Desarrollo y certificación de un Sistema Web basado en Microservicios

Yeison David Ballesteros Toro

Ingeniería de Sistemas

Asesor

Gustavo Andres Marin Lopera

Universidad de Antioquia

Facultad de Ingeniería

Ingeniería de Sistemas

Medellín

2023

Cita	Ballesteros Toro [1]
Referencia Estilo IEEE (2020)	[1] Y. D. Ballesteros Toro, “Desarrollo y certificación de un Sistema Web basado en Microservicios”, Semestre de Industria, Ingeniería en Sistemas, Universidad de Antioquia, Ciudad Universitaria de Medellín, 2023.

Agradecimientos

En el proceso de semestre de industria he recibido el apoyo y colaboración de una institución y varias personas a las cuales deseo expresar mi agradecimiento.

Quiero empezar agradeciendo a la empresa que me brindó la oportunidad de realizar este proyecto y además de empezar mi vida profesional, TCS.

También extiendo este agradecimiento a Laura Toro Mondragón, quien es mí jefe directa en TCS y quien me ayudó en todo este proceso, quien como mí asesora externa me brindaba todas las ayudas y facilidades dentro de la empresa para poder realizar mis labores académicas y laborales, su apoyo, consejos y atención fueron muy valiosos para poder realizar este trabajo.

A mí profesor Gustavo Andrés Marín Lopera, que como tutor me guió durante todo el proyecto y estuvo siempre atento para resolver cualquier inquietud que tuviera, le agradezco mucho porque sin su orientación sé que no habría podido realizar bien este y todos los trabajos realizados durante todo el proyecto.

Además, agradezco a mi familia y amigos por su paciencia y apoyo a lo largo de este proceso. Su motivación me brindó toda la energía suficiente para poder avanzar.

CONTENIDO

LISTA DE FIGURAS

SIGLAS, ACRÓNIMOS Y ABREVIATURAS

IEEE	Institute of Electrical and Electronics Engineers
VSTS.	Visual Studio Team Service
UdeA	Universidad de Antioquia
CRUD	Create, Read, Update, Delete
IT	Tecnología de la información

RESUMEN

En este trabajo se describe el proceso de desarrollo de un sistema web basado en microservicios con comunicación a través de un broker de mensajería. Entre los objetivos está diseñar la arquitectura distribuida, implementar los microservicios utilizando Spring Boot y realizar pruebas funcionales y no funcionales para garantizar la calidad.

La implementación se organiza en fases, comenzando con la definición de requisitos y el contexto de la página web. Se destacan los requisitos funcionales y no funcionales, así como el diseño de la arquitectura de microservicio. Se utiliza PostgreSQL en Amazon RDS como base de datos y se mencionan las tecnologías y librerías utilizadas, incluyendo Spring Boot, R2DBC y RabbitMQ.

El desarrollo sigue un proceso similar en dos microservicios: uno llamado Reglas y el otro Roles y Permisos. Se describen los archivos de configuración de cada ambiente en el que se va a desplegar el aplicativo, las tecnologías utilizadas y la estructura de las carpetas. Se destacan las pruebas unitarias, de aceptación y de performance, realizadas con JUnit, Karate y JMeter respectivamente, así como las pruebas funcionales (E2E) realizadas con Serenity y el patrón de diseño Screenplay.

Por último, también se describe la etapa de despliegue entre ambientes utilizando Docker y Kubernetes, seguido de la ejecución de las pruebas en cada ambiente y la adopción de prácticas de DevOps, incluyendo pipelines de CI/CD y Release Management.

Palabras clave **— Microservicios, Spring Boot, Desarrollo, Java, Screenplay, Rabbit MQ, AWS.**

ABSTRACT

This paper describes the process of developing a web system based on microservices with communication through a messaging broker. The objectives include designing the distributed architecture, deploying the microservices using Spring Boot, and performing functional and non-functional tests to ensure quality.

The implementation is organized in phases, starting with the definition of requirements and the context of the web page. Functional and non-functional requirements are highlighted, as well as the design of the microservice architecture. It uses PostgreSQL from Amazon RDS as the database and the technologies and libraries used, including Spring Boot, R2DBC, and RabbitMQ, are mentioned.

Development follows a similar process in two microservices: one called Reglas and the other Roles y Permisos. The configuration files of each environment in which the application is to be deployed, the technologies used, and the structure of the folders are described. The unit, acceptance, and performance tests, carried out with JUnit, Karate and JMeter respectively are described, as well as the functional tests (E2E) carried out with Serenity and the Screenplay design pattern.

At the end, it also describes the cross-environment deployment stage using Docker and Kubernetes, followed by the execution of the tests in each environment and the adoption of DevOps practices, including CI/CD and Release Management pipelines.

Keywords — **Microservices, Spring Boot, Development, Java, Screenplay, Rabbit MQ, AWS.**

I. INTRODUCCIÓN

Este documento es sobre una práctica profesional, realizada para la empresa Tata Consultancy Services, que es una organización de servicios de Tecnología de la información, consultoría y soluciones comerciales. En este caso, se está prestando apoyo en servicios de TI para un ámbito bancario en el cual se forma parte de un equipo conformado por diez personas: seis personas que conforman el equipo técnico, un líder de personal, un dueño de producto y dos personas que integran el equipo de análisis de datos. En lo personal, hago parte del equipo técnico, entre mis roles están el ser analista de certificación y desarrollador de software.

La necesidad que se tiene como equipo es el desarrollo de un módulo administrativo o página web en la que se controle la gestión de permisos de consumo de microservicios. Es decir, se tienen un conjunto de microservicios, en los que se debe controlar quién tiene o no permitido el consumo ya sea de uno o varios de estos micros, además, aunque se tenga el permiso para consumir uno de los servicios, también es necesario filtrar la información de acuerdo al consumidor(aplicación que realiza peticiones a cualquiera de los servicios), de manera que, cada petición a los microservicios solo se podrá obtener si se tiene el acceso a este y la información que se obtenga debe ser de acuerdo a los intereses del solicitante.

Con la anterior como base, es que se define que es necesaria una arquitectura distribuida basada en microservicios, y aquí se explica el proceso que se siguió para desarrollar completamente este sistema.

Si bien a la hora de desarrollar una aplicación se tienen innumerables formas de hacerlo, ya que existen muchísimos lenguajes de programación, varias metodologías de trabajo, y un sin fin de herramientas de desarrollo. Este documento trata sobre cómo es el proceso de definición e implementación de una de esas tantas maneras de desarrollar un sistema web, que como se mencionaba anteriormente se trata sobre una arquitectura distribuida basada en microservicios con una interfaz o página web.

Se empieza planteando los objetivos generales y específicos para dar una guía hacia donde se quiere llegar y de qué manera, si bien el título del documento es bastante diciente, no nos da muchos detalles de tecnologías o herramientas. Después se tendrá el marco teórico para dar definiciones y conceptos claves que ayuden al entendimiento técnico de lo realizado en el

proyecto, aquí se encuentran las tecnologías, herramientas y estrategias de desarrollo que se usaron para dar solución a la necesidad.

