nube: Micronaut se integra con servicios en la nube como la configuración distribuida, el descubrimiento de servicios y las funciones serverless. Esto permite incorporar características nativas de la nube en los microservicios.

- Modelo de programación sencillo: Micronaut ofrece un modelo de programación muy sencillo. Esta simplicidad hace que sea fácil de aprender y de utilizar, permitiendo a los desarrolladores adoptar el framework rápidamente para sus proyectos de microservicios.

Micronaut incorpora patrones de microservicios establecidos, eliminando la necesidad de combinar herramientas y frameworks. Su arquitectura modular, combinada con el soporte nativo para la programación reactiva, lo convierte en una opción versátil para varios tipos de aplicaciones.

Véase el punto 4 para un ejemplo de microservicios en el framework Micronaut.

3.1.5 Arranque ligero y rápido

A lo largo de todo el documento se ha remarcado la ligereza y rapidez de arranque de Micronaut. Esto es el resultado de varios factores clave que lo diferencian de frameworks Java tradicionales como Spring Framework. Las ventajas de rendimiento son debido a:

- Inyección de dependencias: Ya se ha profundizado en la DI de Micronaut, este enfoque reduce significativamente la cantidad de reflexión utilizada por el frameworks, lo que mejora el tiempo de arranque y reduce el consumo de memoria.

- Programación orientada a aspectos: Micronaut emplea el tiempo de compilación para aplicar aspectos directamente a las clases. A diferencia de los frameworks tradicionales que dependen de proxies en tiempo de ejecución, Micronaut elimina la sobrecarga asociada a la creación de estos proxies y a la interceptación de métodos.

- Compatibilidad con GraalVM: Micronaut es compatible con GraalVM, una máquina virtual universal que permite la compilación de aplicaciones Java en código máquina nativo. Cuando se combina con GraalVM, Micronaut puede lograr tiempos de inicio más rápidos, reduciéndolos normalmente alrededor de 1 segundo a aproximadamente 20 milisegundos. Además, el consumo de memoria se reduce significativamente, lo que permite una utilización más eficiente de los recursos.

- Flexibilidad de configuración: Micronaut proporciona a los desarrolladores flexibilidad en la configuración del servidor HTTP. Los desarrolladores tienen el control sobre el lugar desde el que su aplicación carga la configuración, lo que permite una personalización basada en requisitos de despliegue específicos.

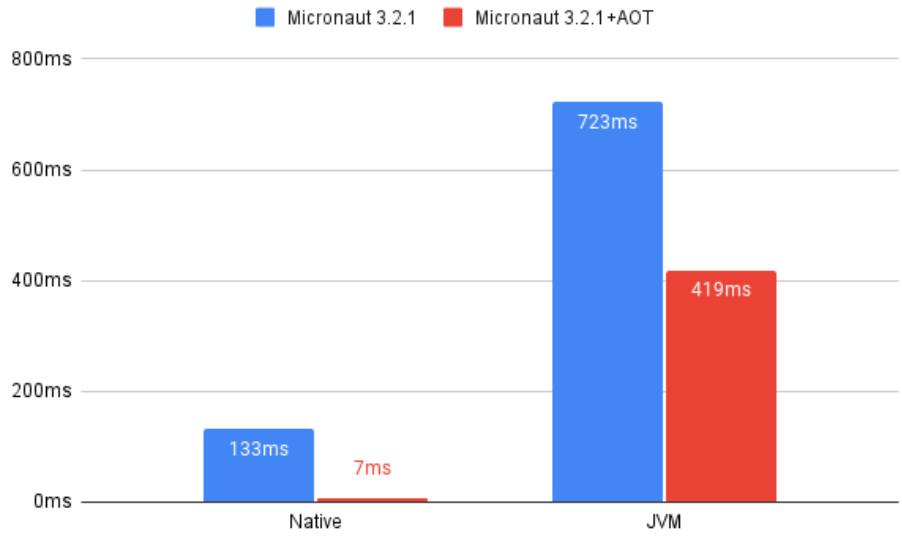

Ilustración 12: Tiempo de inicio GraalVM Native vs JVM

En resumen, la filosofía de diseño de Micronaut gira en torno a minimizar la dependencia de reflexión, evitar el uso de proxies, optimizar el tiempo de arranque, reducir la huella de memoria y garantizar una gestión de errores clara y manejable. Con la incorporación de estas estrategias, Micronaut es capaz de ofrecer un framework ligero y de alto rendimiento.

3.2 Comparación con otros frameworks

Dado el alto número de frameworks que existen en la actualidad, cada uno tiene unos beneficios e inconvenientes diferentes, por lo que se ha reflejado en aspectos generales las ventajas y desventajas de Micronaut. Estas ventajas son las que lo hacen tan valioso para el desarrollador y son las que se vienen explicando en los puntos anteriores.

Ventajas	Desventajas
Arranque ligero y rápido	Curva de aprendizaje
Desarrollo eficiente	Comunidad limitada
Soporte programación reactiva	Ecosistema pequeño y menos recursos
Capacidades nativas de la nube	
Inyección dependencias tiempo compilación	

En las siguientes gráficas se muestra la memoria consumida por cada diferente framework, es importante reflejar la poca memoria consumida por Micronaut durante el tiempo de ejecución.

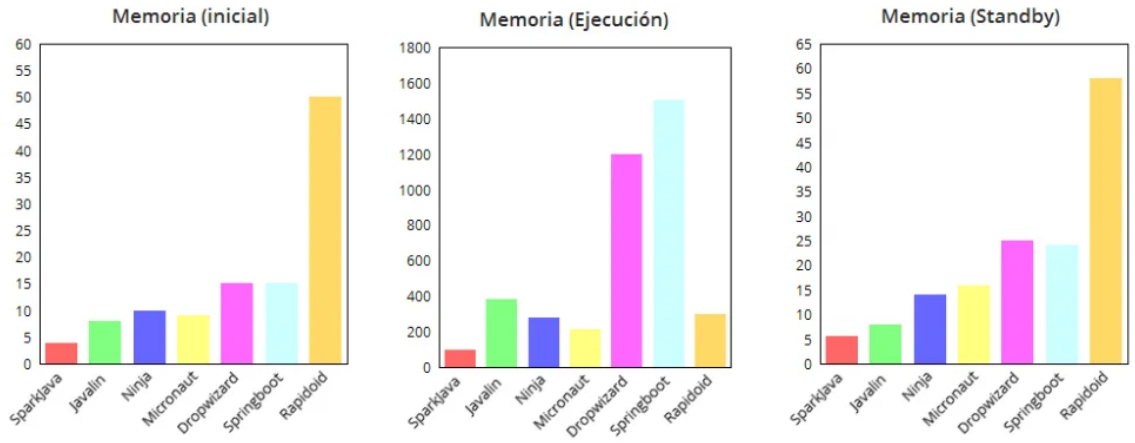

Ilustración 13: Consumo de memoria frameworks

El equipo de Desarrollo de Micronaut ha realizado una comparación[25] de Micronaut 2.0, Quarkus 1.3.1 y Spring Boot 2.3, todos ellos bajo las mismas condiciones. Se utilizó el mismo ordenador y se configuraron los frameworks para que actuasen con las mismas capacidades.

OpenJDK 14 on 2019 iMac Pro Xeon 8 Core. Winner in Red.

METRIC	MICRONAUT 2.0 M2	QUARKUS 1.3.1	SPRING 2.3 M3
Compile Time ./mvn clean compile	1.48s	1.45s	1.33s
Test Time ./mvn test	4.3s	5.8s	7.2s
Start Time Dev Mode	420ms	866ms (1)	920ms
Start Time Production java -jar myjar.jar	510ms	655ms	1.24s
Time to First Response	960ms	890ms	1.85s
Requests Per Second (2)	79k req/sec	75k req/sec	??? (3)
Request Per Second -Xmx18m	50k req/sec	46k req/sec	??? (3)
Memory Consumption After Load Test (-Xmx128m) (4)	290MB	390MB	480MB
Memory Consumption After Load Test (-Xmx18m) (4)	249MB	340MB	430MB

Ilustración 14: Micronaut vs Quarkus vs Spring Boot

Con estos resultados, se puede observar que Quarkus está ligeramente por delante en el tiempo de la primera respuesta (70ms), mientras que Spring Boot únicamente tiene mejores resultados en el tiempo de compilación, debido a que no realiza ningún procesamiento en este tiempo sino en el tiempo de ejecución.

