Universidad de Cantabria

Facultad de Ciencias

Desarrollo de un Simulador de (Micro)servicios.

Development of a (Micro)services Simulator.

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Rubén Lozano Merino
Director: Pablo Sánchez Barreiro
Co-Director: Laura Ruiz Bedoya
Julio - 2023

Agradecimientos

Deseo expresar mi agradecimiento a Pablo Sánchez Barreiro, por su papel como tutor de este Trabajo de Fin de Grado. Su supervisión ha sido parte esencial de este proyecto.

Mi gratitud a Laura Ruiz Bedoya, mi tutora en NTT Data, cuya orientación y experiencia práctica han sido fundamentales para aplicar los conceptos teóricos en un contexto real. Su visión y dedicación han sido apreciadas durante el desarrollo de este trabajo.

Agradezco a mi familia por su apoyo incondicional. Su fe en mis habilidades ha sido un pilar fundamental en todo este proceso y su respaldo ha sido esencial en los momentos más desafiantes.

Un reconocimiento especial a mis amigos por hacerme más amenos estos años, en especial a mi grupo de amigos de la universidad, "Los Chemas". Ellos han estado presentes durante este recorrido, proporcionando tanto alivio de los momentos de estrés como una amistad duradera que ha sido invaluable.

Finalmente, agradezco a todos los que de una manera u otra han formado parte de estos años de viaje académico. Desde los profesores que me han enseñado hasta los compañeros que han compartido este camino, todos han tenido un papel en ayudarme a llegar a este punto.

Resumen

Cada día más sistemas informáticos se desarrollan siguiendo un enfoque basado en microservicios. De acuerdo con este enfoque, la parte del servidor de una aplicación software se descompone en una serie de servicios de pequeño tamaño y con una funcionalidad muy determinada, los cuales se conocen como microservicios. Estos *microservicios* necesitan con frecuencia comunicarse entre ellos para llevar a cabo una determinada operación. Por ejemplo, la realización de una compra siguiendo este enfoque podría requerir del alta de la nueva compra en el microservicio de compra, de la recuperación de la dirección habitual de entrega del microservicio de gestión de perfiles de los usuarios y de la inserción de una nueva orden de entrega en el microservicio de gestión de entregas a domicilio.

Por tanto, para poder desarrollar un determinado microservicio puede ser necesario tener que comunicarse con los otros microservicios de los que se dependa. Por ejemplo, volviendo al caso anterior, para poder desarrollar el microservicio de gestión de compras podríamos necesitar comunicarnos con los microsservicios de gestión del perfiles del usuario y con el de gestión de entregas a domicilio.

El objetivo de este proyecto es desarrollar una herramienta que, a partir de la especificación de un microservicio, permita desplegar un servicio que simule el comportamiento de dicho microservicio. Es decir, el servicio desplegado deberá ser capaz de recibir las mismas peticiones que el microservicio simulado, devolviendo para cada petición un modelo de respuesta predefinido o datos sintéticos. Por ejemplo, para el caso de la recuperación de la dirección habitual de entrega, el microservicio simulado podría devolver siempre la misma dirección de entrega o una dirección aleatoriamente generada.

De esta forma, los microservicios que dependen del microservicio simulado podrán utilizar durante su desarrollo la simulación en lugar del servicio original. Esto proporciona dos ventajas: (1) se evita tener que usar el servicio real durante el desarrollo de otros servicios, evitando que el desarrollo y pruebas de otros servicios interfieran con la operación natural de éste; y, (2) se posibilita que se puedan desarrollar servicios que dependan del microservicio simulado antes de que éste exista. Es decir, para desarrollar el microservicio de gestión de compras no sería necesario desarrollar los microservicios de gestión de perfiles de usuarios y de entregas a domicilio, bastaría con simularlos.

El proyecto se desarrollará dentro de la empresa NTT Data conforme a una metodología ágil. La especificación de la interfaz ofrecida por cada microservicio se realizará con OpenAPI. El proyecto se implementará sobre Spring y se utilizará la librería Swagger Parser para procesar el contenido de las especificaciones OpenAPI.

Palabras clave

Simulacón de microservicios, Spring Boot, Swagger, Pruebas de microservicios, Desarrollo de software, REST.

Abstract

Every day, more computer systems are being developed following a microservices-based approach. According to this approach, the server part of a software application is broken down into a series of small, highly specific services, known as microservices. These microservices often need to communicate with each other to carry out a particular operation. For example, making a purchase following this approach might require the new purchase to be registered in the purchase microservice, the retrieval of the usual delivery address from the user profile management microservice, and the insertion of a new delivery order in the home delivery management microservice.

Therefore, to be able to develop a particular microservice, it may be necessary to communicate with the other microservices on which it depends. For example, returning to the previous case, to develop the purchase management microservice, we might need to communicate with the user profile management and home delivery management microservices.

The goal of this project is to develop a tool that, starting from the specification of a microservice, allows for the deployment of a service that simulates the behavior of said microservice. That is, the deployed service should be able to receive the same requests as the simulated microservice, returning a predefined response model or synthetic data for each request. For example, in the case of retrieving the usual delivery address, the simulated microservice could always return the same delivery address or a randomly generated address.

In this way, the microservices that depend on the simulated microservice can use the simulation during their development instead of the original service. This provides two advantages: (1) it avoids having to use the real service during the development of other services, preventing the development and testing of other services from interfering with its natural operation; and, (2) it enables services that depend on the simulated microservice to be developed before it exists. That is, to develop the purchase management microservice, it would not be necessary to develop the user profile management and home delivery management microservices, it would be enough to simulate them.

The project will be carried out within NTT Data company following an agile methodology. The specification of the interface offered by each microservice will be carried out with OpenAPI. The project will be implemented on Spring and the Swagger Parser library will be used to process the content of the OpenAPI specifications.

Keywords

Microservices simulation, Spring Boot, Swagger, Microservices testing, Software development, REST.

Índice

Índice de figuras

1. Introducción, Objetivos y Planificación

1.1. Introducción

1.1.1. Contexto

Este Trabajo de Fin de Grado (TFG) fue realizado en el seno de la compañia NTT Data, una de las principales empresas de servicios de tecnología de la información en el mundo. NTT Data, con sede en Tokio, Japón, es una compañía líder en el sector de la tecnología que proporciona servicios de consultoría, implementación, gestión y entrega de proyectos en numerosos sectores industriales y comerciales. Con departamentos que abarcan desde la seguridad cibernética hasta la analítica de datos, la empresa se ha consolidado como una fuerza influyente en el escenario tecnológico mundial [2].

Mi contribución en este conglomerado tecnológico se desarrolló específicamente en el departamento de Multi APIs, un equipo dedicado a la creación y mantenimiento de interfaces de sistemas para una variedad de clientes y proyectos. En este departamento, tuve la oportunidad de trabajar para uno de nuestros clientes más destacados, *Santander Digital Services*, concretamente en el proyecto denominado *Integration Layer*.

En el proyecto Integration Layer mi papel consistía en trabajar como ingeniero de software, desarrollando microservicios para mejorar y agilizar las operaciones del cliente. Las responsabilidades de mi puesto implicaban no solo el diseño y desarrollo de microservicios, sino también la colaboración estrecha con otros equipos y departamentos para garantizar la integración sin fisuras de estos servicios en las operaciones más amplias del cliente.

Es importante destacar que, antes de asumir mi papel como ingeniero de software en el departamento de Multi APIs, tuve la oportunidad de realizar prácticas curriculares en la misma empresa. Durante mis prácticas, pude trabajar con Spring Boot y Apache Kafka, una plataforma de procesamiento de streams en tiempo real distribuida y de código abierto. Esta etapa de mi carrera fue de vital importancia para mi desarrollo profesional, ya que me permitió adquirir una valiosa experiencia práctica y establecer las bases para mi futuro trabajo en la empresa. Además, la experiencia práctica adquirida con estas tecnologías me proporcionó unos conocimientos sobre los cuales pude desarrollar y ampliar mis habilidades técnicas en el proyecto *Integration Layer*.