Ya finalizando estará el apartado de conclusiones a las que se llegó después de practicar una metodología de desarrollo ágil, de implementar un microservicio completamente reactivo y si se el aplicativo desarrollado consiguió mejorar o no la gestión de los servicios que ofrece el equipo, dado que la idea es analizar el rendimiento del equipo para administrar los recursos de software mediante el módulo administrativo comparado con el procedimiento manual que se seguía antes de este. No está de más adelantar que el aprendizaje fue muy grande y seguir un flujo completo de desarrollo y pruebas fue bastante enriquecedor.

II. OBJETIVOS

A. Objetivo general

- Desarrollar y probar un sistema Web basado en Microservicios, usando un broker de mensajería para la comunicación entre estos.

B. Objetivos específicos

- Diseñar la arquitectura distribuida del sistema web, definiendo los microservicios y su interacción.
- Implementar los microservicios usando el framework de diseño web Spring Boot.
- Diseñar una estrategia y alcance de pruebas que permita velar por la calidad del desarrollo implementado.
- Realizar pruebas funcionales y no funcionales para verificar la funcionalidad, usabilidad y capacidad del sistema en diferentes escenarios.
- Implementar comunicación asíncrona mediante el uso de un RabbitMQ entre los microservicios.
- Lograr la entrega de resultados a corto plazo, con entregas constantes y de alta calidad, siguiendo los principios de las metodologías ágiles.
- Mantener una constante comunicación con el equipo de trabajo, que sea clara, directa y efectiva.

III. MARCO TEÓRICO

Un estilo de arquitectura para el desarrollo de software en el cual una aplicación se separa en un conjunto de pequeños servicios es llamado microservicios. Cada microservicio se ejecuta de manera independiente de los demás, por lo cual tienen independencia en los despliegues e implementación. Este desacople de funcionalidad en las operaciones de una aplicación, permite mayor facilidad para mantenimiento, dado que no ocurre como en las arquitecturas monolíticas que todo el código está junto, por lo que al aplicar una nueva funcionalidad o al hacer mantenimiento en alguna parte del código, implica el redespliegue de toda la aplicación, aumentando así el riesgo de obtener errores [1]. Esta división de componentes permite que el trabajo en paralelo sea sencillo al estar segregado el proyecto.

La topología o la estructura de despliegue y comunicación entre los microservicios es algo también a definir a la hora de trabajar bajo este modelo de desarrollo. Existen tres topologías principales que son: basada en API REST, basada en aplicaciones REST y centralizada de mensajería. La que es basada en API REST, consiste resumidamente en que cada microservicio expone la información a través de Apis, y los demás acceden a esta mediante una interfaz basada en REST, siendo así aplicaciones síncronas, dado que se solicita a un servicio la información y se debe esperar hasta que la respuesta llegue para poder continuar con la ejecución. Las aplicaciones REST consisten en obtener la información, pero desde pantallas de aplicaciones web. Y en la topología centralizada de mensajería se obtiene la información desde un intermediario de mensajes centralizado [1], permitiendo con esto el manejo de sistemas reactivos, dado que es un sistema de mensajería asíncrona. Esta última es la topología elegida por el equipo en vista de los intereses por desarrollar un código no bloqueante.

A nivel tecnológico existe un framework que facilita la creación de microservicios, llamado Spring Boot, el cual se basa en el lenguaje de programación Java y es bien aceptado en el mundo empresarial por ser de código abierto, es de fácil configuración, incorporación de un servidor embebido, administración de dependencias y proporciona herramientas de métricas y monitoreo. Sin embargo, este framework por sí solo no es suficiente para conseguir flujos reactivos o asíncronos, por lo que se opta por el uso de Spring Webflux, la cual es la librería para el desarrollo de código no bloqueante en Spring Boot, y además usa como servidor Netty, estos

admiten la concurrencia a través de funcionalidades asíncronas y sin bloqueo, por lo que se pueden desarrollar aplicaciones de alto rendimiento [2].

Pasando al frontend del proyecto, nos encontramos con Angular, un marco de trabajo desarrollado en typescript (lenguaje de programación tipado, muy similar a Java en cuanto a estructura de código) el cual sirve para crear aplicaciones web, esta tecnología provee una colección de librerías que facilitan al programador la tarea de crear un sitio web, ahorrando tareas tales como el enrutamiento, la administración de formularios, la comunicación cliente-servidor, entre otras.

En cuanto a las operaciones DevOps, se hace uso de Microsoft Azure DevOps que es una plataforma de servicios en la nube la cual brinda herramientas y servicios para facilitar y agilizar el desarrollo, la entrega y la operación de software. Esta herramienta se integra con Visual Studio Team Services(VSTS) y permite gestionar todo el ciclo de vida de las aplicaciones, comenzando a abarcar desde las decisiones y gestión del proyecto, pasando por el desarrollo o el proceso iterativo dado por la metodología ágil SCRUM, hasta cubrir el trabajo necesario para ejecutar la aplicación, en otras palabras, se cubre todo el desarrollo del proyecto(definición de requisitos, arquitectura, implementación, pruebas, despliegues, etc) y también la gestión y el versionado del mismo[3].

IV. IMPLEMENTACIÓN

Si bien la entrega del proyecto se hacía en pequeñas entregas, tal como lo indica la metodología ágil usada, y además los miembros del equipo trabajaban en paralelo en diferentes partes del sistema (o incluso en tareas ajenas al mismo), no es esto impedimento para seguir con las etapas normales en el desarrollo de software. Por lo que se realiza este apartado justamente separado por cada etapa del ciclo de vida de desarrollo de software:

I. Definición de Requisitos

Contexto:

- Se requiere el desarrollo de un Módulo Administrativo, el cual permita la gestión y/o administración de las reglas de consumo de los microservicios del equipo.
- Las reglas de consumo son por ejemplo: Se tienen 4 microservicios; A, B, C y D, los cuales son utilizados por otros equipos al interior de la empresa (ejemplo: equipo 1 y equipo 2), por lo cual es necesario desarrollar un módulo, mediante el cual se puedan gestionar (crear, editar, eliminar y listar) las reglas que va a tener cada equipo consumidor en cuanto a cada microservicio. Digamos, se requiere que el equipo 1 tenga acceso a los microservicios A y D, pero no a los servicios B y C, además, se necesita que el equipo 1, al consumir el servicio A, únicamente reciba información que es necesaria para este equipo, no toda la información. Esto mismo se puede hacer con el equipo 2 y los demás microservicios.
- Este módulo sólo podrá ser utilizado por los líderes del equipo.
- Al realizar algún cambio en alguna de las reglas, se debe emitir un evento informando a los microservicios restantes.