Para demostrar más a fondo los beneficios de Micronaut se va a realizar una comparación específica, de cada uno de los aspectos destacables de un framework, con Spring Boot.

3.3 Comparación con Spring Boot

Spring Boot (véase 3.2) es un subproyecto del framework Spring que se utiliza para poner en marcha aplicaciones Spring. A continuación, compararemos los dos frameworks en varias áreas, entre ellas, el arranque de una aplicación, el soporte de lenguaje, algunas opciones de configuración y las características que favorecen a los desarrolladores. Para terminar, se van a configurar dos aplicaciones REST básicas para medir las diferencias de inicio y de consumo de memoria.

3.3.1 Características específicas

En este punto, se comparan características básicas de ambos frameworks y algunas específicas que son requeridas por la mayoría de los desarrolladores. Como puede ser la adaptación a la programación reactiva o las características en la nube.

	Micronaut	**Spring Boot**
Tiempo de Arranque y Huella de memoria	Micronaut está diseñado específicamente para tener un tiempo de arranque ligero y rápido. Lo consigue mediante las características explicadas previamente (véase 3.2).	Aunque Spring Boot es un framework adoptado, suele tener un tiempo de arranque más largo y un mayor consumo de memoria en comparación con Micronaut. Se basa en la reflexión en tiempo de ejecución y en proxies dinámicos, lo que puede afectar al rendimiento.
Inyección de dependencias	Micronaut utiliza la inyección de dependencias en tiempo de compilación, en la que el framework genera código durante la compilación. Este enfoque reduce la necesidad de reflexión en tiempo de ejecución y proporciona un mejor rendimiento y una detección temprana de errores.	Spring Boot utiliza también la inyección de dependencias, pero en tiempo de ejecución mediante el contenedor Spring IoC. Se basa en la reflexión en tiempo de ejecución para cablear las dependencias, lo que puede afectar también al tiempo de arranque y rendimiento.
Programación Reactiva	Micronaut tiene soporte nativo para la programación reactiva, como se ha estudiado (véase 3.1.3), lo que permite a los desarrolladores crear aplicaciones reactivas y escalables. Se integra con bibliotecas como Reactor y RxJava, por lo que es adecuado para crear sistemas reactivos.	Spring Boot también proporciona soporte para programación reactiva a través del módulo Spring WebFlux. Sin embargo, en comparación con Micronaut, las capacidades reactivas de Spring Boot se introdujeron en versiones posteriores por lo que no está tan integradas en el framework como en Micronaut que se desarrolló teniéndolas en cuenta.

	Micronaut	Spring Boot
Compatibilidad con GraalVM	Micronaut es compatible con GraalVM, un runtime de alto rendimiento que puede compilar aplicaciones basadas en JVM en código máquina nativo, permitiendo arranques rápidos y consumo de memoria reducido.	Spring Boot no ofrece compatibilidad de manera inicial con GraalVM, sin embargo, esta se puede conseguir con configuración y dependencias adicionales. Las aplicaciones desarrolladas en Spring Boot suelen requerir un mayor esfuerzo de optimización para GraalVM en comparación con las de Micronaut
Características en la nube	Micronaut proporciona soporte incorporado para cloud-native features. Entre ellas incluye soporte para configuración distribuida, descubrimiento de servicios, equilibrio de carga del lado del cliente, rastreo distribuido y funciones serverless. Ofrece una perfecta integración con plataformas en la nube como AWS, Google Cloud y Azure.	Spring Boot también tiene soporte para el Desarrollo native en la nube con características propias como Spring Cloud para sistemas distribuidos y Spring Cloud Function para arquitecturas sin servidor. En este ámbito, Spring Boot cuenta con un ecosistema y comunidad más amplio para el desarrollo nativo en la nube.
Contenedor Servlet	Por defecto, las aplicaciones en Micronaut se ejecutarán en un servidos HTTP basado en Netty. Sin embargo, esto se puede modificar para que se ejecute en Tomcat, Jetty o Undertow.	Spring Boot utilizará Tomcat por defecto, pero también se podrá configurar a Jetty o Undertow, no incluyendo Netty.
Compatibilidad con IDEs	Dado lo prematuro que es el framework, este solo tiene un plugin disponible para el IDE de IntelliJ.	Spring Boot cuenta con plugins para los IDEs más populares permitiendo así una mejor acoplación y mejora del entorno para el framework.
Seguridad	Micronaut ofrece estrategias de autorización que incluyen el inicio de sesión mediante formulario, JWT y LDAP.	Dado la popularidad y antigüedad de Spring Boot, este cuenta con las mismas estrategias que Micronaut además de autorización SAML.
Plantillas para front-end	Micronaut es compatible con Thymeleaf, Handlebars, Apache Velocity, Apache Freemaker, Rocker, Soy/Closure y Pebbles para renderizar plantillas front-end.	Spring Boot es compatible con Thymeleaf, Apache Freemarker, Mustache y Groovy para las plantillas front-end.

3.3.2 Ejemplo con código

En este punto se van a definir dos aplicaciones REST API sencillas para que hagan cálculos aritméticos[26]. La capa de servicio va a consistir en una clase que hace las cuentas matemáticas y la clase con el controller contendrá los endpoints para suma, resta, multiplicación y división.

Aplicación con Micronaut
Se ha definido la clase de servicio con las operaciones matemáticas en Micronaut.

```java
package com.example.services;

import jakarta.inject.Singleton;

2 usages
@Singleton
public class ArithmeticService {
    public float add(float number1, float number2) {
        return number1 + number2;
    }

    1 usage
    public float subtract(float number1, float number2) {
        return number1 - number2;
    }

    1 usage
    public float multiply(float number1, float number2) {
        return number1 * number2;
    }

    1 usage
    public float divide(float number1, float number2) {
        if (number2 == 0) {
            throw new IllegalArgumentException("'number2' cannot be zero");
        }
        return number1 / number2;
    }
}
```

Ilustración 15: Servicio operaciones matemáticas Micronaut

Y el controlador REST con los mismos endpoints que tendrá el de Spring Boot

```java
package com.example.controllers;

import com.example.services.ArithmeticService;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import jakarta.inject.Inject;

@Controller("/math")
public class ArithmeticController {
    4 usages
    @Inject
    private ArithmeticService arithmeticService;

    @Get("/sum/{number1}/{number2}")
    public float getSum(float number1, float number2) {
        return arithmeticService.add(number1, number2);
    }

    @Get("/subtract/{number1}/{number2}")
    public float getDifference(float number1, float number2) {
        return arithmeticService.subtract(number1, number2);
    }

    @Get("/multiply/{number1}/{number2}")
    public float getMultiplication(float number1, float number2) {
        return arithmeticService.multiply(number1, number2);
    }

    @Get("/divide/{number1}/{number2}")
    public float getDivision(float number1, float number2) {
        return arithmeticService.divide(number1, number2);
    }
}
```

Ilustración 16: Controlador Micronaut

Aplicación con Spring Boot

La clase de servicio que implementa los métodos para las cuentas aritméticas de Spring Boot.

```java
package com.master.service;

import org.springframework.stereotype.Service;

2 usages
@Service
public class ArithmeticService {
    public float add(float number1, float number2) {
        return number1 + number2;
    }

    1 usage
    public float subtract(float number1, float number2) {
        return number1 - number2;
    }

    1 usage
    public float multiply(float number1, float number2) {
        return number1 * number2;
    }

    1 usage
    public float divide(float number1, float number2) {
        if (number2 == 0) {
            throw new IllegalArgumentException("'number2' cannot be zero");
        }
        return number1 / number2;
    }
}
```