1.1.2. Descripción del Problema y caso de estudio

Este proyecto proviene principalmente del reto tecnológico que supone simular microservicios en entornos empresariales modernos. En la actualidad, las arquitecturas de software basadas en microservicios han ganado considerable popularidad debido a su escalabilidad, flexibilidad y capacidad para permitir un desarrollo más rápido y eficiente. Sin embargo, también presentan desafíos significativos en cuanto a la coordinación y comunicación entre los servicios.

Un desafío notable se encuentra en el ámbito de las pruebas y simulaciones. En los entornos de desarrollo y pruebas, mantener múltiples instancias de todos los microservicios implicados puede resultar en un proceso costoso y complejo. Para ilustrar estos conceptos, usaremos un caso de estudio específico que será el eje central de este TFG, en un entorno de desarrollo hipotético para *Santander Digital Services*. Supongamos que estamos trabajando con el microservicio *Decision Engine*, el cual depende en cierta medida del microservicio de *Disputas Visa* para realizar algunas

de sus funciones. Para poder probar adecuadamente el *Decision Engine*, sería necesario desplegar una instancia del servicio de *Disputas Visa* con su propia infraestructura. Esta necesidad puede incrementar sustancialmente tanto los costos como la complejidad del proceso.

Además, si se desea probar la funcionalidad del *Decision Engine* de manera aislada, las dependencias con el microservicio de *Disputas Visa* pueden generar complicaciones. Si este último no está disponible o experimenta problemas, la capacidad de probar de manera efectiva el *Decision Engine* se vería limitada. Esta interdependencia resalta los desafíos inherentes a la estructura de los microservicios y la necesidad de soluciones efectivas para su simulación y prueba.

Para enfrentar estos desafíos, existen varios mecanismos que nos permiten especificar de manera precisa las operaciones disponibles en un microservicio. Entre estos mecanismos se encuentran Open API y Swagger, que proporcionan una forma de describir la estructura de un microservicio a través de un archivo .yaml.

A partir de dichos archivos, podríamos desplegar un microservicio simulado que, para cada posible petición, devuelva siempre una respuesta estandarizada. Por ejemplo, en el microservicio de *Disputas Visa* para el sistema de banca en línea del *Santander Digital Services*, si se realiza una petición "GET /visa/disputes/638", en lugar de buscar los detalles de la disputa en la base de datos real, el microservicio simulado devolvería una respuesta estandarizada con detalles de una disputa ficticia, independientemente del "disputeId"proporcionado. De esta manera, se permitiría la realización de pruebas sin acceder a la base de datos real. Esto facilitaría en gran medida el desarrollo y las pruebas del microservicio *Decision Engine*.

Para crear este simulador de microservicios se desarrollará un recurso o *asset* usando Spring Boot, que permita desacoplar microservicios de manera eficiente y efectiva. Este *asset* utiliza archivos de especificaciones Swagger escritas en .yaml para comprender y simular la funcionalidad de los microservicios que se quieren desacoplar.

A grandes rasgos, el proyecto implica procesar los archivos .yaml y generar, a partir de estos, archivos que contienen las respuestas esperadas de los microservicios. Esta funcionalidad es crítica para facilitar las pruebas y el desarrollo de los microservicios, ya que evita la necesidad de que estos microservicios estén activos y comunicándose entre sí durante estas etapas.

Además, la aplicación también es capaz de interceptar y responder a las llamadas REST que se hacen a los microservicios simulados. Es decir, en lugar de que la llamada llegue al microservicio original, es recibida por el *asset*, que devuelve la respuesta predeterminada generada previamente.

En resumen, el TFG se enfoca en el desafío de desacoplar microservicios para pruebas y desarrollo, mediante el uso de Spring Boot para desarrollar un *asset* que genera y devuelve respuestas predeterminadas basadas en especificaciones Swagger. Este enfoque puede ayudar a mejorar la eficiencia y la productividad en los entornos de desarrollo de microservicios.

1.2. Objetivos

El propósito central de este trabajo es el diseño y la implementación de un *asset* utilizando el framework Spring Boot para simular microservicios a partir de sus esoecificaciones Open API realizadas mediate Swagger. Para alcanzar esta meta general, se han planteado los siguientes objetivos específicos:

- **Desarrollo e implementación de un sistema de procesamiento de archivos Swagger en formato .yaml**: El objetivo es crear un módulo capaz de leer y analizar archivos Swagger. Este subsistema deberá interpretar la estructura y las especificaciones de los microservicios representados en los archivos .yaml, y utilizar esta información para el proceso de generación de respuestas predeterminadas.
- **Generación automatizada de respuestas predeterminadas**: Basándose en los datos obtenidos del procesamiento de los archivos Swagger, la aplicación debe ser capaz de generar respuestas predeterminadas que simulan el comportamiento de los microservicios.
- **Implementación de una funcionalidad de intercepción de llamadas REST**: La aplicación debe ser capaz de interceptar las llamadas REST destinadas a los microservicios desacoplados. Esta funcionalidad requerirá la integración con el sistema de enrutamiento existente para redirigir correctamente las llamadas hacia el simulador de microservicios.
- **Validación de llamadas y entrega de respuestas predeterminadas**: Al recibir una llamada REST, la aplicación deberá validarla comparándola con las especificaciones de los archivos Swagger. Si la llamada es válida, se debe devolver la respuesta predeterminada correspondiente.

1.3. Planificación

1.3.1. Metodología Agile

Santander Digital Services, donde se lleva a cabo este proyecto, emplea la metodología Agile en sus procesos de desarrollo de software. La metodología Agile es un enfoque de desarrollo de software que enfatiza la entrega continua de valor, la colaboración entre equipos y la adaptabilidad al cambio.

El principio central de Agile es la adaptabilidad y la flexibilidad. En lugar de tratar de definir y cumplir con un conjunto rígido de requisitos desde el principio, Agile se enfoca en entregar iterativamente incrementos funcionales del software, lo que permite adaptarse a necesidades y requisitos cambiantes.

Agile se basa en cuatro valores fundamentales, según se establece en el Manifiesto Agile:

- Los individuos y las interacciones priman sobre los procesos y las herramientas.
- El software funcionando prima sobre la documentación extensiva.
- La colaboración con el cliente prima sobre la negociación contractual.
- La respuesta al cambio prima sobre seguir un plan.

Durante el desarrollo del proyecto, que duró un total de cuatro sprints de quince días cada uno, estos valores se materializaron en una serie de prácticas y eventos que resultaron particularmente significativos:

- **Planificación del Sprint:** Al comienzo de cada sprint, se realizaba una reunión de planificación para acordar los elementos del backlog del producto que se trabajarían en el sprint. Estas reuniones sirvieron para definir la ruta de trabajo para el período que seguía y para establecer las expectativas y responsabilidades de cada miembro del equipo. En mi caso, debido a que estas reuniones planificaban el funcionamiento normal de mi equipo en el próximo sprint y no el desarrollo de mi proyecto en concreto, mi participación era limitada.

- **Daily Meetings:** A lo largo de cada sprint, se llevaban a cabo reuniones diarias, en las cuales cada miembro del equipo reportaba sobre su progreso, los planes para el día y cualquier obstáculo que se hubiese presentado. En este proyecto, las Daily Meetings se convirtieron en el principal medio de interacción y coordinación con el equipo y el cliente para que estuvieran al tanto de mis avances.

- **Revisión del Sprint:** Al final de cada sprint, se realizaba una revisión para evaluar y mostrar el trabajo realizado durante el sprint. Aunque mi participación en estas revisiones fue más limitada debido a mi rol y a la naturaleza del proyecto, este ejercicio aseguró que todos los miembros del equipo estuvieran al tanto del progreso del proyecto. Al igual que en las planificaciones, en estas revisiones mi participación no era completa ya que mi trabajo llevaba una planificación diferente a la de mi equipo.