Requisitos Funcionales:

- Inicio de sesión: Los usuarios identificados cómo líderes, deben poder iniciar sesión en el sistema mediante su usuario y contraseña.
- Manejo de roles: El sistema deberá permitir la administración y asignación de roles y permisos a los usuarios.
- Administración de reglas: El módulo deberá permitir la creación, eliminación, edición y visualización de las reglas de cada consumidor.

- Administración de consumidores: El módulo deberá permitir la creación, eliminación, edición y visualización de consumidores.
- Administración de microservicios: El módulo deberá permitir la creación, eliminación, edición y visualización de microservicios en el sistema.
- Emisión de eventos: El módulo deberá emitir un evento al crear, editar o eliminar una nueva regla, consumidor o servicio en el sistema.

Requisitos No Funcionales:

- Interfaz de usuario: La interfaz de usuario debe ser intuitiva, fácil de usar y atractiva visualmente para garantizar la satisfacción del usuario.
- Seguridad: El sistema debe permitir el inicio de sesión mediante Azure Active Directory y respetar las indicaciones de seguridad dadas por el área de seguridad de la empresa.
- Rendimiento: El sistema no será muy transaccional, debido a que únicamente será utilizado por los líderes del equipo, sin embargo, se espera un tiempo de respuesta rápido y una carga eficiente, como mínimo 5 transacciones por segundo como pico de rendimiento.

II. Diseño

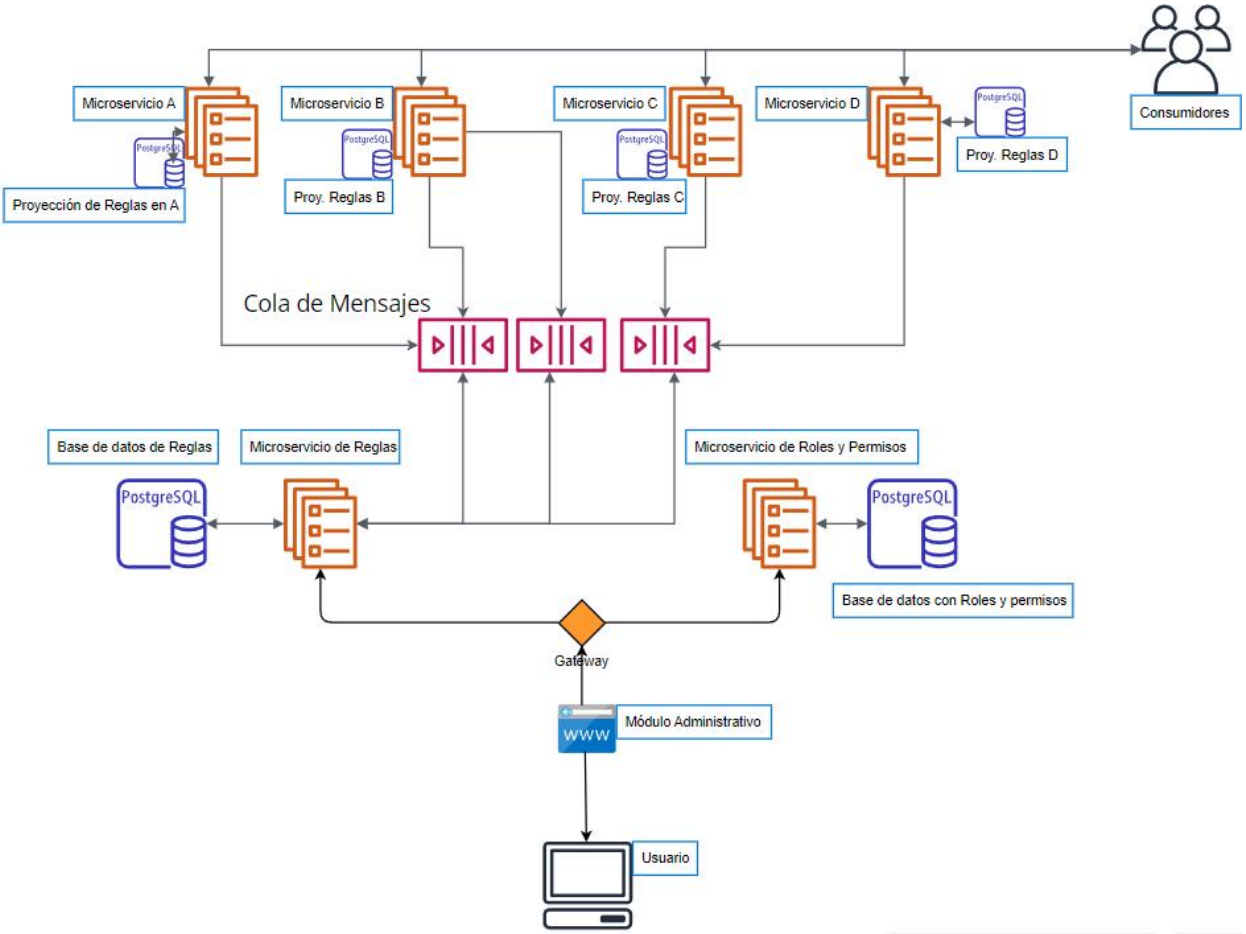

Fig. 1. Diagrama de Arquitectura de Microservicios

Para el diseño, se comienza haciendo un diagrama el cual ilustra los componentes que harán parte del proyecto, es decir, los microservicios, las bases de datos, la página web o el módulo, y el broker de mensajería. Leyendo de abajo hacia arriba se puede observar la comunicación que habrá entre los componentes, comenzando por el Usuario, el cual hará uso del Módulo Administrativo, y sus funciones. La página web a su vez, se conectará a un API Gateway, que estará encargado de direccionar las solicitudes al servicio adecuado. Por ahora se tienen mapeados dos microservicios nuevos, Microservicio de Reglas y Microservicio de Roles y Permisos.

El servicio de Reglas es en el que se podrán gestionar todas las reglas necesarias, así como los consumidores y los microservicios. Mientras que el de Roles y Permisos, es el que se encarga de la gestión de los roles que tendrán los usuarios registrados. Por ejemplo, se requiere que un

usuario únicamente tenga acceso(permisos) a visualizar, pero que no pueda editar, eliminar o crear reglas, entonces es allí donde se tendrá un rol que estará asociado a esos permisos y el cual será asignado al usuario correspondiente. Cada uno de estos dos microservicios contará con sus propios esquemas en la base de datos relacional.

De allí ya podemos ver lo que es el broker de mensajería, en este se tendrán varias colas, cada una con un propósito, una para almacenar los mensajes relacionados a cambios en los consumidores, otra emite cambios en los servicios y la última en las reglas. A partir de allí ya se conectan los servicios que hay actualmente funcionando en el equipo (nombrados A, B, C y D por políticas de privacidad), cada uno de estos se suscribe al broker y recibe los mensajes que vaya a necesitar(no todos van a necesitar todos los mensajes). Una vez reciban algún evento entonces lo que harán será llevar esta nueva información a la proyección que cada uno de los microservicios va a tener de la base de datos original. Se maneja con proyecciones y no todo en la misma base de datos para desacoplar los servicios, así en caso de que alguno esté con alguna falla en algún momento, entonces los demás no se vayan a ver afectados por alguna dependencia faltante. También hay que aclarar, que las proyecciones no son una copia fidedigna de la original, sino una porción de la información, así cada proyección tendrá solo información necesaria por ese microservicio en específico y no estará sobrecargado con datos innecesarios, habilitando con esto consultas más rápidas.