Ilustración 17: Servicio operaciones algorítmicas Spring Boot

El controlador REST que recibirá las peticiones y llamará al servicio.

```java
import com.master.service.ArithmeticService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/math")
public class ArithmeticController {
    4 usages
    @Autowired
    private ArithmeticService arithmeticService;

    @GetMapping("/sum/{number1}/{number2}")
    public float getSum(@PathVariable("number1") float number1, @PathVariable("number2") float number2) {
        return arithmeticService.add(number1, number2);
    }

    @GetMapping("/subtract/{number1}/{number2}")
    public float getDifference(@PathVariable("number1") float number1, @PathVariable("number2") float number2) {
        return arithmeticService.subtract(number1, number2);
    }

    @GetMapping("/multiply/{number1}/{number2}")
    public float getMultiplication(@PathVariable("number1") float number1, @PathVariable("number2") float number2) {
        return arithmeticService.multiply(number1, number2);
    }

    @GetMapping("/divide/{number1}/{number2}")
    public float getDivision(@PathVariable("number1") float number1, @PathVariable("number2") float number2) {
        return arithmeticService.divide(number1, number2);
    }
}
```

Ilustración 18: Controlador Spring Boot

Antes de analizar los resultados de las pruebas, es importante recordar que como ya se ha explicado anteriormente, ambos frameworks proporcionan inyección de dependencias, pero de diferente manera. Spring Boot gestiona la inyección en tiempo de ejecución mediante reflexión y proxies. Mientras que Micronaut construye los datos de inyección de dependencias cuando se compila.

En cuanto a los tiempos de inicio tenemos los siguientes datos:

	Micronaut	**Spring Boot**
Tiempo de inicio	[main] INFO io.micronaut.runtime.Micronaut - Startup completed in 1278ms. Server Running: http://localhost:57535	[main] INFO c.b.m.v.s.CompareApplication - Started CompareApplication in 3.179 seconds (JVM running for 4.164)

Como se esperaba, Micronaut arranca en poco más de un segundo mientras que la aplicación de Spring Boot supera los tres segundos.

La siguiente prueba es la memoria utilizada por cada una de las aplicaciones mientras se llama a los endpoints definidos:

	Micronaut	**Spring Boot**
Memoria Inicial	0.25 GB	0.25 GB
Memoria Utilizada	0.01 GB	0.02 GB
Memoria Máxima	4.00 GB	4.00 GB
Memoria Comprometida	0.03 GB	0.06 GB

Con este ejemplo se demuestra que Micronaut utiliza aproximadamente la mitad de memoria que utiliza la aplicación de Spring Boot, con las mismas situaciones, características y endpoints.

4. Caso de estudio

En este apartado se va a hacer una demostración del funcionamiento del framework Micronaut, para ello se desarrollará un proyecto que consistirá en una tienda online de productos de animales, esta tienda tendrá los productos agrupados por categorías y podrán añadirse al carrito de la cesta del usuario para hacer una compra.

4.1 Arquitectura

Este proyecto va a constar de las tres capas básicas del patrón arquitectónico MVC. Se ha desarrollado las tres capas de este patrón, la capa de la vista, la capa del controlador y la capa del modelo.

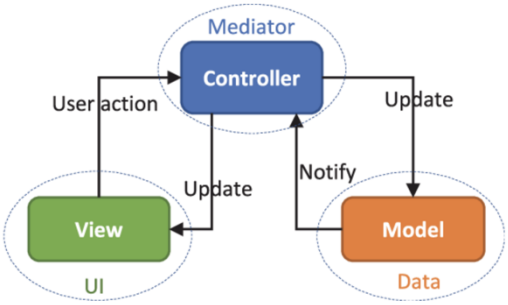

Ilustración 19:Diagrama MVC

En el sistema que compone el desarrollo se puede diferenciar la parte Front-end encargada de la parte de la vista, de esta capa no se mostrará el desarrollo, pero sí que se utilizará para poder dar una visión más natural de la aplicación. Realizará el envío de peticiones HTTP por el usuario al Back end. Para su creación se ha dado uso de la librería React[28].

La parte de Back-end será la encargada de recibir las peticiones siendo la capa de controlador y la capa de modelo, a su vez, para comunicarse con la base de datos. Ha sido desarrollada con el framework Micronaut y funcionará como una API REST para una tienda de productos de animales.

4.1.1 Base de Datos

Para comenzar con el desarrollo se va a crear una base de datos en la que se incluyen los datos de los productos y las categorías de cada uno. Además, se va a crear un una tabla para guardar los usuarios que se hayan registrado y los pedidos que realicen.

Para la creación de la base de datos se ha utilizado Mysql Workbench, herramienta que permite la creación de bases de datos relacionales. El esquema de la base de datos definida es el siguiente.

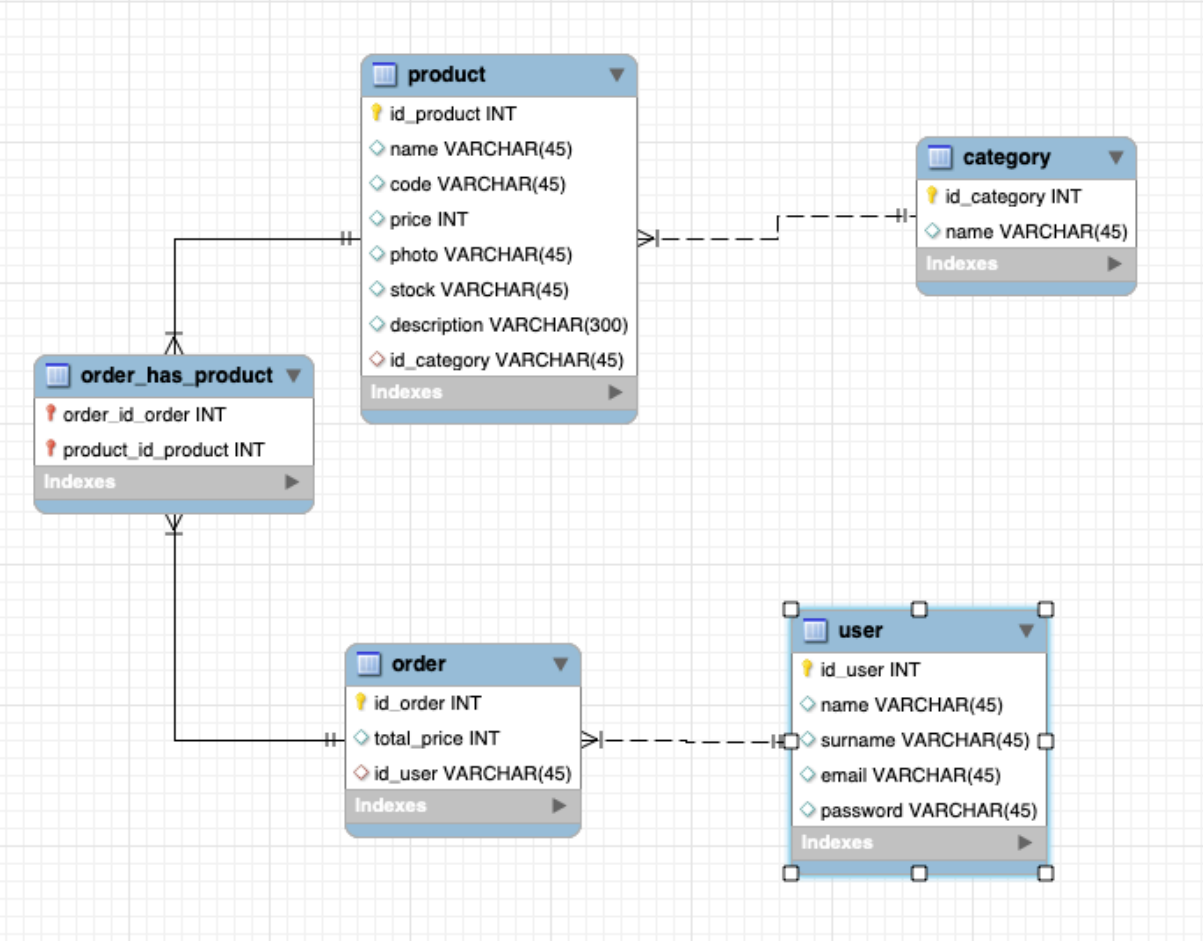

Ilustración 20: Esquema base de datos caso de estudio

Este diagrama de clases representa las entidades que forman parte del sistema. Consta de 4 entidades diferentes y relacionadas entre ellas: Categoría, Producto, Pedido, Usuario y una quinta tabla que se crea con la relación entre Pedido y Producto.