- **Retrospectiva del Sprint:** Después de la revisión del sprint, el equipo realizaba una retrospectiva para reflexionar sobre el desempeño durante el sprint y buscar formas de mejorar en el siguiente. Las retrospectivas nos permitieron aprender de nuestras experiencias y mejorar nuestro desempeño como equipo. A pesar de que mi proyecto no era el mismo que el del resto de mi equipo, ellos estaban informados sobre su desarrollo a través de nuestras reuniones diarias. Esta constante comunicación nos permitía prestarnos ayuda mutua cuando era necesario, por lo que era crucial que participara activamente en las dinámicas de grupo durante las sesiones de retrospectiva.

Además de estos eventos regulares, ocasionalmente se realizaron reuniones extraordinarias para el seguimiento del proyecto y demostraciones a equipos específicos. Estos encuentros permitieron presentar en detalle el avance del proyecto, recibir comentarios y ajustar las tareas en función de las necesidades surgidas. Las demostraciones ayudaron a garantizar que el desarrollo del proyecto se alineara con las expectativas del cliente y con los objetivos generales del equipo.

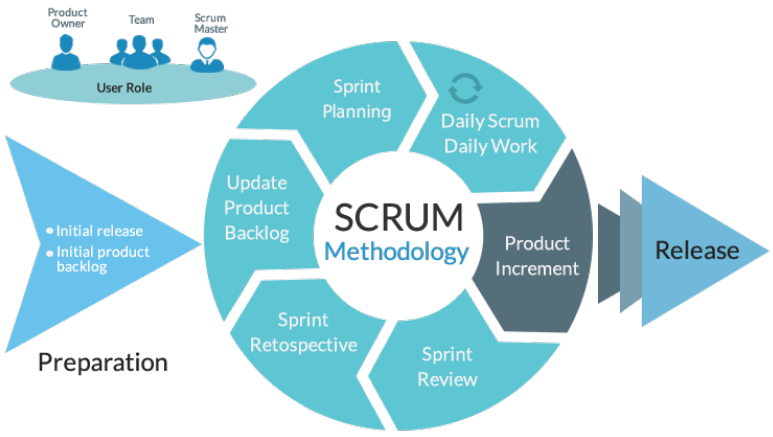

Figura 1: Esquema metodología Agile [1]

A pesar de las particularidades y adaptaciones que presentó la aplicación de una versión más laxa de la metodología Agile en este proyecto, los principios fundamentales de esta metodología, tales como la colaboración, retroalimentación constante, adaptabilidad y entrega continua de valor, se mantuvieron como guías constantes a lo largo del desarrollo del proyecto. Este enfoque "laxo"nos permitió mantener la esencia de Agile, adaptándola a las necesidades y características específicas de nuestro equipo y del proyecto en cuestión.

La aplicación de la metodología Agile a este proyecto no solo permitió una eficiente adaptación a las cambiantes necesidades de las partes interesadas, sino que también promovió una colaboración efectiva entre los miembros del equipo. Esta flexibilidad, en combinación con el compromiso de entregar continuamente valor a través de incrementos funcionales de software, fue fundamental para el éxito de este proyecto.

1.3.2. Gestión de la Configuración y Calidad

En cuanto a la gestión de la configuración y el control de versiones del código, el proyecto se apoyó en Git. Para llevar a cabo este control de versiones de una manera ordenada y coherente, el equipo adoptó el modelo de ramificación *GitFlow*, propuesto por Vincent Driessen [3].

Este modelo estructura el trabajo en dos ramas principales: *master* y *develop*. La rama *master* alberga las versiones oficialmente liberadas del producto, es decir, el código que se encuentra en producción. Por otro lado, la rama *develop* actúa como una rama de integración para las nuevas características o funcionalidades que se están desarrollando.

En el proceso de implementación de una nueva característica, se crea una rama auxiliar específica para ella, llamada *feature branch*, a partir de la rama *develop*. Todo el trabajo de desarrollo relacionado con esa característica se realiza en esta rama *feature*. Una vez finalizada la implementación de la característica, la rama *feature* se fusiona de nuevo con *develop*.

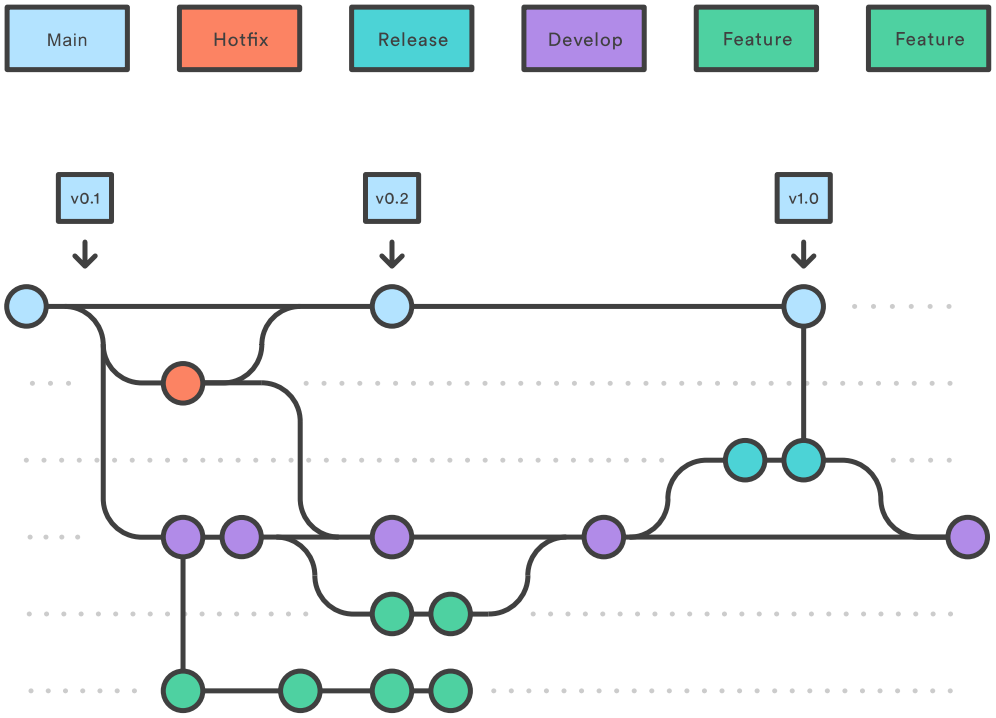

Figura 2: Flujo de trabajo de Gitflow propuesto por Vincent Driessen

Cuando el equipo está listo para lanzar una nueva versión del producto, se crea una rama release a partir de *develop*. Esta rama permite realizar cualquier ajuste necesario para la preparación del lanzamiento, como correcciones de errores menores o ajustes de documentación. Una vez que la versión está lista para el lanzamiento, la rama release se fusiona tanto con *master* (para reflejar la nueva versión en producción) como con *develop* (para asegurar que las futuras características se basen en la versión más reciente del software).

Para las correcciones de errores urgentes o críticos que deben implementarse en la versión de producción, se utilizan las ramas *hotfix*. Estas ramas se crean a partir de *master* y, una vez que la corrección se ha completado, se fusionan de nuevo con *master* y *develop*.

En lo que respecta a las ramas de *hotfix*, no fue necesario su uso durante el desarrollo de este proyecto. Esto se debió a que no surgieron con errores críticos o urgentes que requirieran ser resueltos de manera inmediata en la versión de producción. No obstante, de haber tenido que afrontar tales errores, las ramas de *hotfix* habrían sido la estrategia para abordar dichos problemas.

Este flujo de trabajo de GitFlow permite un desarrollo paralelo de características, preparación de lanzamientos y correcciones de errores, lo que facilita la gestión de la configuración y la calidad del código [4].

Además de Git, en el proyecto también se utilizó *Jenkins*, una herramienta de integración continua que facilita la automatización de varias fases del ciclo del desarrollo del software. *Jenkins* se encargaba de compilar y ejecutar una serie de pruebas automáticas cada vez que se realizaban cambios en el repositorio Git. Este proceso permitía detectar de manera temprana cualquier error de compilación o prueba fallida [5].