Ya para finalizar con este componente, se tiene la conexión microservicios (A, B, C y D) con los consumidores de estos, pero aquí ya cada microservicio contaría con sus propias reglas y podría filtrar esas consultas hechas por estos otros aplicativos.

En esta etapa del proyecto también se diseñó todo el esquema de la base de datos para los dos microservicios (Reglas y Roles), en el caso de Reglas, se tienen cuatro entidades o tablas, una para el consumidor (nombre e identificador), otra para el microservicio (nombre e identificador), otra para la regla(identificador, consumidor, microservicio) y una última para el filtro(identificador, tipo de filtro, valor del filtro). Además, se realizó el diseño en Figma para la página web con todas las vistas necesarias y los elementos con los que va a interactuar el usuario.

III. Desarrollo

El proceso de desarrollo enunciado a continuación es para el Microservicio de Reglas, sin embargo, para lo que son librerías, archivos de configuración, entidades, casos de uso, base de datos, y controladores rest es un proceso similar en el microservicio de Roles y Permisos.

Se empieza con la definición de las tecnologías a usar, para este caso se elige Spring Boot como framework de desarrollo web y las librerías que se usaron son las siguientes:

```
implementation 'org.springframework.boot:spring-boot-starter-actuator'
implementation 'org.springframework.boot:spring-boot-starter-validation'
implementation 'org.springframework.boot:spring-boot-starter-webflux'

//Repos y bd
implementation 'org.springframework.boot:spring-boot-starter-data-r2dbc'
implementation 'io.r2dbc:r2dbc-pool'
implementation 'org.postgresql:r2dbc-postgresql'
runtimeOnly 'org.postgresql:postgresql'
implementation 'org.postgresql:postgresql'

//Mappers
implementation 'org.modelmapper:modelmapper:3.1.1'
implementation 'org.mapstruct:mapstruct:1.5.3.Final'
annotationProcessor 'org.mapstruct:mapstruct-processor:1.5.3.Final'
compileOnly 'org.projectlombok:lombok-mapstruct-binding:0.2.0'

// Secrets
implementation 'software.amazon.awssdk:secretsmanager:2.20.98'
implementation 'software.amazon.awssdk:sts:2.20.98'
```

Fig. 2. Librerías en Microservicios 1

```
// Utils
implementation 'com.fasterxml.jackson.core:jackson-databind:2.15.2'
compileOnly 'org.projectlombok:lombok:1.18.26'
annotationProcessor 'org.projectlombok:lombok:1.18.26'
testCompileOnly 'org.projectlombok:lombok:1.18.26'
testAnnotationProcessor 'org.projectlombok:lombok:1.18.26'
runtimeOnly 'org.springframework.boot:spring-boot-devtools'

//Tests
testImplementation 'org.springframework.boot:spring-boot-starter-test'
testImplementation 'io.projectreactor:reactor-test'
testImplementation 'org.powermock:powermock-module-junit4:2.0.9'
testImplementation 'org.junit.jupiter:junit-jupiter-api:5.9.2'
testRuntimeOnly 'org.junit.jupiter:junit-jupiter-engine:5.9.2'

//RabbitMq
implementation 'org.springframework.boot:spring-boot-starter-amqp'
```

Fig. 3. Librerías en Microservicios 2

Se eligen las librerías propias de Spring Boot que ofrecen herramientas útiles como métricas del microservicio, análisis de tráfico y monitoreo del estado de conexión a la base de datos. También se opta por el uso de Spring WebFlux para escribir código con flujos de datos asincrónicos, complementado con R2DBC para acceder a bases de datos relacionales de manera no bloqueante.

En vista de que el motor elegido para la base de datos es PostgreSQL en Amazon RDS, entonces se deben usar justamente las tecnologías que permitan la conexión con este motor. También se utiliza el SDK de Secrets Manager para obtener las credenciales de conexión a la base de datos desde el servicio de Secrets Manager de AWS.

Además, se incluyen herramientas de apoyo como Lombok para minimizar la escritura de código, Jackson para la conversión de objetos a JSON y viceversa, y librerías de pruebas como Spring Boot Starter Test y JUnit para automatizar pruebas unitarias. Finalmente, se necesita una librería para la comunicación con el servicio de Amazon MQ de AWS, ya que es justamente este el Broker de Mensajería elegido, específicamente es RabbitMQ.

Una vez definidas las librerías a usar, se procede al encarpetado o arquitectura de los microservicios:

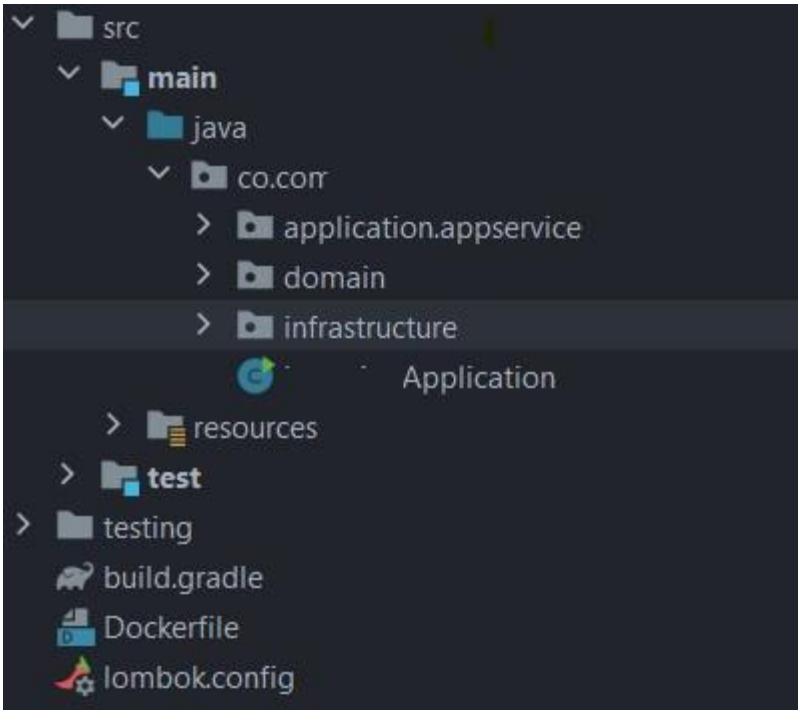

Fig. 4. Arquitectura de Microservicio

En cuanto a la arquitectura definida en el servicio, se optó por una estructura similar a una arquitectura hexagonal, en la que podemos encontrar tres paquetes principales; Paquete de aplicación, encargado de almacenar los casos de uso de nuestro microservicio, paquete de dominio, aquí es donde se ubican las entidades y los repositorios, es decir, tanto lo relacionado al núcleo como a la definición del negocio. También se tiene el paquete de infraestructura, en el que se van a contener todas las tecnologías externas o los puntos por los cuales el exterior va a interactuar con la aplicación, por ejemplo: los controladores REST o la configuración necesaria para la emisión de eventos al RabbitMQ.