4.1.2 Micronaut

Para la parte de back-end se ha hecho uso de Micronaut Launcher, una interfaz que permite a los usuarios crear el arquetipo (Project Structure) de la aplicación con las dependencias ya incorporadas desde la propia página web, evitando tener que añadir las dependencias manualmente en el archivo pom.xml. Al ser configurado desde Micronaut Launcher los conflictos que puedan ocasionar las dependencias entre ellas y las versiones queda autogestionado.

En la siguiente ilustración se muestra la configuración inicial del proyecto, junto con las dependencias necesarias para su correcto funcionamiento.

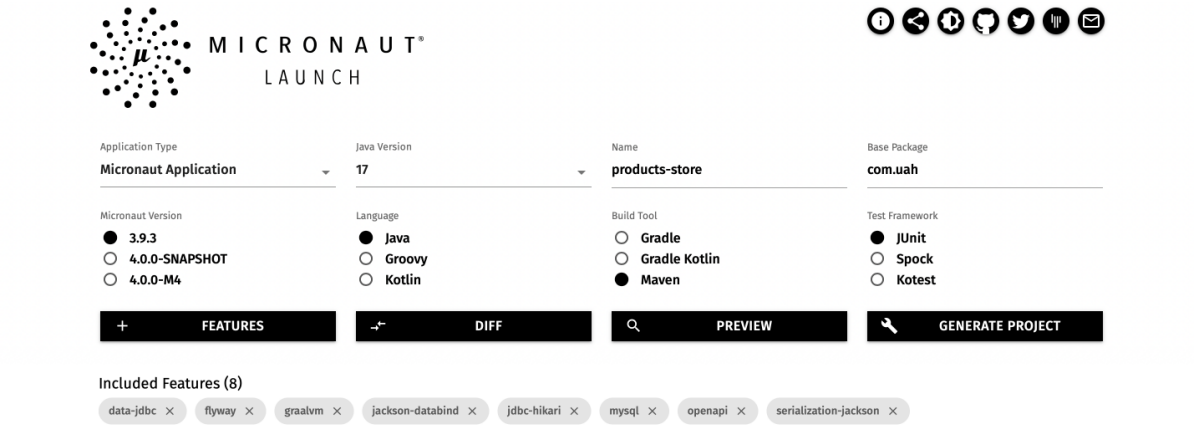

Ilustración 21: Arquetipo y dependencias Micronaut Launcher

Con el arquetipo del proyecto y las dependencias incluidas se procede al desarrollo de la API para ser consumida. Para esto, se ha configurado la conexión con la base de datos gracias a la dependencia de data-jdbc y la definición en el archivo application.yml.

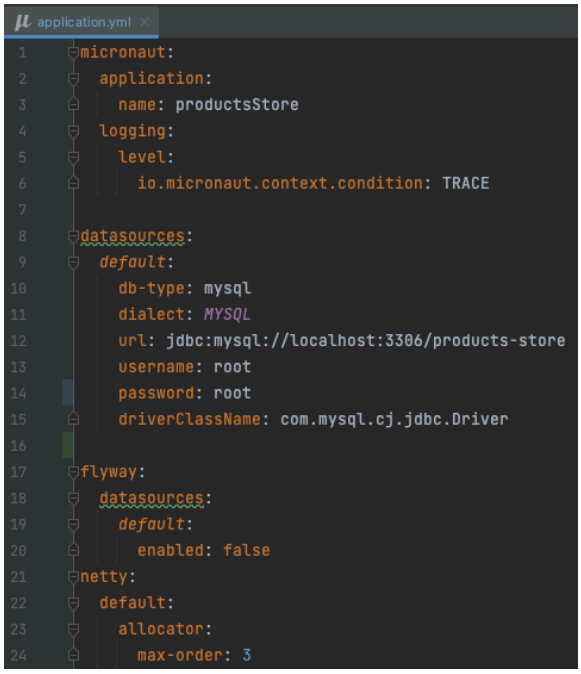

Ilustración 22: Archivo application.yml caso de estudio

Para el desarrollo, se han configurado las siguientes clases e interfaces que van a permitir el correcto funcionamiento de la API. En este caso se ha definido las siguientes carpetas:

- Domain: Incluye las entidades, los dtos (Data Transfer Object) y los mappers que efectúan la conversión entre las entidades y los dtos.
- Repository: Las interfaces de los repositorios efectúan las funcionalidades con la base de datos, es decir, obtener los valores o guardar nuevos datos.
- Service: En esta carpeta se pueden encontrar las interfaces de los servicios y la implementación de cada uno. Es la carpeta encargada de comunicarse con los repositorios.
- Controller: Incluye los controladores que tendrán los endpoints para cada una de las peticiones que se realizarán y que llamarán a los servicios.

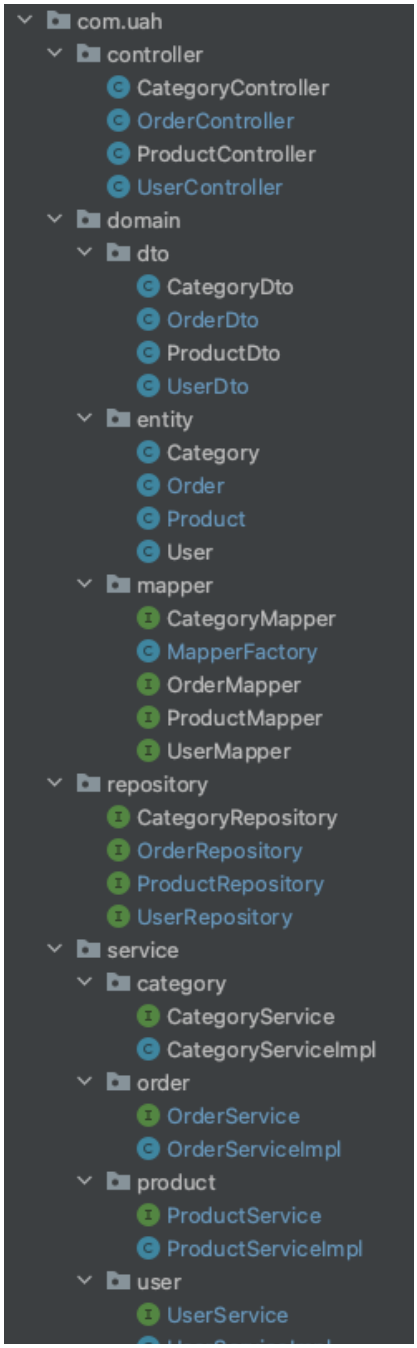

Ilustración 23: Estructura carpetas caso de estudio

Clase Entidad:

En la clase de tipo entidad se definen los atributos que están incluidos en la base de datos. Cada atributo está identificado con una anotación que indica cómo se mapea en la base de datos. Por ejemplo, la propiedad "idProduct" utiliza una anotación para generar automáticamente un identificador único.

Además, la clase debe estar anotada con @Entity para que Micronaut la reconozca y la compile correctamente. Esta clase también incluye los métodos getters y setters para acceder y modificar los atributos.

```
28 usages  SantiTamariz *
@Entity
public class Product implements Serializable {
    5 usages
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Id
    @Column(name = "id_product")
    private int idProduct;
    8 usages
    @Basic
    @Column(name = "name")
    private String name;
    8 usages
    @Basic
    @Column(name = "code")
    private String code;
    8 usages
    @Basic
    @Column(name = "price")
    private Integer price;
    8 usages
    @Basic
    @Column(name = "photo")
    private String photo;
    8 usages
    @Basic
    @Column(name = "stock")
    private String stock;
    8 usages
    @Basic
    @Column(name = "description")
    private String description;
    8 usages
    @Basic
    @Column(name = "id_category")
    private Long idCategory;

    2 usages
    @ManyToMany(mappedBy = "products")
    @JsonIgnore
    private List<Order> orders;
```

Ilustración 24: Clase entidad Product

Clase DTO:

La clase de los DTOs se utiliza para transportar los objetos entre diferentes capas de la aplicación. Contiene las mismas propiedades que la clase entidad. Estos atributos son los que se envían y reciben en las peticiones HTTP.