Para asegurar el cumplimiento de los estándares de calidad, *Jenkins* estaba configurado para

trabajar conjuntamente con *SonarQube*, lo que permitía realizar análisis estáticos de código de manera continua. Así, cada vez que Jenkins realizaba una compilación, se desencadenaba un análisis de SonarQube que evaluaba la calidad del código, detectando posibles bugs, malas prácticas, vulnerabilidades de seguridad y problemas de deuda técnica [6].

Además de este análisis continuo, se definió un umbral de calidad específico que el código debía cumplir. Uno de los indicadores más importantes de este umbral era la cobertura de las pruebas unitarias. Se estableció que la cobertura de estas pruebas debería alcanzar, al menos, el 80 %. Una alta cobertura de pruebas ayuda a detectar y prevenir errores en el código antes de que lleguen a producción.

En nuestro proyecto, la interconexión entre *Jenkins*, *SonarQube* y *GitFlow* fue crítica para mantener un alto nivel de calidad y eficiencia en el desarrollo de software. Cada línea de código escrita para una feature branch debía superar con éxito las compilaciones y pruebas en Jenkins antes de fusionarse con la rama *develop*. Si alguna de estas validaciones resultaba en un fallo, se rechazaba la fusión de la rama, por lo que se debía corregir el código y someterlo de nuevo al proceso de evaluación [3].

La sinergia entre estas herramientas proporcionó un entorno de desarrollo ágil y de alta calidad, permitiendo el rápido desarrollo de características y la pronta detección y corrección de errores y problemas de calidad. Esta combinación de metodologías y herramientas fue fundamental en el éxito del proyecto, logrando mantener la calidad del código en un alto nivel y asegurando que las funcionalidades fueran entregadas de manera eficiente y efectiva.

2. Requisitos

2.1. Contexto

Para comprender mejor el contexto de este proyecto es necesario abordar el concepto de un archivo *Swagger*. Swagger es una herramienta de código abierto que permite a los desarrolladores especificar servicios REST. Los archivos Swagger, generalmente escritos en formato YAML o JSON, se utilizan para describir las API RESTful [7].

Estos archivos incluyen detalles como los *endpoints* o métodos de la API, sus parámetros de entrada y salida, los métodos HTTP aceptados (GET, POST, PUT, DELETE, etc.) y las respuestas proporcionadas con sus los códigos de estado HTTP asociados, entre otros.

```yaml
swagger: "2.0"
info:
  version: "1.0.0"
  title: Account Management Service
paths:
  /accounts/{accountId}:
    get:
      summary: Retrieves account information
      parameters:
      - name: accountId
        in: path
        required: true
        type: string
      responses:
        200:
          description: Account information retrieved successfully
          schema:
            $ref: '#/definitions/Account'
definitions:
  Account:
    type: object
    properties:
      accountId:
        type: string
      balance:
        type: number
```

Figura 3: Ejemplo de un archivo de especificación de Swagger para un servicio de gestión de cuentas.

Para explicar el funcionamiento de un archivo Swagger utilizamos el ejemplo de la Figura 3 que especifica un bien y servicio de gestión de cuentas de usuario. La estructura de un archivo Swagger está compuesta por diferentes secciones, cada una con un propósito específico para describir la API REST.

- swagger: (Figura 3, linea 1) Define la versión de Swagger utilizada para el documento. En

este caso, la versión es “2.0”.

- info: (Figura 3, lineas 2 - 4) proporciona información general sobre la API, incluyendo su versión (“1.0.0”) y título (“Account Management Service”).
- paths: Esta es una de las secciones más importantes del archivo Swagger, ya que define recursos de la API y las operaciones que se pueden realizar en ellos. Aquí, se define un único recurso (“/accounts/accountId”) con una operación GET (Figura 3, lineas 5 - 18). Esta operación permite recuperar la información de una cuenta específica.
- parameters: En la operación GET definida, se describe un parámetro, “accountId”, que debe proporcionarse en la ruta de la URL (Figura 3, lineas 9 - 13). Este parámetro es necesario (como indica la propiedad required: true) y debe ser de tipo cadena (type: string).
- responses: Define las respuestas que puede devolver la operación. En este caso, se describe una única respuesta con código de estado HTTP 200, que indica que la información de la cuenta se ha recuperado con éxito (Figura 3, lineas 14 - 18). La respuesta incluye un objeto de cuenta, cuya estructura se define en la sección definitions.
- definitions: En esta sección se describe el modelo de datos utilizado por el servicio. En el caso concreto del ejemplo de la Figura 3 (lineas 19 - 26), se muestra la clase cuenta utilizada por el único método de esta API. Esta clase tiene dos propiedades, “accountId”(de tipo cadena) y “balance”(de tipo numérico).

Este archivo Swagger, por lo tanto, proporciona una descripción completa y detallada del servicio de gestión de cuentas. Cualquier usuario o desarrollador que tenga acceso a este archivo puede entender fácilmente cómo interactuar con el servicio, qué operaciones están disponibles, qué parámetros se necesitan para cada operación y qué respuestas se pueden esperar.

2.2. Proceso de captura de requisitos

En cuanto a los requisitos para este proyecto, cabe mencionar que estos ya venían predefinidos al principio del proyecto. Por lo tanto, no fue necesario realizar una fase de captura de requisitos como tal, lo que me permitió comenzar directamente con la fase de diseño y desarrollo. Sin embargo, estos requisitos predefinidos han servido como guía constante a lo largo del proyecto, definiendo las características y funcionalidades que la aplicación debía cumplir.

2.3. Fuente de los Requisitos Funcionales

Los requisitos funcionales descritos en la siguiente sección han sido derivados de un documento que me fué proporcionado previamente. Este documento contiene una variedad de información que incluye descripciones textuales de requisitos, especificaciones de la API REST y directivas de implementación. Es decir, incluía tanto unos requisitos descritos en lenguaje natural como una serie de bocetos y pistas iniciales para su diseño e implementación.

Aunque la información en este documento es diversa, cada pieza de información contribuye a formar una comprensión más completa de lo que se espera del proyecto. Estas son algunas de las formas en que la información del documento se ha utilizado para informar los requisitos funcionales:

- **Descripciones textuales de requisitos**: Estas descripciones proporcionaron una visión general de las funciones y comportamientos deseados del sistema. Ayudaron a formar la base de los requisitos funcionales, proporcionando una descripción de alto nivel de las capacidades necesarias.

  Por ejemplo, uno de los requisitos funcionales se puede describir de la siguiente manera:

  Requisito: Gestión de microservicio simulado

  Descripción: El sistema debe ser capaz de crear, configurar y eliminar microservicios simulados basados en archivos Swagger. Los usuarios deben poder proporcionar un archivo Swagger que describa el comportamiento del microservicio que desean simular. Una vez proporcionado, el sistema debe ser capaz de interpretar este archivo y crear un microservicio simulado correspondiente. Los usuarios también deben poder configurar la respuesta que este microservicio simulado proporciona a cada petición. Además, los usuarios deben poder eliminar un microservicio simulado cuando ya no sea necesario.

  Esta descripción textual de un requisito esencial sirve como una guía clara para la implementación del sistema, y establece las funcionalidades esperadas en el desarrollo del proyecto.

- **Especificaciones de la API REST**: Estas especificaciones proporcionaron detalles técnicos cruciales sobre cómo debería interactuar el sistema con otros sistemas y cómo debería manejar y responder a las solicitudes REST. Estos detalles se han incorporado en varios requisitos funcionales, como la capacidad de interceptar y responder a las llamadas REST, y la validación de estas llamadas.

- **Directivas de implementación**: Estas directivas proporcionaron instrucciones más específicas sobre cómo se deben implementar ciertas funciones o características del sistema. Esto ha dado lugar a decisiones de diseño que concretaremos más adelante.