También es de destacar el uso de un Dockerfile, ya que el despliegue de los servicios se planea hacer mediante el servicio de Kubernetes de AWS.

Finalmente, con la estructura lista y el proyecto compilado, es posible comenzar con el desarrollo del microservicio:

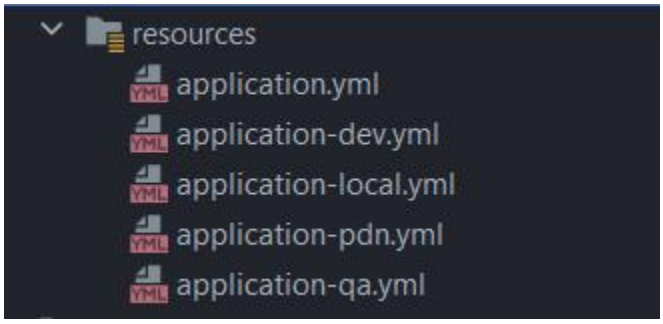

Fig. 5. Archivos de configuración

Para este proyecto se empezó escribiendo los archivos de configuración del entorno, en estos archivos es donde se define el puerto usado en las máquinas para desplegar el servicio, también se definen los endpoints de monitoreo a exponer (dado que hay un archivo por ambiente, entonces se opta por exponer métricas únicamente en desarrollo, y el health para los demás ambientes), además se configura la ruta a los secretos para la base de datos y el broker de mensajería.

A partir de aquí el desarrollo comienza en base al dominio de la aplicación, dado el caso que se comienza con el microservicio de reglas, entonces una de las entidades es justamente la Regla, la cual contiene un identificador único, la asociación de consumidor con microservicio y las restricciones dadas para esa unión. Y así sucesivamente con las demás entidades.

Cuando ya se tengan las entidades listas es posible empezar a crear los mecanismos para el acceso a la base de datos, esto abarca desde la configuración (obtención de los secretos y conexión) hasta crear los beans etiquetados como repositorios, en los que se define el ORM a usar para realizar esa transformación entre las tablas y las entidades. El mismo ORM que se usa, para este caso es un repositorio reactivo (R2DBC), nos provee las operaciones CRUD básicas y además los datos obtenidos vienen en streams reactivos.

Después es posible comenzar con los casos de uso (ubicados en el paquete de application) que se tendrán para cada entidad creada, ya que se trata básicamente de un CRUD (Create, Read, Update, Delete) en las reglas, consumidores y servicios(entidades), entonces son estas las operaciones que se definen para el caso de uso de cada entidad, crear regla, editar regla, eliminar regla, etc. Estos casos de uso harán el llamado al Repositorio correspondiente para poder realizar

las operaciones en la base de datos y a su vez, emitirán el mensaje a la cola de Rabbit con el payload correspondiente (creación, eliminación o edición de algún dato) y a la cola definida.

Ya para finalizar con el desarrollo se realizan las clases que van en el paquete de infraestructura, es posible comenzar con los controladores rest, el cual es el componente del sistema con capacidad de recibir operaciones http y responder a estas. En este mismo paquete se crean las clases de manejo de excepciones, que es posible utilizarlas de forma transversal en el proyecto, ya que una excepción puede ocurrir en cualquier momento de la ejecución.

Dado que este servicio es el emisor(Publisher) de los eventos, entonces lo que seguiría de aquí en adelante es la configuración y adaptación de los servicios receptores, para lograr esto, se comienza creando una proyección de las tablas del servicio de Reglas en estos otros microservicios receptores(entiéndase una proyección como una copia de las tablas originales, pero únicamente con la información que sea útil para este servicio en específico), después se hace la implementación en el código del servicio(se usan las mismas librerías o dependencias del servicio de reglas para RabbitMQ) para recibir mensajes del Broker.

El proceso de transporte de mensajes comienza con el microservicio productor de mensajes, quien configura un topic o exchange, que es el punto de entrada de los mensajes, y a su vez se encarga de enrutar estos a la cola(queue) correspondiente. Una vez los mensajes se encuentran almacenados en una cola de mensajería, esperan allí hasta que un microservicio receptor lo consuma y lo procese (Figura 6). Este proceso, perteneciente al servicio receptor, consiste básicamente en tomar el payload o la información del mensaje y realizar la operación que indica la clave de enrutamiento, ya sea editar, crear o eliminar un dato en las tablas proyectadas del servicio original.

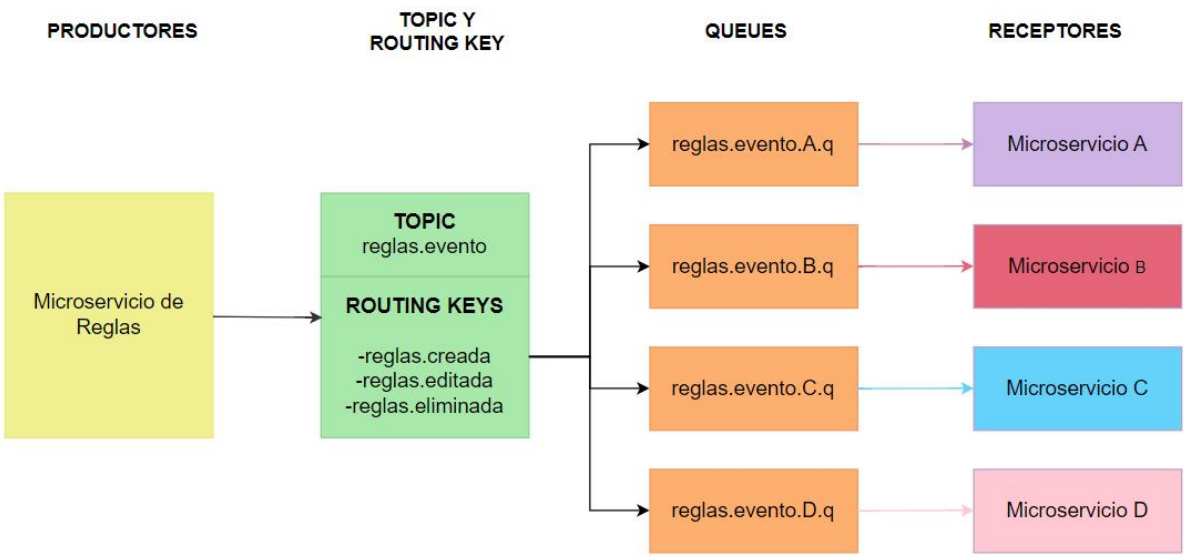

Fig. 6. Diagrama de enrutamiento en Broker de mensajería

IV. Pruebas

Las pruebas realizadas son en base a la pirámide de Cohn [4], en la cual se comienza con las pruebas unitarias, que son las pruebas para probar si determinada parte del código funciona correctamente, después se realizan las pruebas de aceptación, que se encargan de determinar si el servicio está respondiendo correctamente y si se está obteniendo lo que el cliente ha pedido. Luego se realizan las pruebas funcionales, que justamente como su nombre lo indica, se usan para verificar y validar el funcionamiento del servicio, estas pruebas son automatizadas para no tener que estar dedicando mucho tiempo en estas cada que se haga un mantenimiento o soporte a la aplicación.