Para que el objeto pueda ser convertido a formato JSON, se utiliza la anotación @Serdeable proporcionada por Micronaut. Esta anotación permite la serialización y deserialización del objeto.

Además, al igual que la clase Entidad, la clase DTO también incluye los métodos getters y setters para facilitar la conversión de objetos utilizando un mapper.

```java
@Serdeable
public class ProductDto {

    2 usages
    private int idProduct;

    2 usages
    private String name;

    2 usages
    private String code;

    2 usages
    private Integer price;

    2 usages
    private String photo;

    2 usages
    private String stock;

    2 usages
    private String description;

    2 usages
    private String idCategory;

    2 usages  SantiTamariz
    public int getIdProduct() { return idProduct; }

    2 usages  SantiTamariz
    public void setIdProduct(int idProduct) { this.idProduct = idProduct; }

    SantiTamariz
```

Ilustración 25: Clase DTO para Product

Clase Mapper:

La clase de tipo mapper utiliza la biblioteca MapStruct para realizar los mapeos entre las clases Entidad y DTO. Este mapeo permite convertir un tipo de objeto en otro. Para lograr esto, se definen dos métodos en una interfaz y la implementación se genera automáticamente.

```java
package com.uah.domain.mapper;

import com.uah.domain.dto.ProductDto;
import com.uah.domain.entity.Product;
import org.mapstruct.Mapper;

5 usages  1 implementation  SantiTamariz
@Mapper
public interface ProductMapper {
    1 implementation  SantiTamariz
    Product toEntity(ProductDto productDto);
    1 implementation  SantiTamariz
    ProductDto toDto(Product product);
}
```

Ilustración 26: interfaz ProductMapper

Para utilizar los mappers, es necesario declararlos como beans en Micronaut. Para ello, se crea una clase llamada MapperFactory donde se inicializan los mappers utilizando las anotaciones @Bean y @Singleton. Estas anotaciones permiten que Micronaut reconozca y administre los mappers correctamente.

```java
SantiTamariz
@Factory
public class MapperFactory {
    SantiTamariz
    @Bean
    @Singleton
    public CategoryMapper categoryMapper() { return new CategoryMapperImpl(); }

    SantiTamariz
    @Bean
    @Singleton
    public ProductMapper productMapper() { return new ProductMapperImpl(); }

    SantiTamariz
    @Bean
    @Singleton
    public OrderMapper orderMapper() { return new OrderMapperImpl(); }

    SantiTamariz
    @Bean
    @Singleton
    public UserMapper userMapper() { return new UserMapperImpl(); }

    }
```

Ilustración 27: Clase MapperFactory para los mappers

Interfaz Servicio:

La interfaz de un servicio define los métodos y operaciones que deben ser implementados por las clases que proporcionan la lógica de negocio relacionada con la entidad. Esta interfaz define las operaciones disponibles para interactuar con los objetos de la entidad.

```
7 usages  1 implementation  SantiTamariz *
public interface ProductService {
    1 implementation  SantiTamariz
    List<ProductDto> findAll();
    1 implementation  SantiTamariz
    void save(ProductDto productDto);

    1 usage  1 implementation  new *
    ProductDto findById(Long productId);

}
```

Ilustración 28: Interfaz Servicio Product

Clase Implementación Servicio:

La clase de la implementación de un servicio es la implementación concreta de la interfaz. En esta clase se definen y se implementan los métodos que realizan la lógica de negocio relacionada con la entidad. Estos métodos pueden realizar operaciones como guardar, actualizar o eliminar objetos de la entidad en la base de datos.

En este caso se utilizan los métodos para recuperar todos los productos, guardar un nuevo producto o buscar un producto por su ID.

```
@Singleton
public class ProductServiceImpl implements ProductService{
    4 usages
    private final ProductRepository productRepository;
    4 usages
    private final ProductMapper productMapper;

    SantiTamariz
    public ProductServiceImpl(ProductRepository productRepository, ProductMapper productMapper) {
        this.productRepository = productRepository;
        this.productMapper = productMapper;
    }

    SantiTamariz
    @Override
    public List<ProductDto> findAll() {
        return productRepository.findAll().stream().map(productMapper::toDto).collect(Collectors.toList());
    }

    SantiTamariz
    @Override
    public void save(ProductDto productDto) { productRepository.save(productMapper.toEntity(productDto)); }

    1 usage  new *
    @Override
    public ProductDto findById(Long productId) {
        return productMapper.toDto(productRepository.findById(productId).get());
    }

}
```

Ilustración 29: Implementación servicio Product

Clase Controlador:

La clase de tipo controlador actúa como punto de entrada para las peticiones HTTP. Esta clase está anotada con @Controller y define los métodos que manejan las diferentes rutas y verbos HTTP para interactuar con la entidad.

En los métodos del controlador se pueden utilizar los servicios y mappers correspondientes para realizar las operaciones necesarias, como obtener objetos de la entidad, crear nuevos objetos o actualizarlos.

ProductController.java

```java
package com.uah.controller;

import com.uah.domain.dto.ProductDto;
import com.uah.service.product.ProductService;
import io.micronaut.http.annotation.Controller;
import io.micronaut.http.annotation.Get;
import io.micronaut.http.annotation.Post;

import java.util.List;

SantiTamariz
@Controller("/products")
public class ProductController {

    3 usages
    private final ProductService productService;

    SantiTamariz
    public ProductController(ProductService productService) { this.productService = productService; }

    SantiTamariz
    @Get("/list")
    public List<ProductDto> findAll() { return productService.findAll(); }

    SantiTamariz
    @Post("/save")
    public void save(ProductDto productDto) { productService.save(productDto); }
}
```

Ilustración 30: Clase controlador producto

En resumen, estas son las diferentes clases que componen una API en Micronaut. Cada una cumple un propósito específico y colabora en la implementación de una arquitectura bien estructurada y modular.

4.2 Proyecto

Para mostrar el funcionamiento de la API desarrollada con Micronaut se va a consumir con un desarrollo front end dando uso de la librería de React. Para ello se han configurado los servicios que van a hacer las peticiones a la API y que van a mostrar los resultados.

Estos servicios únicamente tendrán que realizar las llamadas con el siguiente formato:

http://localhost:8080/orders/save

El ejemplo es la llamada necesaria para guardar un nuevo pedido de un usuario, en el body de la llamada se debe incluir los datos para guardar el pedido.

4.2.1 Registro e Inicio de sesión

En la pantalla de inicio, se presenta al usuario la opción de iniciar sesión o registrarse en la tienda online de productos. Para registrarse, el usuario debe proporcionar su nombre, apellido, correo electrónico y contraseña.

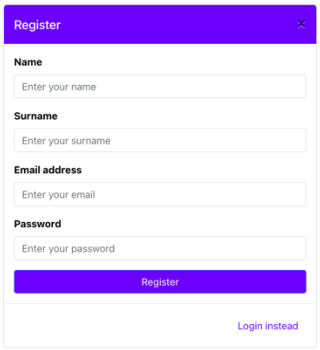

Ilustración 31: Registro de usuario

En caso de tener una cuenta existente, simplemente debe iniciar sesión con su correo electrónico y contraseña. Una vez iniciada la sesión, el usuario será redirigido a la página principal de productos.

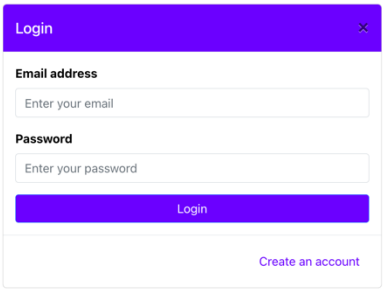

Ilustración 32: Inicio sesión de usuario

4.2.2 Categorías y Productos

Una vez que el usuario ha iniciado sesión, accederá a la pantalla de categorías y productos. Desde esta pantalla, puede hacer clic en las diferentes categorías, y se mostrarán los productos asociados a cada una. Cada producto tiene varios datos, incluyendo su nombre, descripción, precio, stock disponible, código de producto y una imagen.