El proceso de extraer requisitos funcionales de este documento ha requerido interpretar y sintetizar la información proporcionada, y traducirla a un conjunto de características y comportamientos concretos que el sistema debe tener.

2.4. Requisitos funcionales

Este proyecto debía realizar únicamente dos grandes requisitos funcionales o casos de uso, los cuales se describen a continuación:

1. **RF1: Despliegue del servicio simulado**: Este requisito funcional implica el procesamiento de archivos Swagger para la generación de todos los artefactos que fuesen necesarios para simular el servicio especificado dentro de dichos archivos. Este análisis debe permitir identificar las especificaciones clave de los microservicios, incluyendo rutas, métodos, parámetros, y respuestas esperadas. A partir de la información obtenida de los archivos Swagger, el sistema debe generar respuestas predeterminadas que simulen el comportamiento de los microservicios. Cada respuesta generada debe corresponder a un *endpoint* del microservicio.

2. **RF2: Intercepción y validación de llamadas REST**: Este requisito funcional engloba la interceptación de llamadas REST, la validación de estas y la entrega de respuestas

predeterminadas. El sistema debe poder interceptar las llamadas REST que se dirigen a los microservicios simulados. Al interceptar una llamada REST, el sistema debe validarla en base a las especificaciones extraídas del archivo Swagger correspondiente. La validación debe verificar que la ruta, el método y los parámetros de la llamada coincidan con los especificados en el archivo Swagger. Si una llamada REST es validada exitosamente, el sistema debe entregar la respuesta predeterminada correspondiente.

2.5. Requisitos No Funcionales

Los requisitos no funcionales juegan un papel crucial en la calidad y el rendimiento de un sistema de software. Pueden afectar la experiencia del usuario, la capacidad de mantenimiento del sistema y la facilidad con la que se puede escalar o adaptar el sistema para satisfacer futuras demandas.

Sin embargo, en el caso de este proyecto, después de una cuidadosa evaluación de la ISO 25010, se ha determinado que no hay requisitos no funcionales relevantes dentro de este proyecto.

El estándar ISO 25010 proporciona un modelo para evaluar y describir varias características de calidad del software. Este modelo, conocido como Sistema de Calidad del Producto de Software (SQuaRE), incluye características como la funcionalidad, la eficiencia en el rendimiento, la compatibilidad, la usabilidad, la fiabilidad, la seguridad, la mantenibilidad y la portabilidad [8].

Aunque estas características son relevantes para muchos proyectos de software, en el caso de este Trabajo de Fin de Grado, se ha determinado que los requisitos no funcionales basados en el modelo ISO 25010 no precisan de un tratamiento especial más allá del que se supone en cualquier proyecto de ingeniería del software. Esto se debe a varios factores:

- El enfoque principal de este proyecto está en los aspectos funcionales del sistema, es decir, en el desarrollo de un activo que pueda manejar eficientemente la simulación de microservicios. Las características del modelo ISO 25010, en su mayoría, tratan sobre aspectos no funcionales y, por lo tanto, no son directamente relevantes para los objetivos de este proyecto.

- Este proyecto no implica el desarrollo de una aplicación de software completa, sino de un microservicio específico para ser utilizado en el contexto de un sistema de microservicios más grande. Por tanto, muchos de los requisitos no funcionales comúnmente asociados con las aplicaciones de software (como la usabilidad o la portabilidad) no son relevantes en este contexto.

- El proyecto está orientado a resolver un problema técnico específico en lugar de a satisfacer las necesidades de los usuarios finales. Como resultado, las características relacionadas con la experiencia del usuario, como la usabilidad o la accesibilidad, no son pertinentes en este caso.

Sin embargo, es importante destacar que aunque el modelo ISO 25010 no se aplique en su totalidad, este proyecto contaba con una serie de requisitos no funcionales estándar y básicos que se pueden esperar de cualquier proyecto informático profesional.

Estos requisitos no funcionales estándar incluían aspectos como su capacidad de integración con otros sistemas. La capacidad de integración del activo, es decir, su capacidad para interactuar

e intercambiar información con otros sistemas y componentes, era un requisito crítico dada su función de soporte a la arquitectura de microservicios.

Por tanto, aunque los requisitos no funcionales basados en el modelo ISO 25010 no se aplicaban por completo, se siguieron considerando y cumpliendo varios requisitos no funcionales clave, asegurando así que el proyecto mantuviera un alto estándar de calidad y profesionalidad.

3. Arquitectura e implementación

3.1. Arquitectura

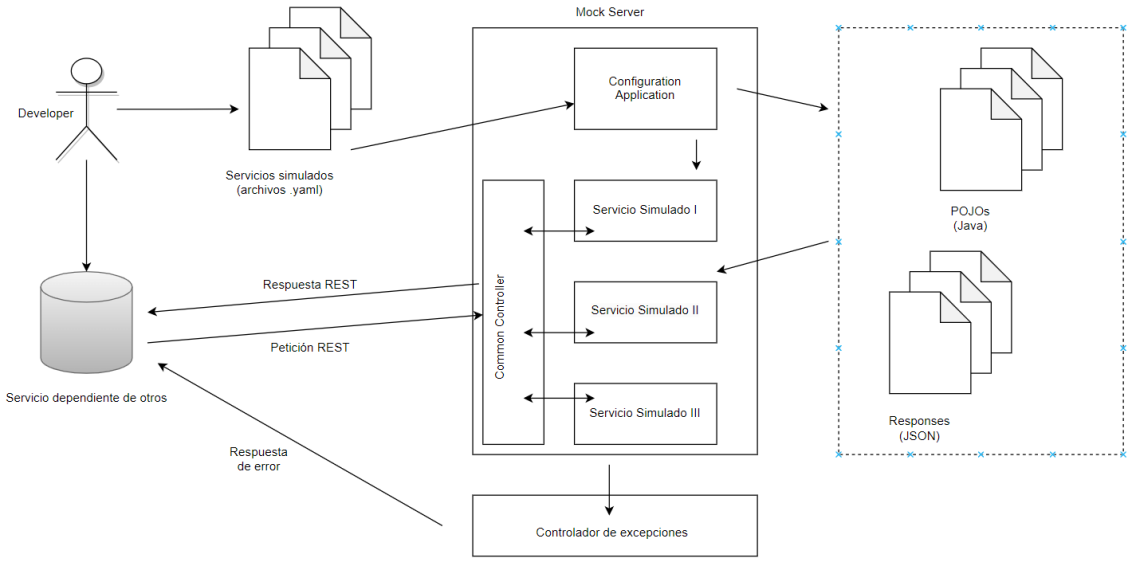

Figura 4: Esquema de la arquitectura

La arquitectura del sistema desarrollado en este Trabajo de Fin de Grado se muestra en la Figura 4.

El punto de partida para comprender la arquitectura es un microservicio específico dependiente de otros. Siguiendo con el ejemplo de la introducción, este servicio sería *Decision Engine*. Este microservicio tiene una serie de dependencias de otros microservicios, como ya hemos vito anteriormente un ejemplo seria *Disputas Visa*. Estas dependencias son esenciales para el funcionamiento de *Decision Engine* y están especificadas de manera detallada en archivos .yaml utilizando las normativas de Swagger y Open API.

La pieza central de nuestra arquitectura es la aplicación llamada *MockService* que es la pieza que simula los microservicios. *MockService* se despliega a través de una clase llamada *ConfigurationApplication* implementada como un Bean de Spring que se ejecuta al arrancar la aplicación y que realiza su configuracón.

ConfigurationApplication se encarga de procesar los archivos Swagger para crear clases Java simples, es decir, POJOs, para validar los datos de entrada y archivos que almacenen las respuestas predeterminadas a cada petición.

Una vez que *MockService* ha sido desplegado y configurado, está listo para interceptar las llamadas REST dirigidas a los microservicios que ha simulado.