Por último, se realizan y ejecutan las pruebas de rendimiento, que son pruebas no funcionales y que permiten conocer el comportamiento del sistema bajo cargas de trabajo bajas, medias o extremas.

Si bien esta etapa se encuentra como una de las últimas en la implementación, en realidad es posible trabajar las pruebas desde cualquier punto, dado que son diferentes tipos de pruebas y no todos son realizados por el mismo desarrollador, por ejemplo, las pruebas unitarias pueden ser inmediatamente abarcadas después de cada pequeña implementación en el desarrollo. Las

pruebas funcionales las puede comenzar a realizar el analista de certificación incluso sin el proyecto estar funcionando, en vista de que es un proyecto aparte, de hecho, es de esta manera en cómo se fueron realizando las pruebas, para que en cada Sprint se pudiera tener un avance en el proyecto desde diferentes puntos.

- Pruebas Unitarias:

Para estas pruebas, se hace uso de mocks, los cuales imitan el comportamiento de un objeto del proyecto, evitando el uso del objeto real y controlando los valores de retorno de cada método simulado, así también se evitan dependencias y se aíslan las pruebas unitarias del código original.

A nivel tecnológico se hace uso de Mockito, que es un framework de Mock de Java y de Junit5.

A nivel estructural se usa el patrón AAA o Arrange, Act, Assert, que traduce en comenzar inicializando los objetos (Arrange), después realizar el llamado de los métodos a probar(Act) y finalmente comprobar que el método de prueba se comporta tal cual cómo se espera(Assert).

Se realiza el cubrimiento sobre el 80% del código realizado en el microservicio, ya que por cuestión de lineamientos o estándares se debe cumplir esta medida como mínimo.

- Pruebas de aceptación:

En la definición de los requerimientos se tienen unas expectativas por parte del cliente, por lo cual estas pruebas se centran en verificar que esas necesidades sí se están supliendo. Son ejecutadas después de cada despliegue en un ambiente determinado para así garantizar que el cambio realizado en un aplicativo o la nueva funcionalidad sí cumple con las expectativas.

Estas pruebas se realizaron bajo el framework de Karate, este nos regala un enfoque BDD, o pruebas basadas en el comportamiento, lo cual es ideal para realizar pruebas que unan el negocio con la tecnología, así, es posible definir los tests bajo las necesidades del negocio(requerimientos) e implementarlos tecnológicamente.

El framework usa el lenguaje Gherkin, que usa una sintaxis muy simple y entendible no solo por los desarrolladores y certificadores de calidad sino también por personas gestoras del proyecto:

```
Feature: Probar funcionamiento de microservicio de reglas

  Scenario: Validar la creacion exitosa de una regla
    Given url 'http://localhost:8080/reglas/listar'
    And method get
    And print response
    Then status 200
```

Fig. 7. Ejemplo 1: Prueba aceptación en Karate

En el ejemplo anterior, se puede observar la sintaxis usada por Gherkin. En esta se define un Feature, que es la descripción general de los escenarios de prueba que vendrán a continuación. Cada escenario de prueba se define con la palabra “Scenario” y una breve descripción de la funcionalidad a probar. Después se tiene la lógica del test, el cual es dado mediante una narrativa muy sencilla, Para el caso anterior, es el ejemplo de un consumo a un endpoint de un microservicio, por lo que se necesita la URL y/o el recurso, además del método HTTP, para este caso es un GET, pero bien podría ser un POST, PUT, DELETE, etc. Posteriormente se imprime la respuesta, en caso de querer ver mediante logs lo que se está obteniendo y por último se tienen las condiciones del escenario, puestos en los “Then”, para este breve ejemplo se pregunta si el código de respuesta es un 200, el cual indica un caso de éxito.

- Pruebas de Performance:

Siempre es necesario e indispensable conocer la capacidad de respuesta de un aplicativo, para así saber cuánto puede soportar nuestro sistema. Para obtener esta información son necesarias pruebas de rendimiento.

Estas pruebas se realizaron mediante la herramienta JMeter propiedad de Apache y de código abierto, que es ideal para probar el rendimiento de las aplicaciones web. En si esta herramienta se puede ejecutar de dos maneras; modo CLI o interfaz GUI. Para este proyecto se ejecutan de las dos maneras, primero mediante la interfaz creamos el test plan o el script a ejecutar, este se ubica en una carpeta de Testing al interior del proyecto y una vez el proyecto esté en el pipeline de Release Management se ejecuta este script mediante comandos en un agente.

El script se configura para enviar peticiones a todos los endpoints existentes en la aplicación. También se configuran los datos de prueba a usar, la cantidad de hilos y el periodo de aumento de estos hilos, por ejemplo, queremos simular 300 usuarios pero como es normal, no todos empezarán a enviar solicitudes al sistema al mismo tiempo, sino que irán llegando usuarios de a poco, entonces se definen 300 hilos en total(cantidad de usuarios a simular) y un ramp-up de 100 segundos, lo que traduce en que jmeter irá aumentando los hilos cada segundo hasta que al segundo 100 ya tendrá 300 hilos funcionando.

Cabe destacar que la cantidad de hilos y el periodo de aumento pueden ser variables de acuerdo con el tipo de prueba. Para este proyecto se realizaron pruebas de línea base (10% de la capacidad total del sistema), de carga (100% de la capacidad total del sistema) y estrés (125% de la capacidad total del sistema). Como uno de los requisitos era que el sistema debe soportar 5 transacciones por segundo, entonces para carga son 5 hilos y un segundo de periodo. Para línea base serían 0.5 hilos y un segundo de periodo. Finalmente, para estrés se configuran 6,25 hilos y un segundo de periodo.

- Pruebas Funcionales o E2E:

Estas pruebas ensayan el sistema de principio a fin, analizando toda la funcionalidad del mismo (o al menos, todo lo que sea posible) para garantizar un aplicativo de calidad y funcional. Antes de comenzar con la implementación o escritura del código para estos escenarios, se debió crear un Plan de Pruebas, en el que se hace un contexto del microservicio, una estrategia o descripción general de cómo se va a abarcar la automatización, después se especifica un alcance, en este se definen las validaciones y verificaciones que se van a realizar, y finalmente se describen las limitaciones que se tienen con estas pruebas.