Si un usuario selecciona el botón “Add to cart” en cualquier producto, se reducirá en uno la cantidad disponible en stock y se añadirá al carrito de la compra. En caso de que un producto haya agotado su stock, el botón no estará disponible, y la imagen se mostrará en gris, indicando que no hay más unidades disponibles para comprar.

En el carrito de la compra, el usuario puede ver los productos seleccionados, y tiene la opción de aumentar o disminuir la cantidad de cada uno. Además, se muestra el precio total por tipo de producto y el precio final para realizar el pedido.

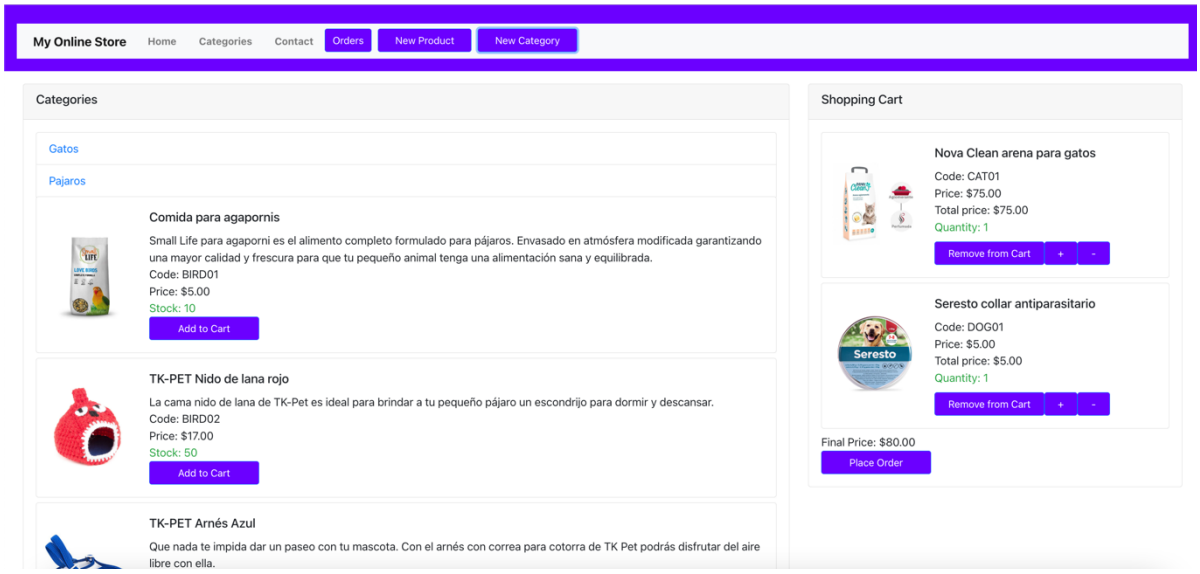

Ilustración 33: Página principal con las categorías y sus productos

4.2.3 Nueva categoría y producto

Para que un usuario pueda crear una nueva categoría o un nuevo producto (aunque no debiese ser potestad del usuario) se han definido dos botones, “New Product” y “New Category”, para mostrar el uso de la API para guardar en la base de datos nuevos productos y nuevas categorías.

En estos formularios aparecen los valores que debe rellenar el usuario para poder crearlos.

En el caso de crear una nueva categoría únicamente deberá ser necesario indicar el nombre de la nueva categoría.

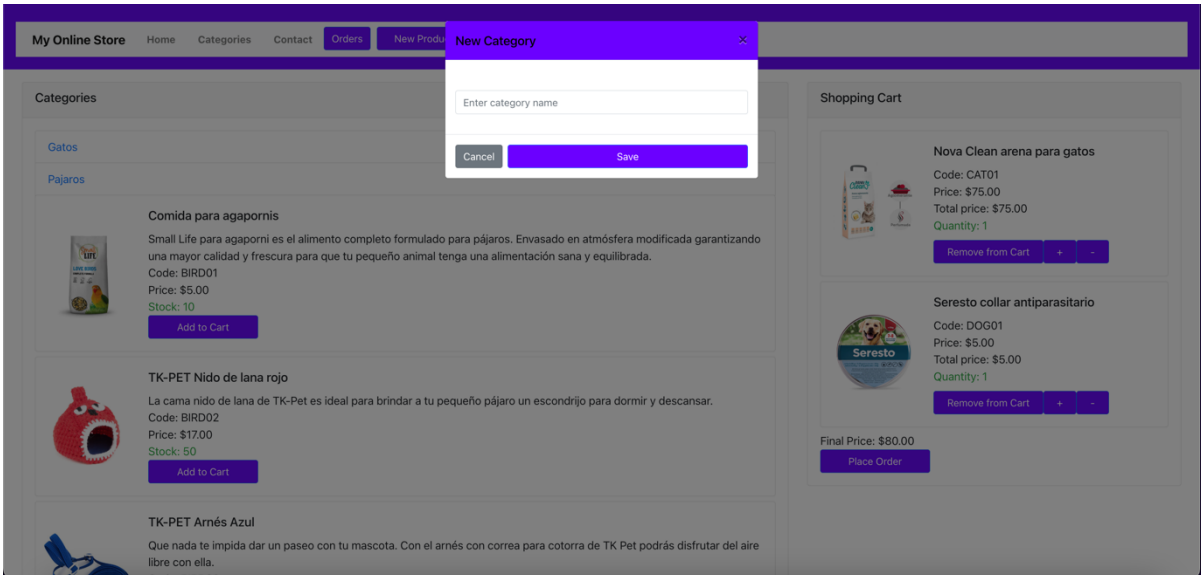

Ilustración 34: Nueva categoría

En el caso de añadir un nuevo producto el usuario deberá indicar el nombre, imagen, descripción, código de producto, precio, la categoría a la que pertenece y el stock disponible.

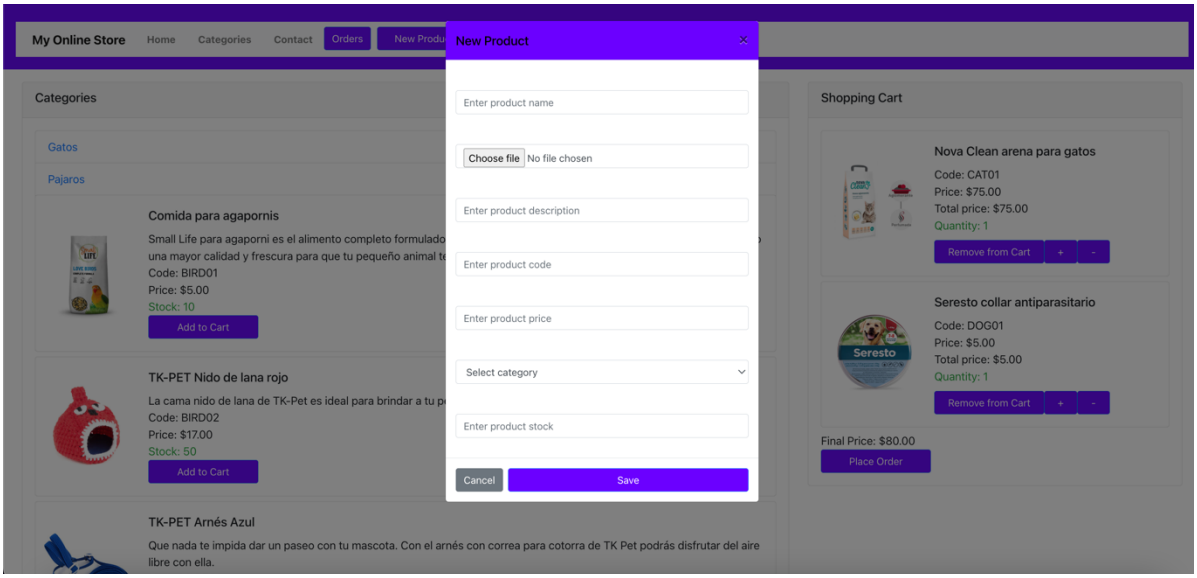

Ilustración 35: Nuevo producto

Se da por hecho que el usuario que cree una nueva categoría o un nuevo producto deberá tener un rol de administrador de la tienda.