Para ilustrar el funcionamiento de esta arquitectura, utilizaremos como ejemplo una petición concreta que se podría realizar dentro de nuestro caso de estudio. En esta petición, el servicio de toma de decisiones necesitaría recuperar una disputa concreta del servicio de visas. Para ello ejecuta una petición REST, por ejemplo “GET/disputas/657”, que correspondería a uno de los

microservicios simulados que se han definido en los archivos .yaml, como *Disputas Visa*. Esta petición es recogida por *CommonController*.

CommonController es responsable de verificar que la URI de la solicitud es válida y corresponde a uno de los microservicios simulados. Además, utiliza los POJOs generados para ese servicio para validar que todos los datos de la petición sean correctos. Por ejemplo, que las fechas sean válidas. Si la solicitud se valida correctamente, *CommonController* procede a devolver la respuesta predeterminada correspondiente.

Es importante notar que estas respuestas predeterminadas no son arbitrarias, sino que han sido creadas para reflejar respuestas reales que los microservicios simulados podrían proporcionar. Por ejemplo, una respuesta predeterminada podría ser una representación JSON de una tarjeta de crédito.

También mencionar que existe un controlador de excepciones encargado de interceptar cualquier tipo de error producido durante el funcionamiento de la aplicación y devolver un mensaje de error si algo ha ido como se esperaba.

3.2. Componente de Configuración

El Componente de Configuración de la aplicación se implementa a través de la clase *ConfigurationApplication*, un componente clave que realiza varias operaciones esenciales para preparar la aplicación para su correcto funcionamiento. La clase *ConfigurationApplication* es un *@Bean* que se ejecuta al desplegar la aplicación y tiene dos responsabilidades principales:

1. Generar POJOs (Plain Old Java Objects) que representen los datos a validar en las solicitudes REST. Estos POJOs se crean a partir de la información recogida de los archivos Swagger. Su objetivo principal es encapsular la lógica de validación de las solicitudes entrantes.

2. Crea archivos de respuestas predeterminadas. Estas respuestas se diseñan para simular las respuestas que podrían provenir de los microservicios dependientes en un escenario real. Estas respuestas predeterminadas se almacenan y están listas para ser utilizadas una vez que la aplicación se haya desplegado completamente.

Ambas funciones se llevan acabo mediante la lectura de los archivos Swagger que se lleva a cabo mediante el uso de la libreria *Swagger Parser*. Esta librería permite extraer la infomación que necesitamos de los archivos. En estos archivos es donde viene especificados los ejemplos y la lógica de validación de los POJOs.

Para la generación de las clases POJO (Plain Old Java Object) que se utilizan en la validación de las peticiones, se hace uso de la herramienta *OpenAPI Generator*. Esta herramienta permite traducir las especificaciones Swagger de los microservicios simulados en clases POJO Java.

En la Figura 5 adjunta, se muestra un ejemplo de un POJO generado para la gestión de una tarjeta Visa.

```java
public class VisaCard {

    @NotNull(message = "Card number cannot be null")
    @Pattern(regexp="^4[0-9]{12}(?:[0-9]{3})?$", message="Card number must be valid")
    private String cardNumber;

    @NotNull(message = "Cardholder name cannot be null")
    @Size(min = 1, max = 30, message = "Cardholder name must be between 1 and 30 characters")
    private String cardholderName;

    @NotNull(message = "Expiration date cannot be null")
    @Future(message = "Expiration date must be in the future")
    private Date expirationDate;

    @NotNull(message = "CVV cannot be null")
    @Pattern(regexp="^[0-9]{3}$", message="CVV must be exactly 3 digits")
    private String cvv;

    // Getters and setters ...

}
```

Figura 5: Ejemplo de un POJO para una tarjeta Visa

Las anotaciones de `javax.validation` en el POJO `VisaCard` (Figura 5) proporcionan reglas de validación para los campos de la clase:

- La anotación `@NotNull` (líneas 3, 7, 11, 15) indica que el campo no debe ser `null`. El atributo `message` permite personalizar el mensaje de error que se mostrará si el campo es `null`.
- La anotación `@Pattern` (líneas 4, 16) especifica que el campo debe coincidir con la expresión regular proporcionada. En este caso, se utiliza para validar el formato del número de la tarjeta y el código de seguridad (CVV).
- La anotación `@Size` (línea 8) se utiliza para especificar que el tamaño del nombre del titular de la tarjeta debe estar entre 1 y 30 caracteres.
- La anotación `@Future` (línea 12) se utiliza para verificar que la fecha de vencimiento de la tarjeta sea en el futuro.

Estas anotaciones son útiles para validar automáticamente los datos antes de que sean procesados, permitiendo detectar errores de forma temprana.

En lo que respecta a las respuestas predeterminadas, estas se almacenan en archivos *.properties* cuyo nombre se deriva de la operación y la URI, por ejemplo *GET-cuenta/idcuenta.properties*. Cada

archivo *.properties* contiene una única propiedad, *output*, que almacena la respuesta predeterminada correspondiente.

En la Figura 6 se muestra un ejemplo de un archivo *.properties*.

```
output={
    "accountId" : "123456789",
    "balance" : 5000.0
}
```

Figura 6: Ejemplo de archivo .properties

La Figura 6 muestra una hipotética respuesta con un JSON (Línea 1) con dos campos ficticios: "accountId"(Línea 2) y "balance"(Línea 3), que simulan un identificador de cuenta y un balance respectivamente.

Gracias a este proceso de configuración, la aplicación puede conocer y validar las peticiones REST, y devolver las respuestas predeterminadas apropiadas. De esta manera, la aplicación simula de forma eficaz el comportamiento de los microservicios, facilitando el desarrollo y las pruebas de otros componentes que dependen de ellos.

3.3. Controlador Común

El Controlador Común actúa como el intermediario entre el microservicio que realiza la llamada REST y el servicio de nuestra aplicación que se encarga de procesar y responder a dicha llamada. A continuación, describiremos el algoritmo que rige el funcionamiento del Controlador Común paso a paso.

1. **Recepción de llamadas REST**: El Controlador Común recibe las llamadas REST que se realizan a los microservicios simulados. Cabe destacar que este componente está preparado para recibir cualquier tipo de llamada REST, independientemente de su operación o ruta.

```
@RequestMapping(value = "**")
```

Figura 7: Ejemplo de archivo de la anotación del controlador

Como se puede ver en Figura 8 la propiedad *value = "**"* de la anotacion *@RequestMapping* define la URL o las rutas para las que el método controlador debería procesar las solicitudes. En este caso, el patrón ** actúa como un comodín, lo que significa que el método controlador gestionará todas las rutas.

2. **Validación de la existencia de la llamada**: Una vez recibida una llamada REST, el Controlador Común pasa la información de esta al servicio. El primer paso que realiza el servicio es verificar que la llamada corresponde a una de las operaciones de los microservicios simulados.

Esto se hace buscando la URI de la llamada junto con el método por ejemplo "GET-disputas/657" entre las respuestas almacenadas. Si la llamada no corresponde a ninguna operación conocida, el servicio generará un error.

3. **Validación del contenido de la llamada**: Si la llamada corresponde a una operación conocida, el servicio procede a validar su contenido. Para ello, localiza el POJO que corresponde a dicha operación mediante Java Reflection y realiza un mapeo del contenido de la llamada a este POJO. En caso de que se produzcan errores durante este mapeo (por ejemplo, si el contenido de la llamada no se ajusta a lo especificado en el POJO), el servicio generará un error.