Ahora, en cuanto a la tecnología usada se destaca el uso de Serenity, librería de código abierto para automatizar pruebas funcionales, que trabajan con Java y usa el lenguaje Gherkin, similar a lo visto en las pruebas de aceptación con Karate. En cuanto a la arquitectura se sigue el patrón de diseño Screenplay, que define el proyecto en capas, separando la definición de las pruebas de su implementación.

Dado que estas pruebas son un proyecto totalmente independiente, entonces se debió crear un repositorio propio para estas y los respectivos pipelines de Build y Release.

V. Despliegues

Para la etapa de despliegue entre ambientes se procede a configurar el archivo de Docker. En este se define la imagen base con la que se construye nuestra propia imagen, para este caso es una imagen de Alpine con Java 17, después se tiene la configuración de un health check, que si bien es opcional, es muy útil para monitorear el estado del aplicativo, en dado caso que el healthcheck falle entonces el contenedor será considerado no saludable, de aquí en adelante ya se tienen una variables de entorno, en la cual podemos pasar el perfil de ejecución basado en el ambiente, y así el sistema sabrá que archivo de configuración usar(visto en la Figura 5). Con la instrucción COPY lo que se hace es justamente copiar los archivos JAR de la aplicación a la carpeta bin del contenedor. Ya al final con el comando ENTRYPOINT se especifica el comando que se ejecutará cuando se inicie el contenedor. En este caso, se ejecuta el comando java -jar para iniciar la aplicación. El comando -g 'daemon off;' evita que la aplicación Java se ejecute en segundo plano. Y con EXPOSE se expone el puerto 8080 en el contenedor, permitiendo que la aplicación dentro del mismo pueda recibir tráfico en ese puerto.

```
FROM #{eclipse-temurin_17_alpine}#
HEALTHCHECK CMD curl -f http://localhost/api/rules/management/health || exit 1
ENV PROFILE=#{PROFILE}#
ENV LOG4J_FORMAT_MSG_NO_LOOKUPS=true
COPY --chown=#{eclipse-temurin_11_alpine-user}#:#{eclipse-temurin_11_alpine-user}#  */**/*.jar /bin/
USER #{eclipse-temurin_11_alpine-user}#:#{eclipse-temurin_11_alpine-user}#
ENTRYPOINT java -jar /bin/service-rules-1.0.jar -g 'daemon off;'
EXPOSE 8080
```

Fig. 8. Dockerfile

Para la creación del pipeline de CI CD, se usa la siguiente configuración:

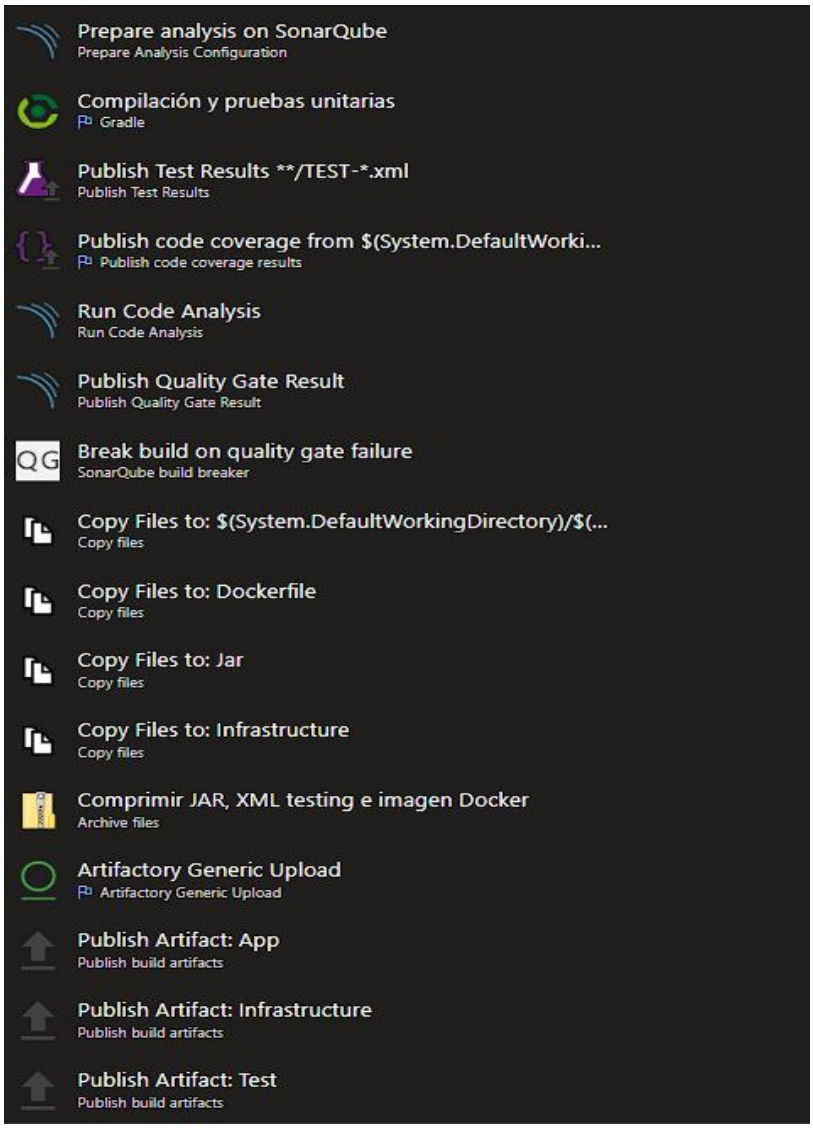

Fig. 9. Pipeline de CI CD

Estas son todas las tareas que se ejecutarán al momento de ejecutar el pipeline de build o compilación, es decir, al momento de crear un Pull Request, intentando unir la rama trunk(o main en muchos proyectos) con la rama feature(en la cual se trabaja cada nueva característica) entonces este pipeline será el encargado de analizar el código estático para ver si se cumple con buenas prácticas y además si se ejecutan correctamente las pruebas unitarias. Una vez se realicen

exitosamente estas validaciones, es posible unir estos nuevos cambios y posteriormente crear los artefactos o paquetes de la aplicación.

Comenzando a analizar cada tarea en esta automatización se empieza por la tarea de preparación de análisis de código estático en sonar, esta tarea lo que hace es asociar el proyecto a un tablero de sonar, en el cual se va a realizar análisis del código, ya sea en busca de malas prácticas de acuerdo con la empresa y en busca de que haya una cobertura de pruebas unitarias superior al 80% del total del código del microservicio.

En segundo, tercer y cuarto lugar está la ejecución de las pruebas unitarias, publicación de los resultados y la publicación de la cobertura de pruebas. En caso de fallar algún escenario de prueba, entonces fallará la ejecución de este pipeline y no se podrá realizar un merge entre la rama feature y la rama trunk (o main).