4.2.4 Pedidos

Una vez que el usuario presiona el botón “Place Order”, aparece una ventana modal para confirmar el pedido, que luego se agrega a la lista de pedidos del usuario registrado.

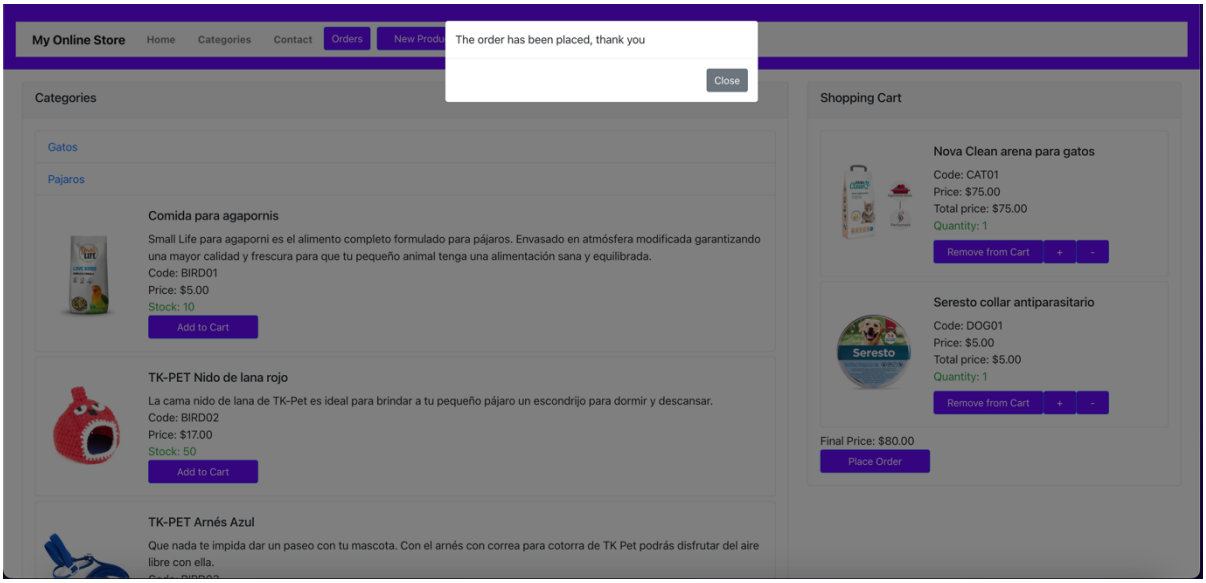

Ilustración 36: Confirmación pedido

El usuario puede acceder al listado de pedidos realizados, donde se mostrarán todos los pedidos que ha realizado, incluyendo los productos y el precio final de cada pedido.

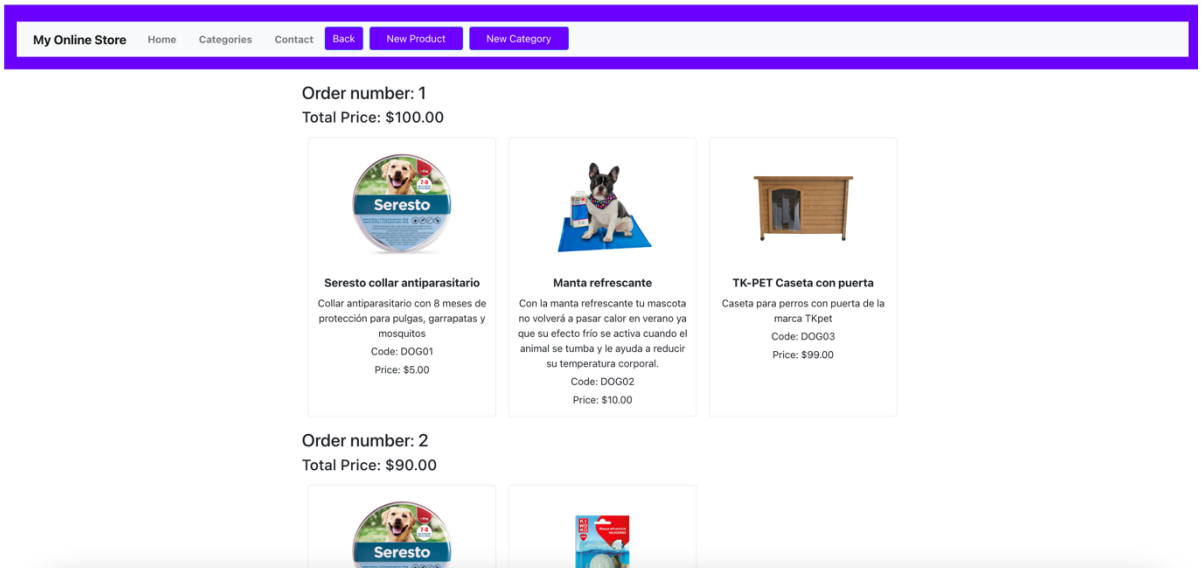

Ilustración 37: Listado pedidos

5. Conclusión

5.1 Resultados y conclusiones

Realizar este trabajo de fin de master supone un gran estímulo para la investigación y el descubrimiento, brindándome la posibilidad de explorar un nuevo framework de desarrollo back-end en pleno auge. A través de esta investigación, he adquirido un amplio conocimiento sobre la diversidad de frameworks disponibles, cada uno con características distintas.

Teniendo en cuenta los objetivos establecidos para este trabajo, puedo afirmar que se ha logrado con gran éxito, el estudio, análisis y evaluación de Micronaut, abarcando la mayoría de sus aspectos y lo que ofrece como framework. Espero que este estudio genere interés entre los desarrolladores y proporcione un conocimiento profundo sobre Micronaut, brindando un ejemplo práctico para su comprensión.

Personalmente, la investigación sobre Micronaut me ha permitido descubrir sus capacidades y cómo es posible obtener resultados excepcionales en comparación con los frameworks más populares, lo cual no creía posible. También he aprendido sobre la programación reactiva y las oportunidades que ofrece. Es importante destacar la falta de comunidad en torno a Micronaut, pero confío en que en el futuro esta comunidad crezca significativamente y este pequeño inconveniente sea superado.

El desarrollo del caso de estudio me ha brindado la oportunidad de aplicar lo aprendido durante la investigación y observar los resultados de las capacidades que hacen destacar a Micronaut por encima de otros frameworks.

En resumen, este TFM ha sido una experiencia enriquecedora que me ha permitido explorar nuevas tecnologías y comprender las fortalezas y debilidades de Micronaut en comparación con otros frameworks. Estoy emocionado por las oportunidades que Micronaut ofrece y su potencial para impulsar el desarrollo de aplicaciones back-end modernas y eficientes.

5.2 Trabajos Futuros

En cuanto a las líneas de trabajo futuro, existen diversas oportunidades para profundizar en aspectos clave que ofrece Micronaut y explorar nuevas capacidades. A continuación, se mencionan algunos posibles trabajos futuros que podrían enriquecer aún más el conocimiento sobre este framework y su aplicación práctica.

1. Investigación en Programación Reactiva: Una línea de trabajo interesante, sería realizar una investigación más profunda sobre la programación reactiva en Micronaut, centrándose en la integración de los frameworks reactivos de Project Reactor y RxJava, tal como se ha definido en la documentación. Esta exploración permitiría comprender en mayor detalle cómo aprovechar al máximo el potencial de la programación reactiva en el desarrollo de aplicaciones con Micronaut.

2. Despliegue en la Nube: Otro aspecto prometedor para futuras investigaciones sería estudiar y aplicar la capacidad de despliegue de microservicios en la nube que ofrece Micronaut. Sería interesante explorar la configuración y las mejores prácticas para implementar y gestionar microservicios en entornos de computación en la nube, aprovechando las características y beneficios que Micronaut proporciona en este ámbito. Esto incluiría aspectos como la escalabilidad, la gestión de recursos y la integración con servicios en la nube populares.