   La Figura 8 presenta una implementación de un método "`mapper`" en Java, diseñado para convertir una cadena de texto en formato JSON a una instancia de una clase específica.

```
1  public void mapper(String jsonString, String nombreClase) throws Exception
       {
2
3                  Class<?> objectClass = null;
4                  Object jSONObject = null;
5                  try {
6                          objectClass = Class.forName("com.package." +
                               nombreClase);
7                          // Creamos el Mapper
8                          ObjectMapper mapper = new ObjectMapper();
9                          // Incluimos el modulo de Java Time y mapeamos
10                         jSONObject = mapper.registerModule(new
                               JavaTimeModule()).readValue(jsonString,
                               objectClass);
11                 } catch (Exception e) {
12                         //Lanzamos una excepcion personalizada dependiendo
                               del error
13                 }
14 }
```

Figura 8: Ejemplo de Java Reflection y del mapper

El método está estructurado en varias secciones:

- En la línea 6, se utiliza Reflection de Java para obtener una referencia a la clase deseada. El método "Class.forName()" se utiliza para obtener dicha clase a partir de su nombre, que se concatena con el prefijo "com.package.". La referencia a la clase se almacena en la variable "objectClass".
- En la línea 10, se utiliza la biblioteca Jackson para mapear la cadena de texto JSON a una instancia de la clase especificada. Primero, se crea un objeto de la clase "ObjectMapper", que proporciona funcionalidades para convertir entre objetos Java y representaciones JSON. Después, se registra el módulo "JavaTimeModule" en el objeto

"ObjectMapper" para poder manejar el mapeo de fechas y horas. Finalmente, se llama al método "readValue()" en el objeto "ObjectMapper", pasándole la cadena de texto JSON y la referencia a la clase deseada. El método "readValue()" analiza la cadena de texto JSON y crea una nueva instancia de la clase especificada que contiene los datos de la cadena de texto JSON.

- En la línea 12, se captura y maneja cualquier excepción que pueda ocurrir durante el proceso de mapeo. Si se produce una excepción, el método lanzará una excepción personalizada dependiendo del error específico.

4. **Búsqueda del HTTPStatus**: A continuación, el servicio busca el HTTPStatus que debe devolver en respuesta a la llamada. Este código de estado HTTP se obtiene de las especificaciones de la operación correspondiente.

5. **Generación de la respuesta**: Si todo ha ido bien hasta este punto, el servicio procede a generar la respuesta. Para ello, localiza el archivo .properties que corresponde a la operación y extrae el contenido de la propiedad *output*. Este contenido, que se encuentra en formato JSON, se incorpora en una ResponseEntity junto con el código de estado HTTP obtenido previamente.

```
1 new ResponseEntity<>(responseContent,
    HttpStatus.valueOf(Integer.parseInt(code));
```

Figura 9: Ejemplo de creación de una ResponseEntity

En la Figura 9 se puede ver como se realiza esto, creando una nueva ResponseEntity con el con el contenido de la respuesta y el código de estado obtenido a partir de un String del .yaml.

6. **Devolución de la respuesta**: Por último, el servicio devuelve la ResponseEntity generada al Controlador Común, que la envía como respuesta a la llamada REST inicial.

Este algoritmo permite procesar las llamadas REST, validándolas y generando respuestas adecuadas basándose en las especificaciones de los microservicios simulados.

3.4. Controlador de Excepciones

El *Controlador de Excepciones* se encarga de gestionar las situaciones en las que se producen errores o excepciones durante el procesamiento de las llamadas REST. Su objetivo es proporcionar respuestas útiles y descriptivas a las llamadas que han provocado una excepción, facilitando así la identificación y resolución de problemas.

El funcionamiento del Controlador de Excepciones se basa en excepciones personalizadas. Durante el procesamiento de una llamada REST, si se genera una excepción, el servicio que la gestiona lanza una de estas excepciones personalizadas. Esta excepción contiene información sobre el error producido, incluyendo un código de error único y un mensaje de error descriptivo.

Cuando se lanza una excepción personalizada, el Controlador de Excepciones la captura e inicia su proceso de gestión. Los pasos que sigue son los siguientes:

1. **Detección de la excepción**: El Controlador de Excepciones detecta la excepción personalizada lanzada y extrae la información de error contenida en ella.

2. **Lectura de los errores definidos**: A continuación, el Controlador de Excepciones busca el archivo .properties que corresponde al código de error extraído de la excepción. Este archivo contiene una definición detallada del error, incluyendo su código, un mensaje descriptivo, una descripción más detallada y un nivel de gravedad.

   En la Figura 10 podemos ver un ejemplo hipotético de uno de estos archivos de error.

```
error.level=warning
error.code=CD006
error.msg=undefined property
error.description=undefined property violation
```

Figura 10: Ejemplo de archivo .properties de un error

3. **Generación de la respuesta de error**: Con la información obtenida del archivo .properties, el Controlador de Excepciones genera una respuesta de error. Esta respuesta se presenta en formato JSON y contiene un array de errores. Cada error del array incluye la información detallada del error obtenida del archivo .properties.

4. **Devolución de la respuesta de error**: Por último, el Controlador de Excepciones devuelve la respuesta de error generada. Esta respuesta se envía como respuesta a la llamada REST que ha provocado la excepción, sustituyendo así a la respuesta que se habría enviado en caso de que la llamada se hubiera procesado correctamente.

```
{
    "errors": [
        {
            "level": "warning",
            "code": "CD006",
            "message": "undefined property",
            "description": "undefined property violation"
        }
    ]
}
```

Figura 11: Ejemplo de archivo .properties de un error

La Figura 11 muestra un ejemplo de devolución de respuesta para una excepción que solo contiene el error de Figura 10.

Este mecanismo permite que nuestra aplicación gestione las situaciones de error, proporcionando respuestas detalladas que facilitan la identificación y resolución de problemas. Además, gracias al uso de excepciones personalizadas y archivos .properties para la definición de errores, este mecanismo puede adaptarse fácilmente para gestionar nuevos tipos de errores.

4. Pruebas y despliegue

4.1. Pruebas

Para cumplir con los estándares de calidad establecidos en el inicio del proyecto, que especificaban una cobertura mínima de pruebas del 80 %, se diseñaron, implementaron y ejecutaron una serie de pruebas unitarias.

Para la implementación de estas pruebas se decidió utilizar JUnit 5 y Mockito, dos frameworks de pruebas ampliamente reconocidos y utilizados en el mundo de Java.

Con estos frameworks se escribieron más de 50 pruebas unitarias con el objetivo de verificar la funcionalidad de cada componente de la aplicación. La escritura y mantenimiento de estas pruebas requirió una estructura de organización robusta y coherente, que se implementó mediante la creación de una carpeta de recursos dedicada únicamente a las pruebas. En esta carpeta, se incorporaron varios archivos de apoyo que se utilizaron durante las pruebas, como datos de entrada simulados, respuestas esperadas y otros recursos relevantes.

```
1  @Test
2  void
       givenUnexpectedException_whenCallingCommonController_thenReturnStatus()
3          throws Exception {
4
5      // given - precondition or setup
6      String level = "error";
7      String message = "message";
8      String description = "description";
9      String code = "code";
10     CustomError ce = new CustomError(code, description, message, level);
11     HttpStatus ht = HttpStatus.ACCEPTED;
12
13     // when - action or behavior that we are going test
14     CustomException e = new CustomException(Collections.singletonList(ce),
           ht);
15     Mockito.when(mockController.commonController(any(),
           any())).thenThrow(e);
16
17     // then - verify the result or output using assert statements
18     mockMvc.perform(get("/api/status"))
19             .andDo(print()).andExpect(status().isAccepted());
20 }
```

Figura 12: Ejemplo de un test para el controlador

La Figura 12 presenta una prueba unitaria en la cual se prueba el comportamiento del controlador ante una excepción inesperada. La prueba se organiza en tres secciones principales:

- En las línea 5 - 11, se configura el estado inicial del sistema antes de realizar la prueba. Aquí

se define un objeto “CustomError”que encapsula la información de un error, incluyendo un nivel, mensaje, descripción y código. Este objeto se utiliza para crear una excepción “CustomException”, la cual se lanzará durante la prueba.

- En las líneas 13 - 15, se especifica la acción que se está probando. En este caso, se está probando el comportamiento del método “commonController”del “mockController” cuando se lanza una “CustomException”. Para simular este comportamiento, se utiliza el método “when”de Mockito para especificar que cuando se llame al método “commonController”, debe lanzarse la excepción “CustomException”previamente configurada.