En la tarea número cinco, seis y siete está todo lo relacionado con el tablero de sonar, aquí ya se realiza el análisis del código estático, la publicación de resultados y un break build, el cual se encarga de romper la ejecución del pipeline cuando no se cumpla con las métricas establecidas.

De aquí en adelante ya se tienen las tareas asociadas a la creación de los artefactos necesarios para el despliegue entre ambientes, se comienza copiando los archivos de testing, de docker, del proyecto y del archivo de definición de infraestructura o definición de kubernetes. Después se comprimen y publican en las rutas especificadas en la tarea correspondiente.

En cuanto ya se tengan los artefactos listos, se procede a configurar el pipeline de Release Management para continuar con la adopción de los pilares DevOps, este pipeline sigue la siguiente configuración:

Fig. 10. Pipeline de Release Management

Se comienza con los stages o etapas de despliegue en el ambiente de desarrollo (primera fila). Comienza con la compilación y creación de la imagen de Docker (se usa el Dockerfile visto antes), después se despliega en el ambiente mediante el servicio de EKS de Amazon, el cual construye los pods de acuerdo con el archivo de infraestructura también encontrado en la raíz del proyecto y al final tenemos las pruebas de aceptación, construidas en Karate y estas permiten saber si el servicio quedó respondiendo correctamente en el ambiente dado.

Este mismo proceso se realiza en el ambiente de certificación o de calidad (segunda fila), sin embargo, después de las pruebas de aceptación se tienen otras pruebas adicionales, como las pruebas de rendimiento, funcionales o E2E y de seguridad (estas no son enunciadas en la fase de pruebas dado que son realizadas por un equipo ajeno al nuestro).

Ya en la última fila tenemos el despliegue en el ambiente productivo, este comienza con la creación de una orden de cambio, la cual es firmada por la líder del equipo y posteriormente la creación y construcción de los pods en base a la imagen de docker. Al final se tienen pruebas de humo y manuales para verificar que el servicio esté funcionando.

Cuando ya se ha ejecutado la totalidad del pipeline, se tiene un servicio funcionando en los ambientes de desarrollo, certificación y producción, consumiendo en cada ambiente una base de datos y un servicio de broker.

V. CONCLUSIONES

- Una de las conclusiones a las que se llega, al haber realizado todo un microservicio reactivo, desde el acceso a base de datos mediante un repositorio reactivo como Spring data r2dbc, hasta el uso de Web Flux para el manejo de flujos reactivos de datos, es que aunque el sistema se realiza completamente asíncrono, a lo mejor no es esta una muy buena práctica, en el sentido de que los repositorios reactivos no permiten el mapeo objeto relacional en tablas relacionadas tan fácilmente como lo hace Spring data JPA, por ejemplo, para realizar una operación de guardado en varias tablas relacionadas, toca construir los objetos referentes a cada entidad que se va a almacenar en la base de datos y usar el método que provee la interfaz para guardar cada entidad por separado, mientras que con JPA al estar relacionadas entonces únicamente se crea el objeto completo(tiene las otras entidades como atributos) y se usa un único método de guardado. Claramente esto agilizaría el desarrollo y hasta el entendimiento del código.
- En el marco de trabajo ágil SCRUM que se usó para el desarrollo de este proyecto, se definieron intervalos de 14 días, es decir, cada Sprint comenzaba con la planeación de este, y a los 14 días se daban cuentas de lo que se logró o alcanzó a hacer de lo que se había planeado, esto en una de las ceremonias llamada Review. De esta manera Sprint tras Sprint se iba logrando un avance, en el cual, aunque no siempre se haya generado valor (realizar algún paso a producción, sea de una nueva funcionalidad o de una aplicación completamente nueva), pero sí se evidenciaba un avance. También fue posible el paralelismo por los integrantes del equipo, dado que mientras alguien se encargaba de desarrollar una aplicación, algún otro se encargaba de otra aplicación, mientras alguien más diseñaba un plan de pruebas, y así cada integrante aportaba algo. Por lo tanto, esta metodología resultó en una gran herramienta, dado que posibilitó las entregas de valor constante, la adaptación al cambio y el trabajo en equipo.
- La gestión de las reglas de un consumidor se realizaba mediante otro servicio, realizado en python y la forma de hacer las inserciones o modificaciones en los datos, se realizaba mediante una automatización, la cual recibía un query de base de datos y lo ejecutaba en el ambiente que se eligiera, ya sea desarrollo, certificación o producción (o los tres). Por lo que se pasó de crear un query por cada ajuste, realizar un Pull Request para subir este

nuevo cambio a un repositorio y posteriormente ejecutar un pipeline que llevaría los cambios entre las bases de datos, a simplemente abrir una página web y mediante clicks elegir los cambios. Esto finalmente llevó a liberar gran capacidad del equipo, ya que usualmente esta tarea era realizada por un integrante elegido al azar y tomaba varios días del sprint. Ahora no lleva sino un par de minutos y cualquiera puede hacerlo.

- En cuanto a los objetivos planteados, se consigue la implementación de una arquitectura distribuida, basada en microservicios, los cuales fueron totalmente desarrollados con Spring Boot y Spring WebFlux, permitiendo así un flujo de datos reactivo. Además, se hace uso de un Broker de mensajería, específicamente Rabbit MQ, alojado en AWS para una comunicación asíncrona entre los emisores y receptores, evitando así dependencias directas. También se consigue aplicar todo el flujo de pruebas de la pirámide de Cohn: Pruebas Unitarias, Pruebas de Aceptación, Pruebas de Rendimiento y Pruebas E2E. Siendo estas últimas previamente diseñadas y planificadas en un Plan de Pruebas, el cual describe totalmente el alcance de estas. Claramente todo esto se consiguió mediante un trabajo en equipo bien definido, gracias a la metodología ágil SCRUM, la cual mediante ceremonias o reuniones constantes permitía organizar toda la gestión del proyecto.

REFERENCIAS

[1] Maya, E., y López, D. (2017). Arquitectura de Software basada en Microservicios para Desarrollo de Aplicaciones Web. Septima Conferencia de Directores de Tecnologia de Informacion 2017, July, 2–6.

[2] Rakshith, R. R., & Swamy, S. R. (2020). Review on Spring Boot and Spring Webflux for Reactive Web Development. *International Research Journal of Engineering and Technology (IRJET)*, *0*(0).

[3] Rodríguez, A. (2016). Servicios en la Nube con Microsoft Azure: Desarrollo y Operación de una aplicación Android con DevOps. Oa.Upm.Es, 22–23.

[4] Cubas M, Rafael. (2017). *Testing y Calidad de Software. Automatización de pruebas con Selenium WebDriver*. Proyecto Fin de Carrera / Trabajo Fin de Grado, E.T.S.I. y Sistemas de Telecomunicación (UPM), Madrid, 23.