3. Modularización de Microservicios: Micronaut es conocido por su enfoque modular y su capacidad para desarrollar aplicaciones basadas en microservicios. Un trabajo futuro podría centrarse en la implementación y exploración de la modularización de microservicios utilizando Micronaut. Esto implica la identificación de diferentes funcionalidades y bases de datos que puedan ser separadas en módulos independientes, lo que permitiría un desarrollo más flexible y escalable de aplicaciones basadas en microservicios. En el caso de estudio desarrollado no tendría tanto sentido ya que no se diferencian muchos módulos, pero sí que sería interesante aplicarlo en un caso de estudio de una aplicación monolítica de gran tamaño y como subdividirla en pequeños módulos independientes.

4. Mejora de la Comunidad: Aunque Micronaut ha ganado popularidad en los últimos años, aún se encuentra en desarrollo y su comunidad no es tan extensa como la de otros frameworks establecidos. A mi parecer, el trabajo futuro más valioso sería centrarse en fomentar el crecimiento y la participación de la comunidad de Micronaut, promoviendo la colaboración, el intercambio de conocimientos y la creación de recursos y herramientas adicionales. Esto ayudaría a fortalecer la comunidad y a impulsar la adopción y evolución continua del framework.

En resumen, los trabajos futuros mencionados anteriormente representan solo algunas de las posibilidades que podrían explorarse para profundizar en el conocimiento de Micronaut y mejorar su aplicación en el desarrollo de aplicaciones back-end. Estas investigaciones y desarrollos adicionales contribuirían a la evolución y adopción continua de Micronaut, permitiendo a los desarrolladores aprovechar al máximo sus características y beneficios en el contexto de las tecnologías emergentes y las demandas cambiantes del desarrollo de software.

6. Bibliografía

[1] Arquitectura de microservicios vs arquitectura monolítica (3 Mayo 2018). https://www.viewnext.com/arquitectura-de-microservicios-vs-arquitectura-monolitica/ (Accedido Julio 2023)

[2] AST. Abstract Syntax Tree https://deepsource.com/glossary/ast (Accedido Julio 2023)

[3] Compilador AOT, IBM (31 Mayo 2022) https://www.ibm.com/docs/es/sdk-java-technology/8?topic=reference-aot-compiler (Accedido Julio 2023)

[4] El patrón MVC, arquitectura cliente vs servidor, Cecilio Álvarez Caules (2016) https://www.arquitecturajava.com/patron-mvc-arquitectura-cliente-vs-servidor/ (Accedido: julio 2023)

[5] Frameworks, ¿Qué es un framework de software?, TIC. Portal https://www.ticportal.es/glosario-tic/framework-software (Accedido Julio 2023)

[6] Grails Docs. Introduction https://docs.grails.org/5.3.2/ (Accedido Julio 2023)

[7] Inversión de control IoC Spring, Auribox https://blog.auriboxtraining.com/java/introduccion-la-ioc/ (Accedido Julio 2023)

[8] JVM, Java Virtual Machine, javaTpoint https://www.javatpoint.com/jvm-java-virtual-machine (Accedido Julio 2023)

[9] Jakarta EE, Priya Kharia-Hanks (22 febrero 2022) https://blog.payara.fish/jakarta-ee-java-ee-guide (Accedido Julio 2023)

[10] Java, ¿Qué es y principales características? https://www.javatpoint.com/java-basics#:~:text=Java%20is%20a%20high%2Dlevel,Microsystem%20takeover%20by%20Oracle%20Corporation. (Accedido Julio 2023)

[11] Java, Oracle documentation https://docs.oracle.com/javase/8/docs/technotes/guides/language/index.html (Accedido Julio 2023)

[12] Programación orientada a aspectos (AOP)
Kizcales G., Lamping J., Mendhekar A., Maeda C., ***Aspect-Oriented Programming.*** *The 11th European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science (LNCS), Springer-Verlag, Vol. 1241, Jyväskylä, Finland, June 9-13, 1997.*

[13] Micronaut Docs. Introduction
https://docs.micronaut.io/snapshot/guide/index.html
(Accedido Julio 2023)

[14] Micronaut Docs. Dependency Injection, inversion control
https://docs.micronaut.io/latest/guide/#ioc
(Accedido Julio 2023)

[15] Micronaut IOC container JSR-330, Robert Pudlik (01 Junio 2023)
https://softwaremill.com/introduction-to-micronaut-ioc-basics/#summary
(Accedido Julio 2023)

[16] Micronaut. AOT, Sergio del Amo Caballero (20 Diciembre 2021)
https://micronaut.io/2021/12/20/micronaut-aot-build-time-optimizations-for-micronaut-applications/
(Accedido Julio 2023)

[17] Micronaut. Features & Benefits
https://objectcomputing.com/products/micronaut/features-benefits
(Accedido Julio 2023)

[18] Micronaut Launcher
https://micronaut.io/launch/
(Accedido Julio 2023)

[19] Micronaut. Open Webminar
https://openwebinars.net/blog/micronaut-framework-full-stack/
(Accedido Julio 2023)

[20] Micronaut, ¿Por qué Micronaut es tan adecuado para microservicios?, Krypton
https://kryptonsolid.com/que-es-micronaut-introduccion-al-marco-de-micronaut/
(Accedido Julio 2023)

[21] Micronaut, Programación reactiva
https://www.genbeta.com/desarrollo/desarrollando-microservicios-reactivos-micronaut
(Accedido Julio 2023)

[22] Micronaut. Reactive HTTP Client
https://www.baeldung.com/micronaut
(Accedido Julio 2023)

[23] Micronaut. Reactive, Piotr Minkowski (12 Noviembre 2019)
https://piotrminkowski.com/2019/11/12/micronaut-tutorial-reactive/
(Accedido Julio 2023)

[24] Micronaut & Microservices, Kerry Doyle (16 Junio 2019)
https://www.techtarget.com/searchapparchitecture/tip/The-fundamentals-of-Micronaut-and-microservices-development
(Accedido Julio 2023)

[25] Micronaut vs Quarkus vs Spring Boot, Graeme Rocher (7 Abril 2020)
https://micronaut.io/2020/04/07/micronaut-vs-quarkus-vs-spring-boot-performance-on-jdk-14/
(Accedido Julio 2023)

[26] Micronaut vs Spring Boot Baeldung, Amy DeGregorio (20 Agosto 2022)
https://www.baeldung.com/micronaut-vs-spring-boot
(Accedido Julio 2023)

[27] Microservicios, que son y para qué sirven
https://www.redhat.com/es/topics/microservices
(Accedido Julio 2023)

[28] React. Doc
https://es.react.dev
(Accedido Julio 2023)

[29] Reactive Streams
http://www.reactive-streams.org
(Accedido Julio 2023)

[30] SOAP, web services. (15 Abril 2020)
https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/soap-simple-object-access-protocol/
(Accedido Julio 2023)

[31] Soporte integrado Programación reactiva, Bhaunadar (26 Marzo 2023)
https://blog.bhanunadar.com/benefits-of-using-micronaut/
(Accedido Julio 2023)

[32] Spring - A Manager's Overview
https://assets.spring.io/drupal/files/Spring_A_Managers_OverviewVer05.pdf
(Accedido Julio 2023)

[33] Spring Framework, projects
https://spring.io/projects
(Accedido Julio 2023)

[34] Spring Container e Inyección de Dependencias – Daniel Diaz Suarez
https://www.adictosaltrabajo.com/2013/07/25/spring-container-inyeccion-dependencias/
(Accedido Julio 2023)

[35] The Java Language Environment
https://www.oracle.com/java/technologies/language-environment.html
(Accedido Julio 2023)

[36] Twelve-Factor App
https://12factor.net
(Accedido Julio 2023)

[37] Web on Servlet Stack, Spring
https://docs.spring.io/spring-framework/docs/current/reference/html/web.html
(Accedido Julio 2023)

[38] XML:¿qué es y para qué sirve este lenguaje de marcado?, Iván de Souza (2019)
https://rockcontent.com/es/blog/que-es-xml/
(Accedido Julio 2023)