- Finalmente, en las líneas 17 - 19, se verifica que el resultado de la prueba es el esperado. Aquí se realiza una solicitud GET al endpoint utilizando “mockMvc.perform()”, que simula la ejecución de una solicitud HTTP en el controlador. Luego se verifica que el estado de la respuesta HTTP es el esperado (en este caso, “HttpStatus.ACCEPTED”) con el método “andExpect(status().isAccepted())”.

En este proyecto, además de los test unitarios implementados con JUnit5 y Mockito, se utilizaron Jenkins y SonarQube para realizar pruebas de construcción y análisis de calidad de código.

En este proyecto, Jenkins se utilizó para ejecutar pruebas de construcción cada vez que se realizaba un commit en el repositorio de Git.

Las pruebas de construcción aseguran que el código fuente puede ser compilado correctamente y que el proyecto se construye sin errores. Esto es esencial para detectar problemas tempranos que podrían hacer que la aplicación falle en tiempo de ejecución. Si Jenkins detecta algún error durante la construcción, el equipo de desarrollo es notificado inmediatamente para que pueda corregir el problema.

Además de las pruebas de construcción, Jenkins fue responsable de ejecutar SonarQube, una herramienta de inspección de calidad de código. SonarQube realiza análisis automáticos de la calidad del software. A través de este análisis, se pudo asegurar que el código cumpliera con los estándares de calidad establecidos.

En el analisis final el código de la aplicación quedó completamente libre de errores de sonar y un 80 % de cobertura como se pedía.

4.2. Despliegue

En este proyecto, la versión final de la aplicación se subió a un repositorio Git determinado, convirtiéndola en una herramienta accesible para diversos equipos dentro de Santander Digital Services, entre ellos el equipo de *Integration Layer*.

Este proceso de despliegue va más allá de solo poner a disposición el software. Se tomaron medidas para asegurar que los usuarios potenciales puedan entender y usar la aplicación de manera efectiva. Por lo tanto, al terminar el desarrollo de la aplicación, se incluyó un archivo README.md en el repositorio de Git. Este archivo proporciona una descripción detallada de la aplicación, instrucciones claras sobre su operación y guías para su correcta utilización. Esto incluye información sobre cómo configurar y ejecutar la aplicación, los requisitos del sistema, las dependencias de software y cualquier otra información pertinente que pueda facilitar su uso a los equipos de desarrollo.

Para ello el README.md esta compuesto de 7 secciones:

- **Descripción:** En esta sección, se proporciona una visión general de la aplicación. Esto incluye una descripción de las funciones y capacidades principales de la aplicación, su propósito y los problemas que resuelve.
- **Cómo descargar:** Aquí se dan las instrucciones detalladas sobre cómo descargar el código fuente de la aplicación desde el repositorio Git. Esto incluye comandos específicos de Git, como "git clone", junto con la URL del repositorio.
- **Configuración:** En esta sección, se proporcionan detalles sobre cómo configurar la aplicación después de su descarga.
- **Cómo ponerlo en funcionamiento:** Aquí se describen los pasos necesarios para poner en marcha la aplicación.
- **Funcionamiento:** Esta sección ofrece una explicación de cómo utilizar la aplicación, su flujo de trabajo y cómo interactuar con ella.
- **Ejemplos:** Aquí se proporcionan ejemplos prácticos de uso de la aplicación.
- **Soporte:** Finalmente, en esta sección se proporciona información sobre dónde y cómo obtener ayuda si los usuarios encuentran problemas al utilizar la aplicación.

Cada una de estas secciones juega un papel fundamental para asegurar que los usuarios potenciales puedan entender y usar la aplicación de manera efectiva.

5. Conclusiones

5.1. Sumario

A lo largo del desarrollo de este Trabajo de Fin de Grado se ha llevado a cabo un estudio y aplicación de diversas tecnologías y metodologías para abordar el desafío de simular microservicios en entornos empresariales modernos. Este desafío, a pesar de ser complejo, ha permitido la obtención de resultados significativos, destacando la creación de una aplicación sólida, funcional y de alta calidad que ahora está a disposición de varios equipos de desarrollo dentro de *Santander Digital Services*.

Desde la fase inicial de planificación hasta el despliegue final, cada etapa del proyecto ha estado marcada por la metodología Agile, aunque de manera laxa, y las diversas buenas prácticas de desarrollo de software. Este enfoque metodológico, junto con las herramientas utilizadas como Jira, Git, Jenkins y SonarQube, ha permitido una gestión eficiente del proyecto, la colaboración fluida dentro del equipo y la entrega continua de valor.

El trabajo realizado incluye el uso de tecnologías como Open API y Swagger para especificar operaciones disponibles en microservicios, la implementación de pruebas unitarias con JUnit5 y Mockito para garantizar la calidad y la eficacia de la aplicación y, finalmente, la creación de una guía de usuario detallada para facilitar la adaptación y utilización de la aplicación por parte de otros equipos.

5.2. Trabajos futuros

En cuanto a los trabajos futuros, existen varias oportunidades para la expansión y mejora del proyecto. A continuación, se detallan algunas de estas posibles vías de desarrollo.

Uno de los posibles trabajos futuros sería la adaptación de la aplicación para trabajar con otras tecnologías de microservicios más allá de Open API y Swagger. A medida que las tecnologías evolucionan, se podrían añadir nuevos adaptadores para soportar otros lenguajes de descripción de interfaz como gRPC [9] o GraphQL [10], aumentando así la utilidad y aplicabilidad del software.

Además, podría considerarse la incorporación de una interfaz de usuario. Aunque la aplicación fue diseñamente principalmente para uso interno, un panel de control intuitivo y amigable podría hacer que sea más fácil para los equipos de desarrollo manipular y entender el comportamiento de los microservicios simulados, especialmente si estos equipos no están familiarizados con la aplicación.

Otro posible avance sería la implementación de funcionalidades adicionales en base a las necesidades emergentes de los equipos de desarrollo. Por ejemplo, podría ser útil proporcionar más opciones de configuración para las respuestas de los microservicios simulados, permitiendo a los desarrolladores simular situaciones más variadas y específicas como podría ser validar también los *query params* de las llamadas.

Estas son solo algunas de las muchas direcciones que podría tomar la evolución de este proyecto, y cada una de ellas contribuiría significativamente a su valor y eficacia como herramienta de desarrollo y prueba.

5.3. Experiencia personal

Finalmente, a nivel personal, este proyecto ha sido una experiencia enriquecedora e invaluable. No solo he adquirido y profundizado conocimientos técnicos sobre las tecnologías y metodologías utilizadas, sino que también he aprendido mucho sobre trabajo en equipo, gestión de proyectos y comunicación efectiva. Sin duda, las habilidades y experiencias adquiridas durante el desarrollo de este proyecto serán de gran utilidad en mi futura carrera profesional.

Referencias

[1] "Ventajas y desventajas de la metodología scrum," 2023. [Online]. Available: https://blog.wearedrew.co/productividad/-ventajas-y-desventajas-de-la-metodologia-scrum

[2] N. D. Team, "Ntt Data About us," 2023. [Online]. Available: https://www.nttdata.com/global/en/about-us

[3] V. Driessen, "A successful Git branching model," *nvie.com*, 2010. [Online]. Available: https://nvie.com/posts/a-successful-git-branching-model/

[4] "Comparing Workflows: Gitflow workflow," 2023. [Online]. Available: https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow

[5] "Jenkins: Build great things at any scale," 2023. [Online]. Available: https://www.jenkins.io/

[6] "Continuous code quality, sonarqube," 2023. [Online]. Available: https://www.sonarqube.org/

[7] "Swagger: Simplify api development for users, teams, and enterprises," 2023. [Online]. Available: https://swagger.io/

[8] "Systems and software quality requirements and evaluation (square) - system and software quality models," 2011. [Online]. Available: https://www.iso.org/standard/35733.html

[9] "grpc," 2023. [Online]. Available: https://grpc.io/

[10] "Graphql: A query language for your api," 2023. [Online]. Available: https://graphql.org/